

## Module-2

### Combinational Logic

#### Syllabus:

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

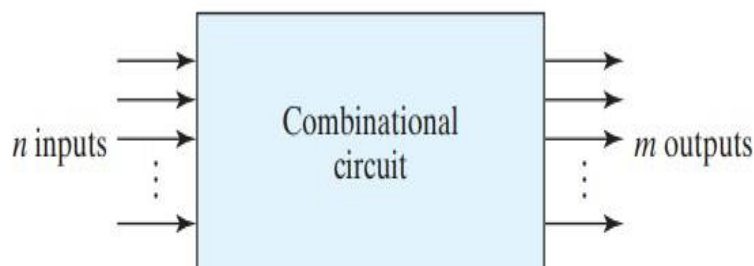
#### Introduction

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of logic gates whose outputs at any time are determined from only the **present combination of inputs**.
- A combinational circuit performs an operation that can be specified logically by a set of Boolean functions.
- sequential circuits employ storage elements in addition to logic gates. Their outputs are a function of the inputs and the state of the storage elements
- Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on **present** values of inputs, but also on **past inputs**, and the circuit behavior must be specified by a time sequence of inputs and internal states.

#### combinational circuit

A combinational circuit consists of an interconnection of logic gates.

- Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, **transforming binary** information from the given input data to a required output data.



**FIGURE 4.1**  
Block diagram of combinational circuit

The  $n$  input binary variables come from an external source; the  $m$  output variables are produced by the internal combinational logic circuit and go to an external destination.

Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0.

- If the **registers** are included with the combinational gates, then the total circuit must be considered to be a **sequential circuit**.
- For  **$n$  input variables**, there are  **$2^n$  possible combinations** of the **binary inputs**.
- For each possible input combination, there is one possible value for each output variable. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.

### Design procedure

The procedure to design combinational circuit involves the following steps:

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

### Example for design Procedure

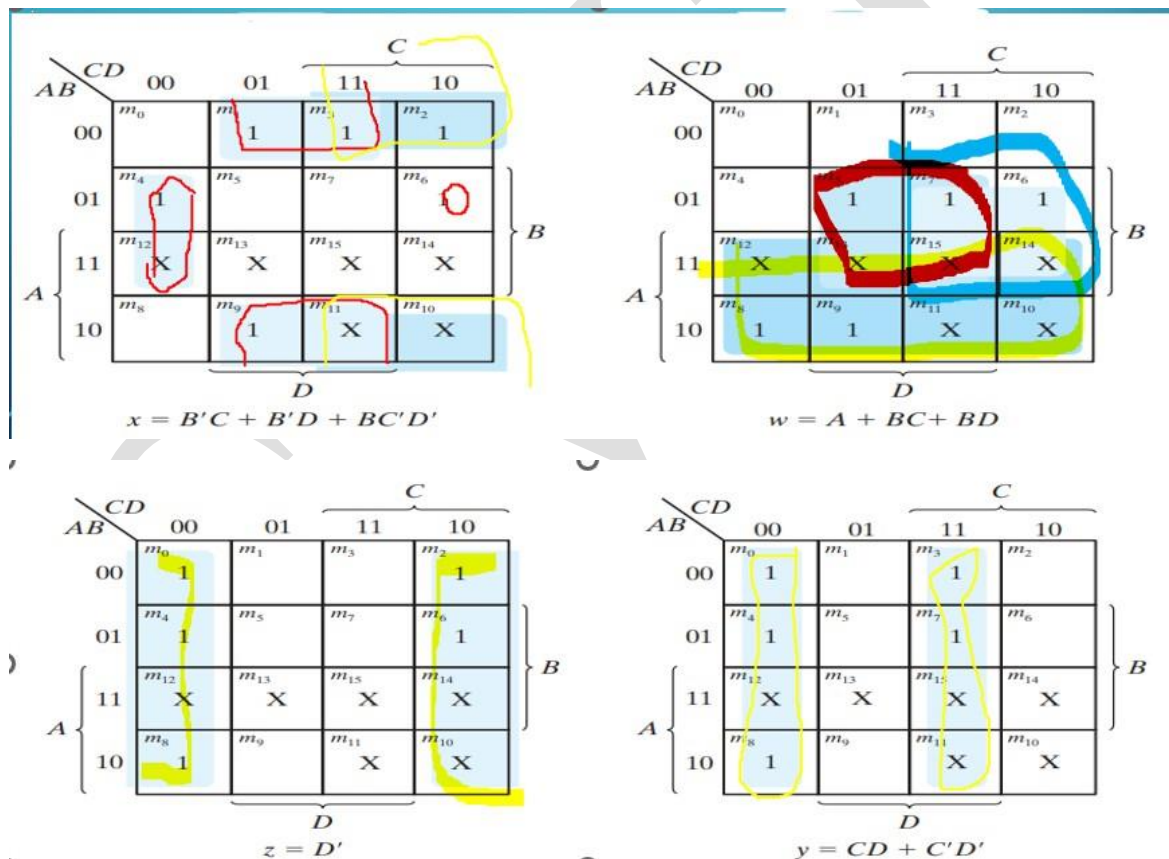
#### Code Conversion (Convert BCD to Excess-3 Code)

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

**Table 4.2**  
**Truth Table for Code Conversion Example**

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

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



implemented with three or more levels of gates:

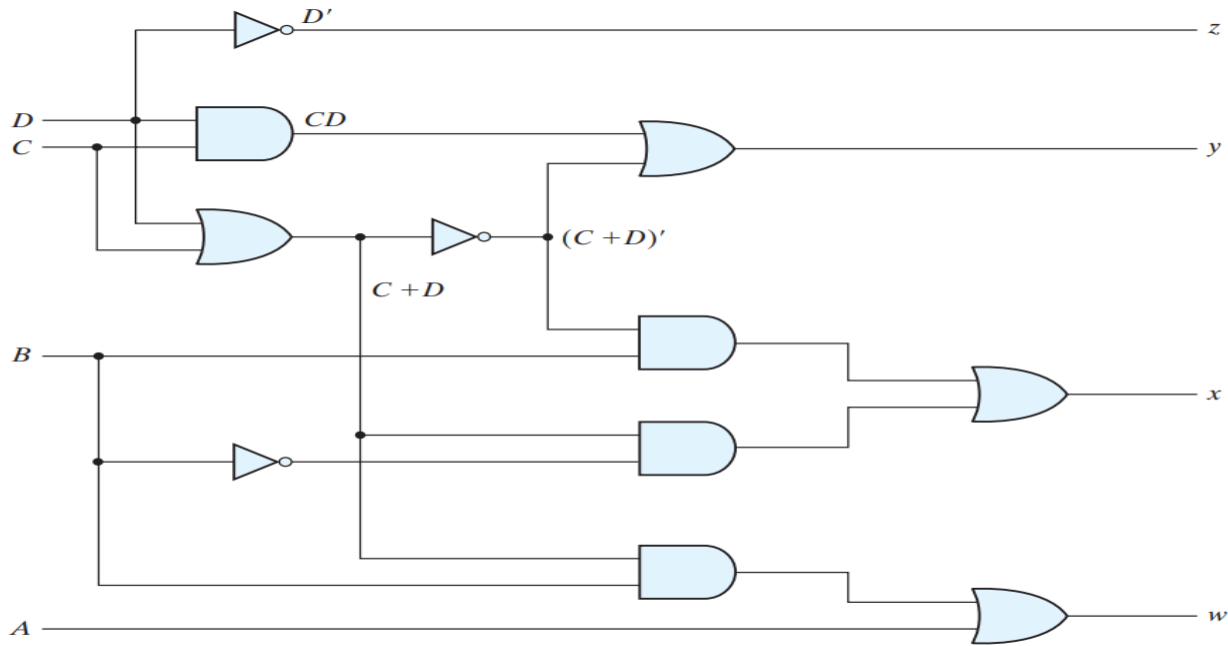
$$z = D'$$

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

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

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

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



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

## Binary Adder- Subtractor

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

### Half Adder

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

**Table 4.3**  
**Half Adder**

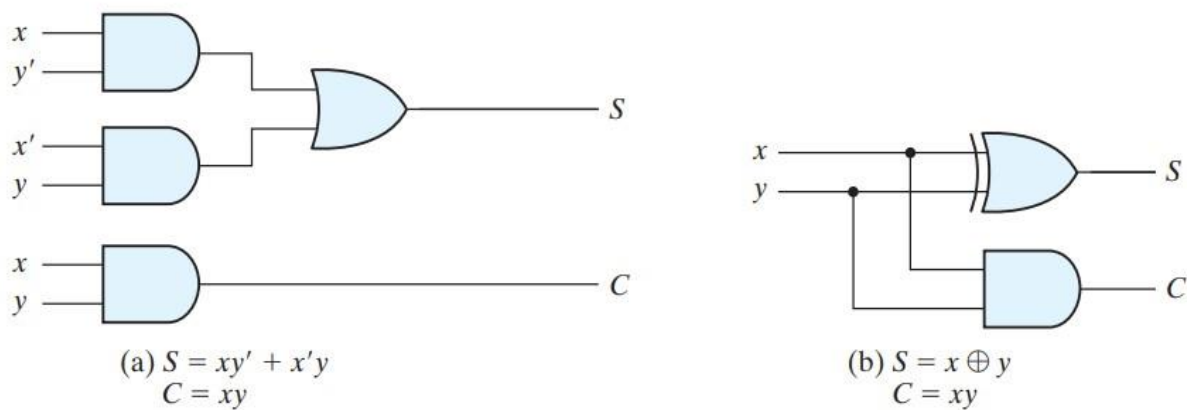
$x$	$y$	$C$	$S$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

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

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

$$C = xy$$

The logic diagram of the half adder implemented in sum of products is shown in Fig. 4.5(a) . It can be also implemented with an exclusive-OR and an AND gate as shown in Fig. 4.5(b)



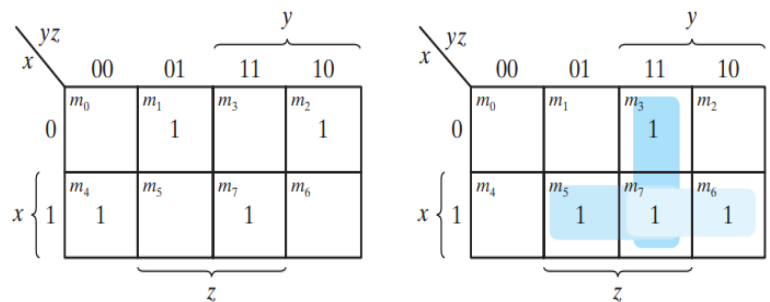
**FIGURE 4.5**  
**Implementation of half adder**

### Full Adder

- A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs.
- Two of the input variables, denoted by  $x$  and  $y$  , represent the two significant bits to be added. The third input,  $z$  , represents the carry from the previous lower significant position. The two outputs are designated by the symbols  $S$  for sum and  $C$  for carry.

**Table 4.4**  
Full Adder

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



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

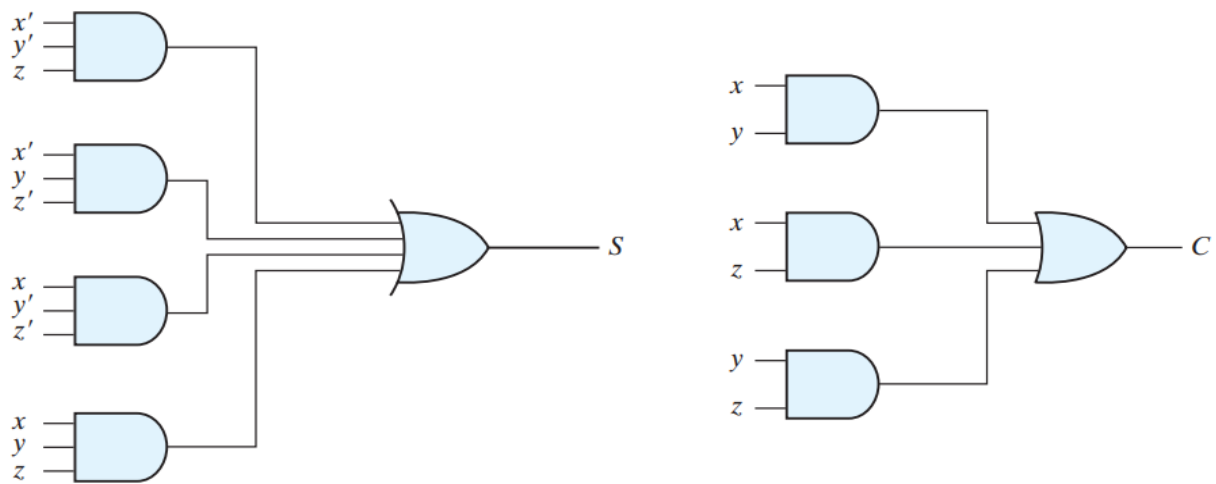
(b)  $C = xy + xz + yz$

**FIGURE 4.6**  
K-Maps for full adder

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

$$C = xy + xz + yz$$

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



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

### Implementation of Full adder using 2 half adder :

We know that

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

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

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

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

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

$$= z \oplus A$$

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

$$C = xy + xz + yz$$

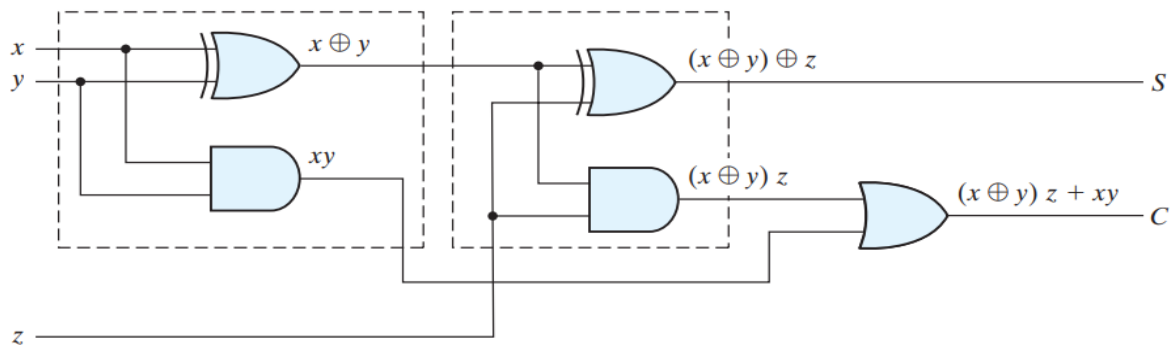
$$= xy + xz(y + y') + yz(x + x')$$

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

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

$$= xy + xyz + z(x \oplus y)$$

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



**FIGURE 4.8**

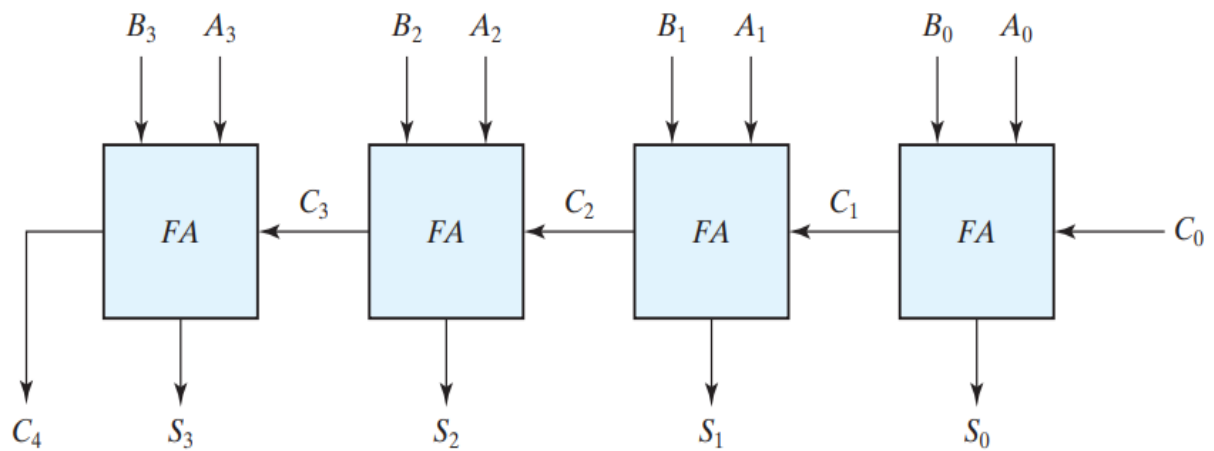
Implementation of full adder with two half adders and an OR gate

### Binary Adder:

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

- n-bit numbers requires a chain of n full adders or a chain of one-half adder and n-1 full adders.
- Eg:4bit numbers requires a chain of 4 fulladders or one HA and 3FAs.
- interconnection of **four full-adder (FA)** circuits to provide a **four-bit binary ripple carry adder**
- The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit. The carries are

connected in a chain through the full adders. The input carry to the adder is  $C_0$ , and it ripples through the full adders to the output carry  $C_4$ .



**FIGURE 4.9**  
Four-bit adder

To demonstrate with a specific example, consider the two binary numbers  $A = 1011$  and  $B = 0011$ . Their sum  $S = 1110$  is formed with the four-bit adder as follows:

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

The **input carry  $C_0$**  in the least significant position must be **0**.

The value of  $C_{i+1}$  in a given significant position is the output carry of the full adder.

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



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

```

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

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

```

2. Write Verilog code for 4 bit adder .

HDL (Dataflow: Four-Bit Adder)

```

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

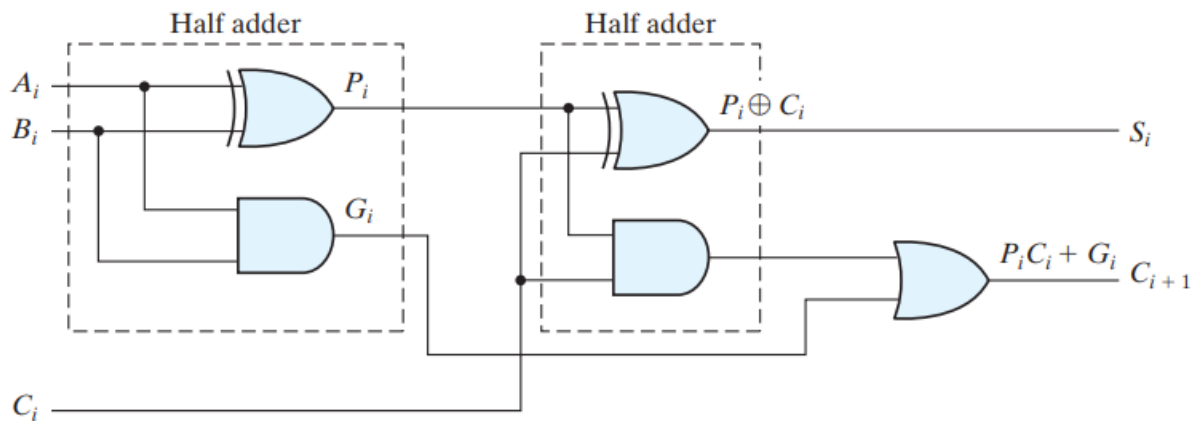
```

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

#### Carry Propagation

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

- $G_i$  is called a carry generate, and it produces a carry of 1 when both  $A_i$  and  $B_i$  are 1, regardless of the input carry  $C_i$ .
- $P_i$  is called a carry propagate, because it determines whether a carry into stage  $i$  will propagate into stage  $i + 1$ .



**FIGURE 4.10**

Full adder with  $P$  and  $G$  shown

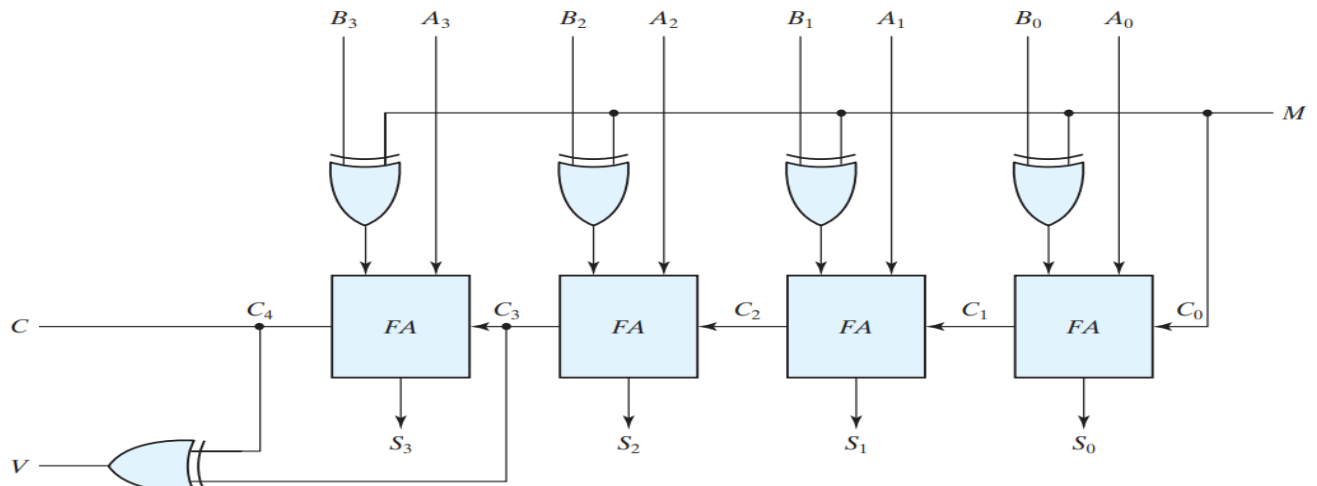
- $P_i = A_i \oplus B_i$
- $G_i = A_i B_i$
- $S_i = P_i \oplus C_i$
- $C_{i+1} = G_i + P_i C_i$

### Binary ADDER-Subtractor

The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder.

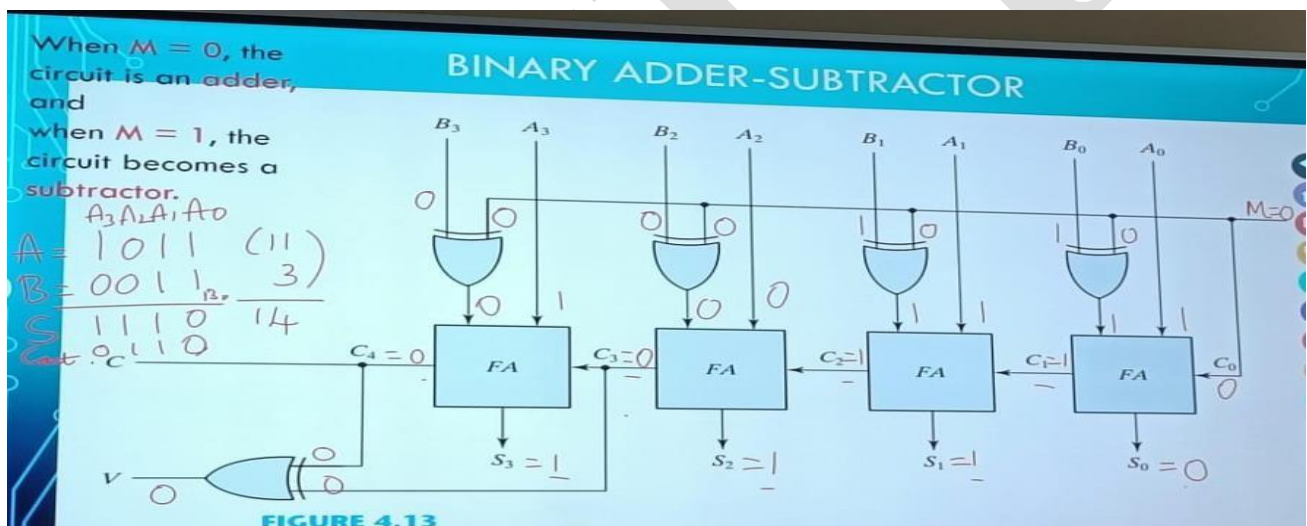
A four-bit adder-subtractor circuit is shown in Fig. 4.13. The mode input  $M$  controls the operation. **When  $M = 0$ , the circuit is an adder**, and when  **$M = 1$ , the circuit becomes a subtractor**. Each exclusive-OR gate receives input  $M$  and one of the inputs of  $B$ .

When  $M = 0$ , we have  $B \oplus 0 = B$ . The full adders receive the value of  $B$ , the input carry is 0, and the circuit performs  $A$  plus  $B$ . When  $M = 1$ , we have  $B \oplus 1 = B'$  and  $C_0 = 1$ . The  $B$  inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation  $A$  plus the 2's complement of  $B$ . (The exclusive-OR with output  $V$  is for detecting an overflow.)

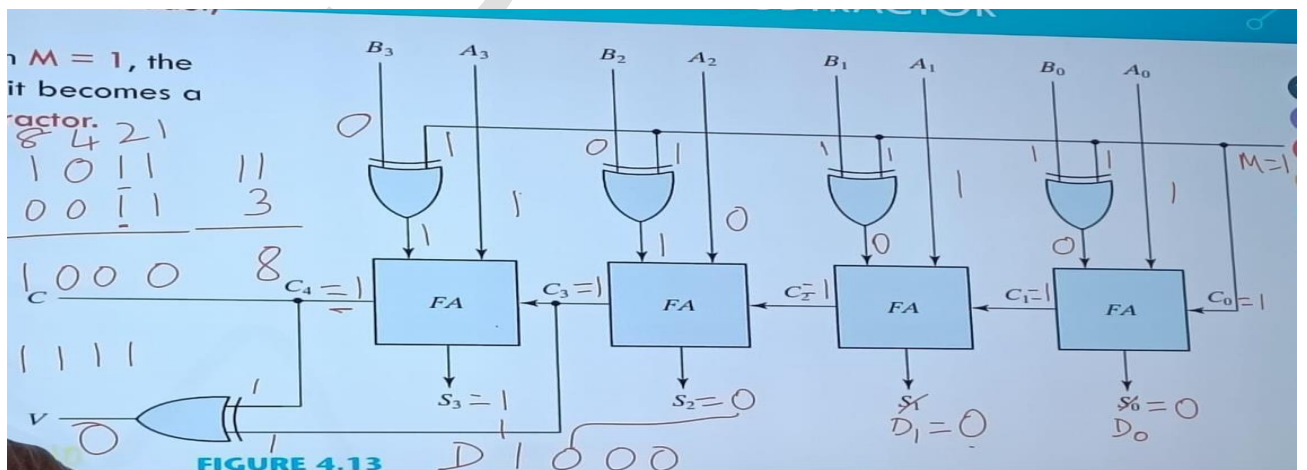


**FIGURE 4.13**  
Four-bit adder-subtractor (with overflow detection)

Binary Addition Example:



Binary Subtraction Example:

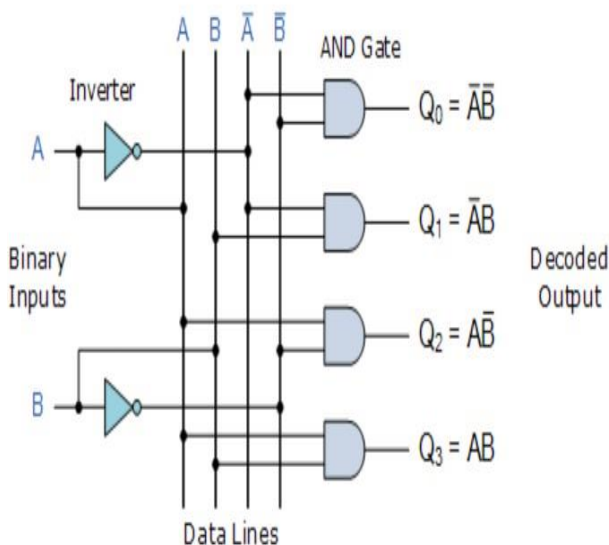
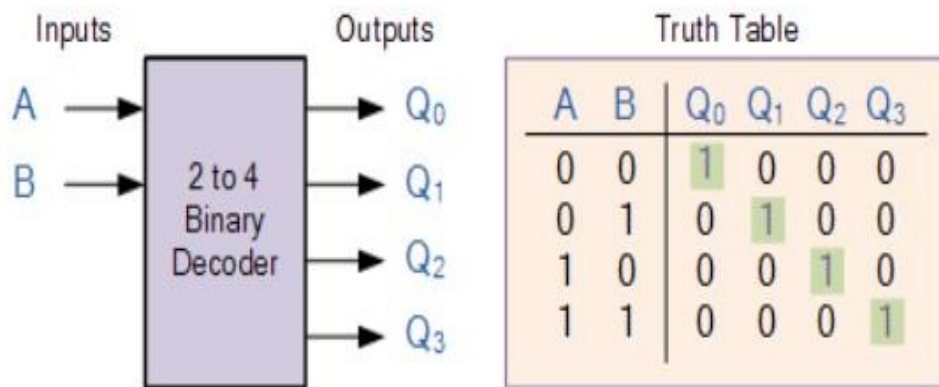


## DECODERS

- A Decoder is a combinational circuit that converts binary information from  $n$  input lines to a maximum of  $2^n$  unique output lines.
- The decoders presented here are called  $n$ -to- $m$ -line decoders, where  $m \leq 2^n$ . Their purpose is to generate the  $2^n$  (or fewer) minterms of  $n$  input variables.
- Each combination of inputs will assert a unique output. The name decoder is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

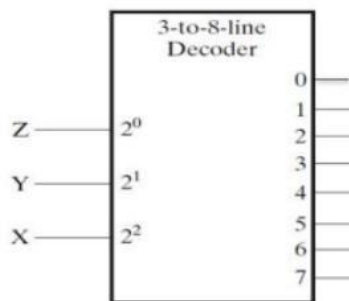
### 2:4 decoder ( 1 of 4 decoder)

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



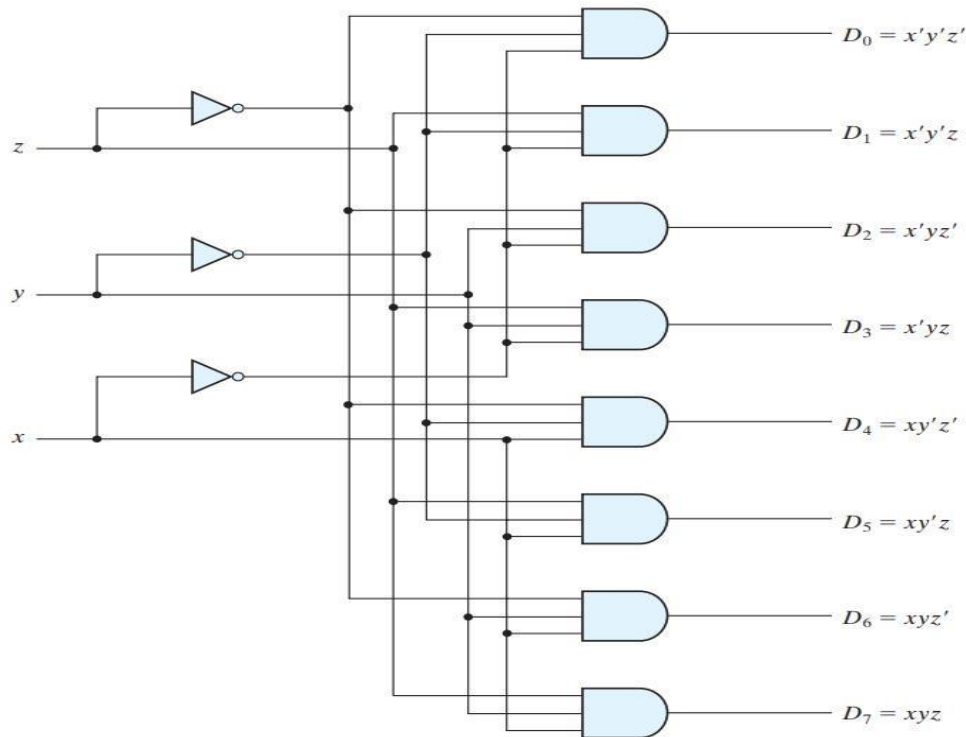
### 3:8 Decoder

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



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

Inputs			Outputs							
<i>x</i>	<i>y</i>	<i>z</i>	<i>D</i> <sub>0</sub>	<i>D</i> <sub>1</sub>	<i>D</i> <sub>2</sub>	<i>D</i> <sub>3</sub>	<i>D</i> <sub>4</sub>	<i>D</i> <sub>5</sub>	<i>D</i> <sub>6</sub>	<i>D</i> <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



**FIGURE 4.18**  
Three-to-eight-line decoder

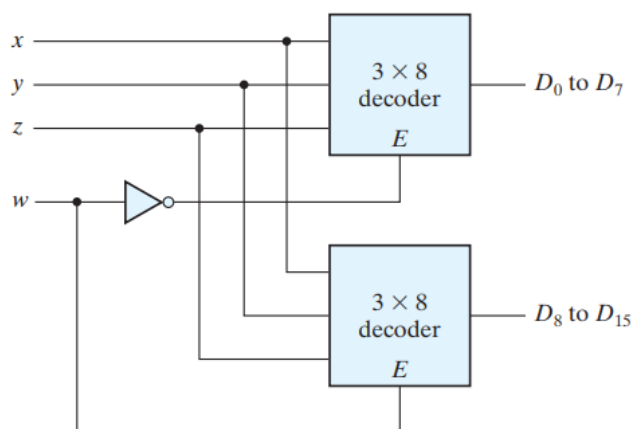
Decoders with enable inputs can be connected together to form a larger decoder circuit.

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

two 3-to-8-line decoders with enable inputs connected to form a 4-to-16-line decoder.

When  $w = 0$ , the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate **minterms 0000 to 0111**.

When  $w = 1$ , the enable conditions are reversed: The bottom decoder outputs generate minterms **1000 to 1111**.



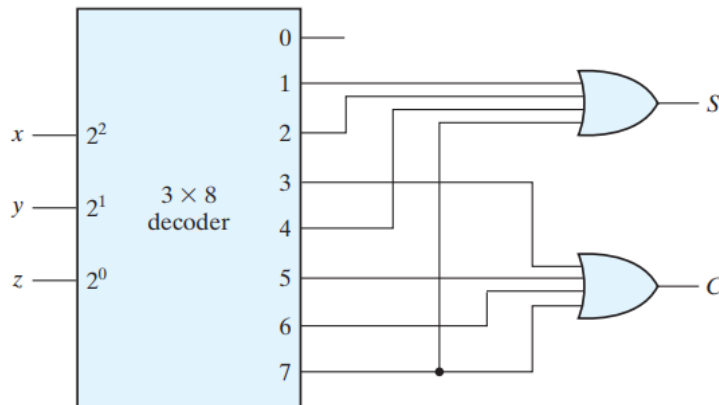
**FIGURE 4.20**  
4 × 16 decoder constructed with two 3 × 8 decoders

Implement the following boolean function using 3:8 decoder

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

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

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



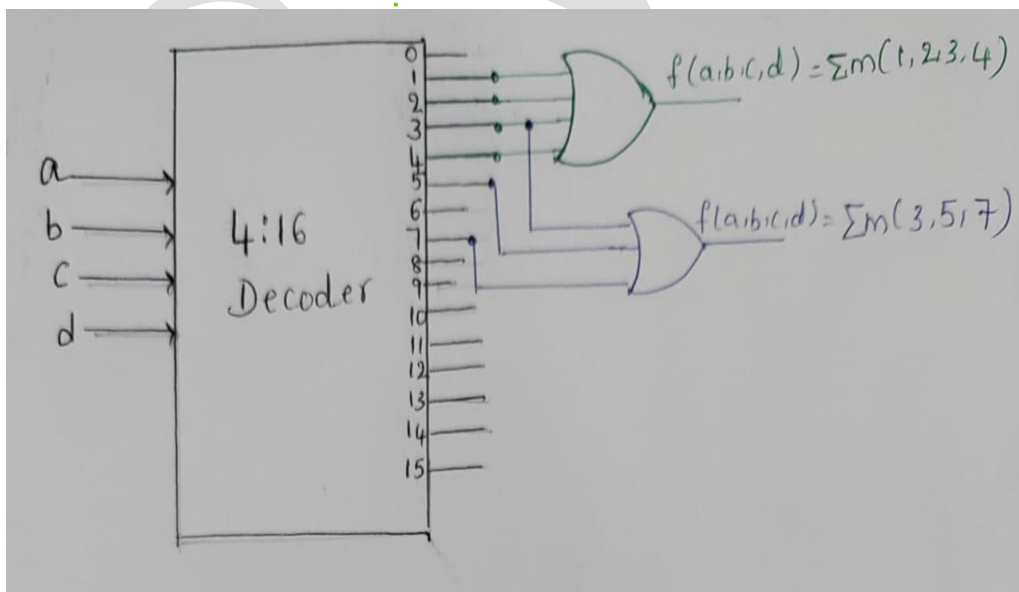
**FIGURE 4.21**  
Implementation of a full adder with a decoder

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

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

i)  $f(a, b, c, d) = \sum m(1, 2, 3, 4)$

ii)  $f(a, b, c, d) = \sum m(3, 5, 7)$



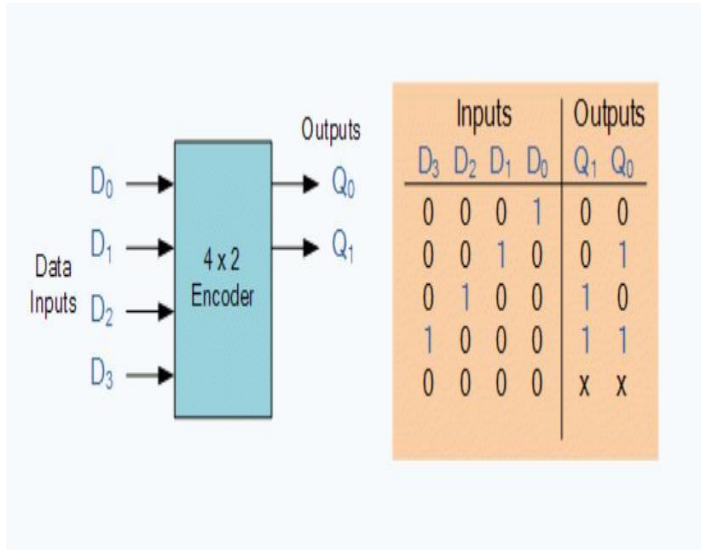
### Encoder

- An encoder is a digital circuit that performs the inverse operation of a decoder.



- An encoder has  $2^n$  (or fewer) input lines and n output lines.
- 4:2 Encoder(n=2)

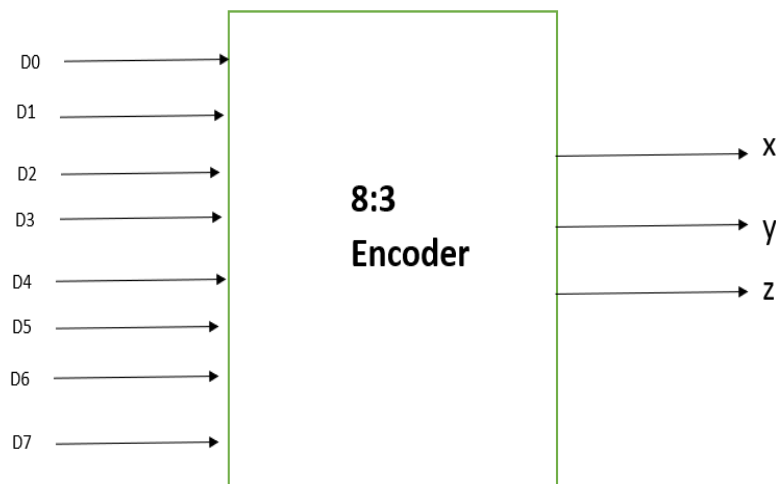
### 4:2 Encoder



### 8:3 Encoder

- an encoder is the octal-to-binary encoder whose truth table is given in Table 4.7
- It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.
- The encoder can be implemented with OR gates whose inputs are determined directly from the truth table
- Output z is equal to 1 when the input octal digit is 1, 3, 5, or 7.
- Output y is 1 for octal digits 2, 3, 6, or 7, and
- output x is 1 for digits 4, 5, 6, or 7.





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

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

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

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

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

- The encoder defined in Table 4.7 has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if  $D_3$  and  $D_6$  are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1.
- To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded.
- The output 111 does not represent either binary 3 or binary 6. To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both  $D_3$  and  $D_6$  are 1 at the same time, the output will be 110 because  $D_6$  has higher priority than  $D_3$ . Another ambiguity in the octal-to-binary encoder is

that an output with all 0's is generated when all the inputs are 0; but this output is the same as when D0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

## Priority Encoder

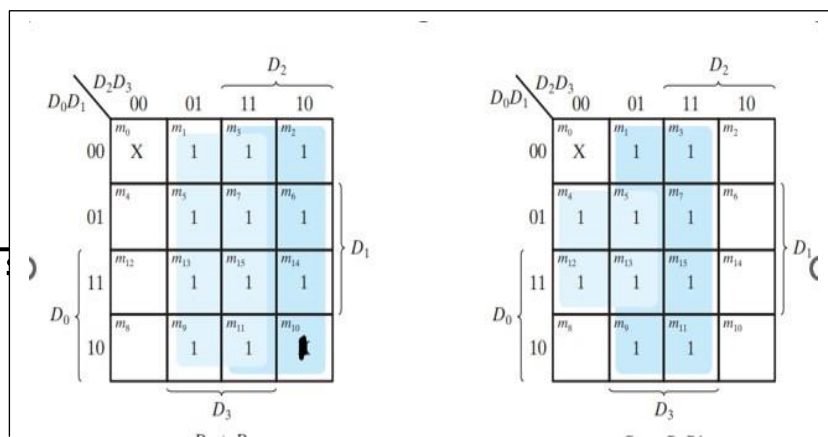
- A priority encoder is an encoder circuit that includes the priority function.
- The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.
- The truth table of a four-input priority encoder is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

**Table 4.8**  
*Truth Table of a Priority Encoder*

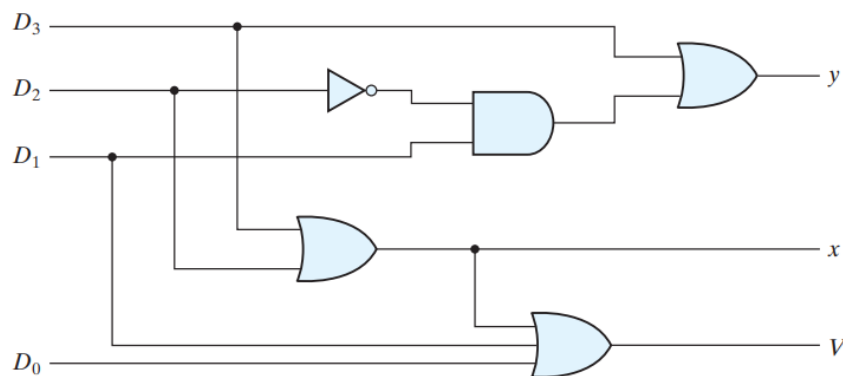
Inputs				Outputs		
$D_0$	$D_1$	$D_2$	$D_3$	$x$	$y$	$V$
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

- In addition to the two outputs  $x$  and  $y$ , the circuit has a third output designated by  $V$ ; this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1.
- If all inputs are 0, there is no valid input and  $V$  is equal to 0. The other two outputs are not inspected when  $V$  equals 0 and are specified as don't-care conditions.
- Input  $D_3$  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).
- $D_2$  has the next priority level. The output is 10 if  $D_2 = 1$ , provided that  $D_3 = 0$ , regardless of the values of the other two lower priority inputs. The output for  $D_1$  is generated only if higher priority inputs are 0.

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



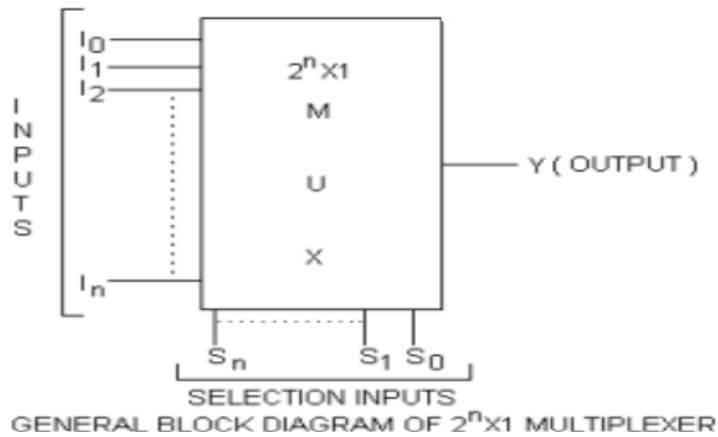
0	0	1	1	1	1	1
0	1	0	0	0	1	1
0	1	0	1	1	1	1
0	1	1	0	1	0	1
0	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	1	1	1	1
1	0	1	0	1	0	1
1	0	1	1	1	1	1
1	1	0	0	0	1	1
1	1	0	1	1	1	1
1	1	1	0	1	0	1
1	1	1	1	1	1	1



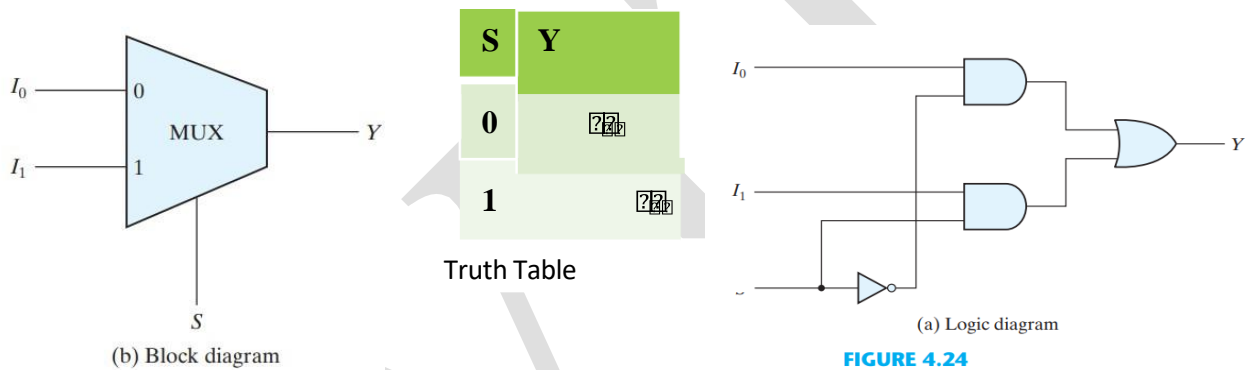
**FIGURE 4.23**  
Four-input priority encoder

## Multiplexer

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



## Design 2:1 Multiplexer



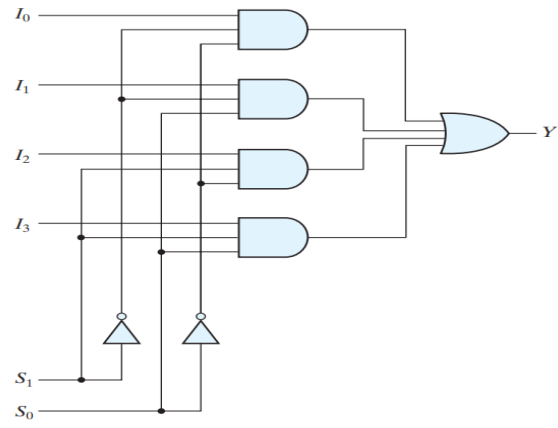
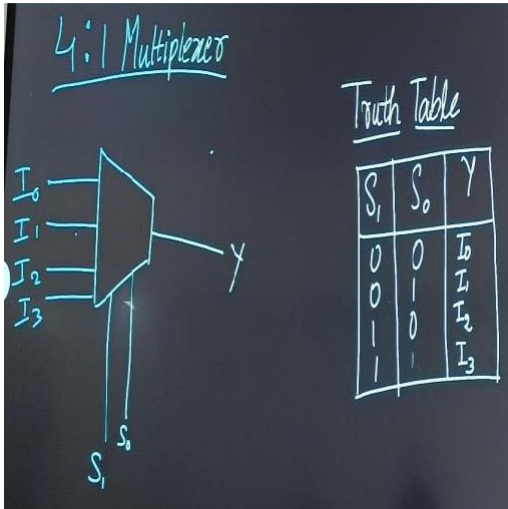
**FIGURE 4.24**  
Two-to-one-line multiplexer

$$y = s'I_0 + sI_1$$

## Boolean Expression

A 2-to-1 multiplexer consists of two inputs  $I_0$  and  $I_1$ , one select input  $S$  and one output  $Y$ . Depending on the select signal, the output is connected to either of the inputs. Since there are two input signals, only two ways are possible to connect the inputs to the outputs, so one select is needed to do these operations.

## 4:1 Multiplexer

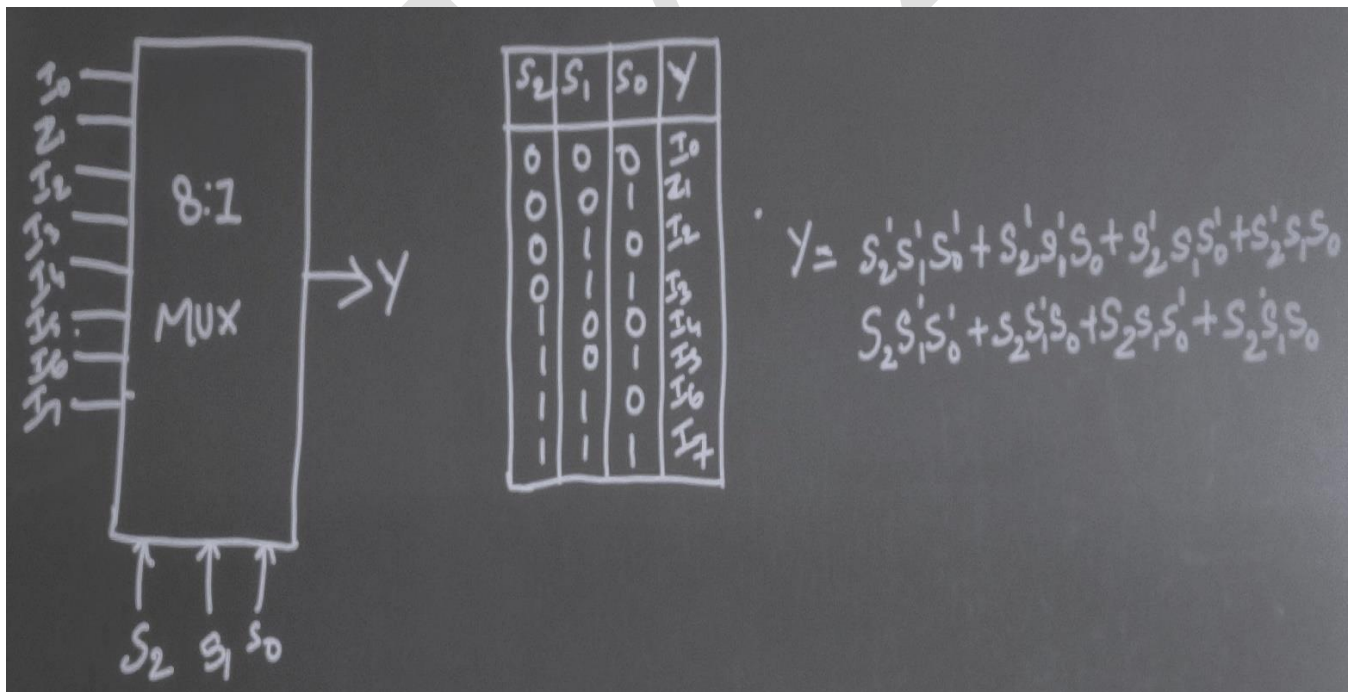


(a) Logic diagram

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

4x1 Multiplexer has four data inputs  $I_0, I_1, I_2$  &  $I_3$ , two selection lines  $S_0$  &  $S_1$  and one output  $Y$ . One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines.

## 8:1 multiplexer



- **Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic.** As an illustration, a **quadruple 2-to-1-line multiplexer** is shown in Fig. 4.26 . The circuit has four multiplexers, each capable of selecting one of two input lines. Output  $Y_0$  can be selected to come from either input  $A_0$  or input  $B_0$ . Similarly, output  $Y_1$  may have the value of  $A_1$  or  $B_1$ , and so on. Input selection line  $S$  selects one of the lines in each of the four multiplexers. The enable input  $E$  must be active (i.e., asserted) for normal operation.
- As shown in the function table, the unit is enabled when  $E = 0$ . Then, if  $S = 0$ , the four  $A$  inputs have a path to the four outputs. If, by contrast,  $S = 1$ , the four  $B$  inputs are applied to the outputs. The outputs have all 0's when  $E = 1$ , regardless of the value of  $S$ .

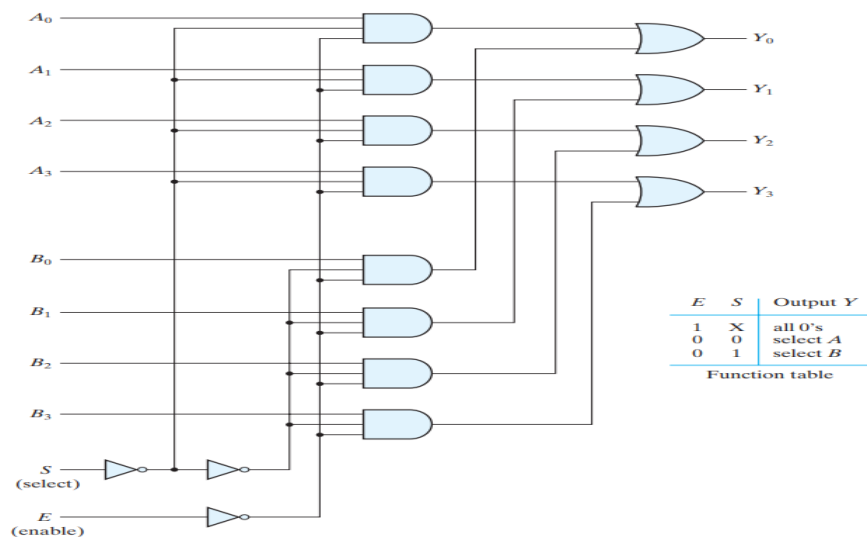
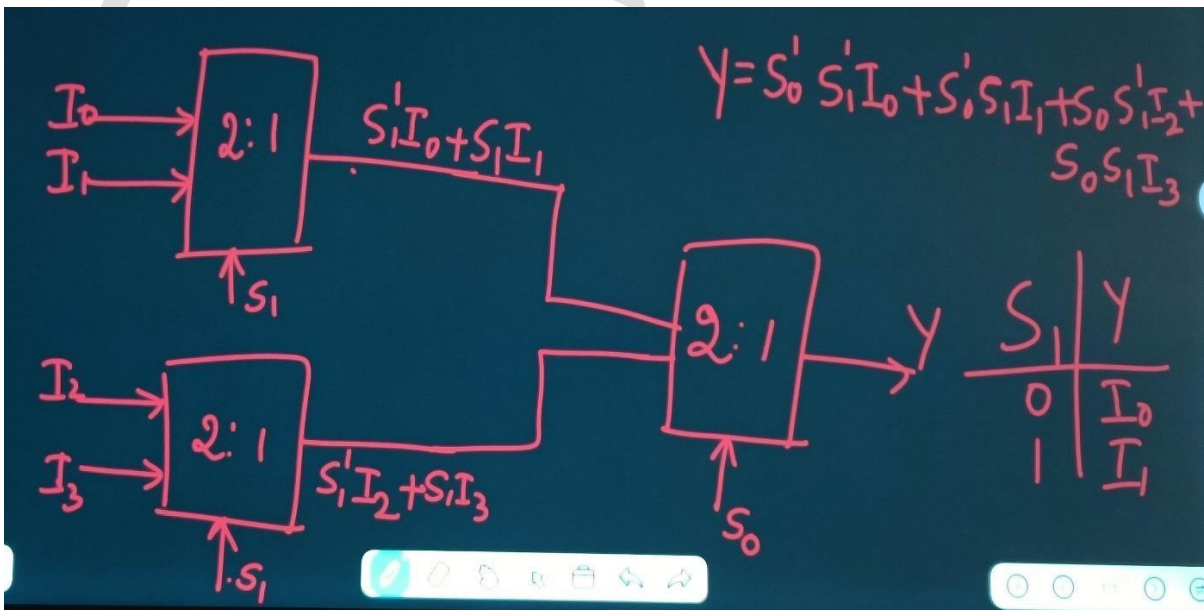


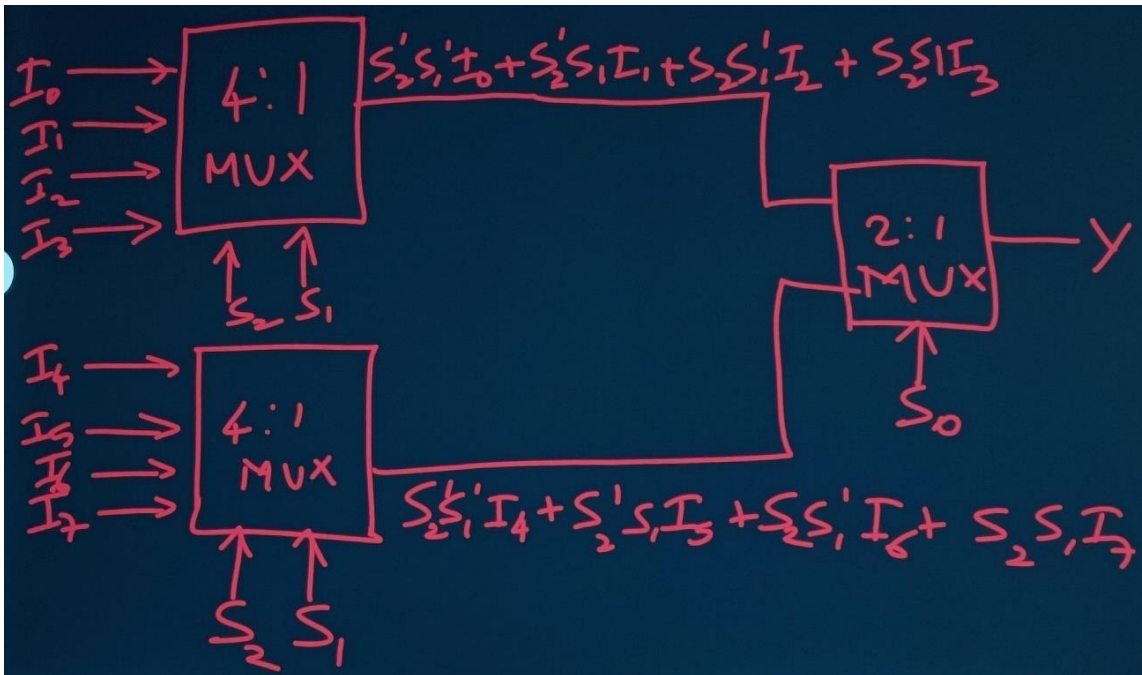
FIGURE 4.26  
Quadruple two-to-one-line multiplexer

Design 4:1 MUX using only 2:1 MUX





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



8:1 MUX Truth Table

$S_0$	$S_2$	$S_1$	$Y$
0	0	0	$I_0$
0	0	1	$I_1$
0	1	0	$I_2$
0	1	1	$I_3$
1	0	0	$I_4$
1	0	1	$I_5$
1	1	0	$I_6$
1	1	1	$I_7$

$$Y = S_0'S_2'S_1'I_0 + S_0'S_2'S_1'I_1 + S_0'S_2S_1'I_2 + S_0'S_2S_1'I_3 + S_0S_2'S_1'I_4 + S_0S_2'S_1'I_5 + S_0S_2S_1'I_6 + S_0S_2S_1'I_7$$

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

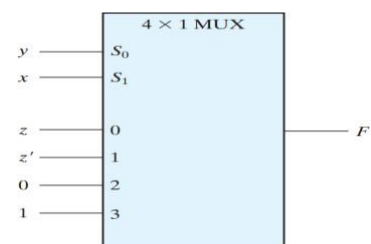
$n=3$

$n-1=2$  select lines

$2^{n-1}=4$  data inputs

$x$	$y$	$z$	$F$
0	0	0	0
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

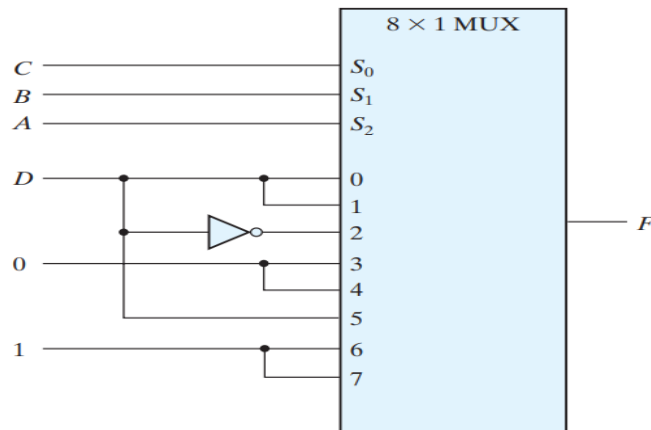
(a) Truth table



(b) Multiplexer implementation

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

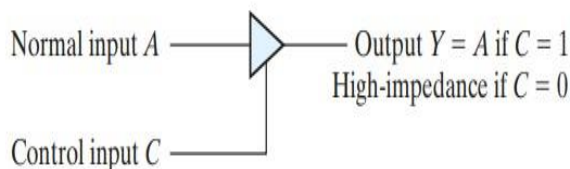
A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



**FIGURE 4.28**  
Implementing a four-input function with a multiplexer

### Three-State Gates

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

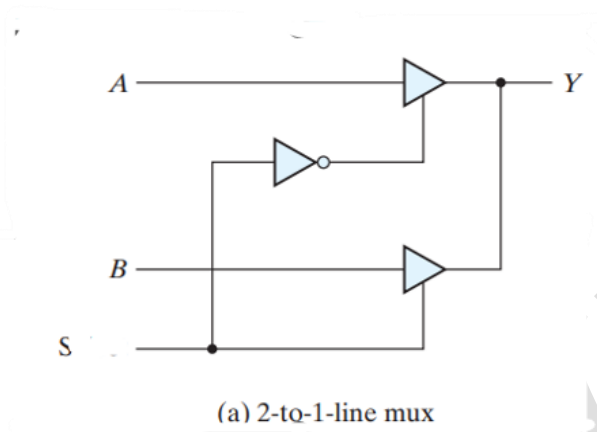


The buffer has a normal input, an output, and a control input that determines the state of the output. When the control input is equal to 1, the output is enabled and the gate behaves like a conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled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



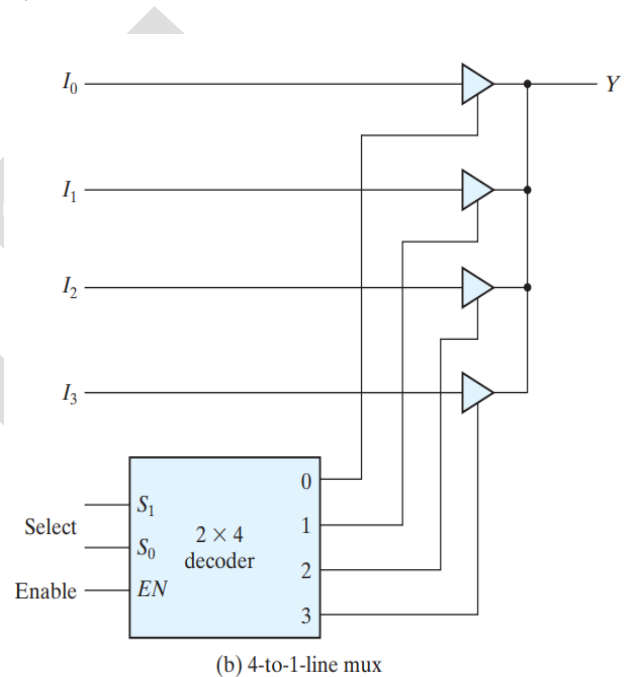
feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common line without endangering loading effects.

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



S	Y
0	A
1	B

$$Y = AS' + BS$$



### HDL models of combinational circuits

- The logic of a module can be described in any one (or a combination) of the following modeling styles:
- Behavioral modeling** using procedural assignment statements with the **keyword always**.
- Gate-level (structural) modeling** describes a circuit by specifying its gates and how they are connected with each other. Gate-level modeling using instantiations of predefined and user-defined primitive gates
- Dataflow modeling** is used mostly for describing the Boolean equations of combinational logic, Dataflow modeling using continuous assignment statements with the keyword assign.

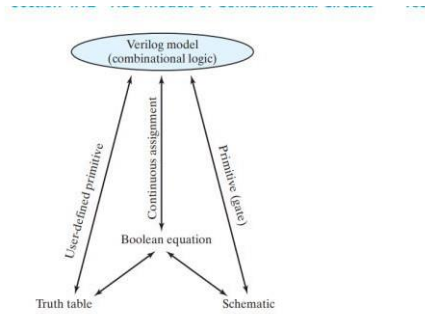


FIGURE 4.31 Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

Table 4.10  
Some Verilog HDL Operators

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

Write a Verilog Program For Binary Adder(4bit )

## HDL (Dataflow: Four-Bit Adder)

```

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

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

Using **cond itional operator**

*condition ? true-expression : false-expression;*

s	y
0	$I_0$
1	$I_1$

$$y = s'I_0 + sI_1$$

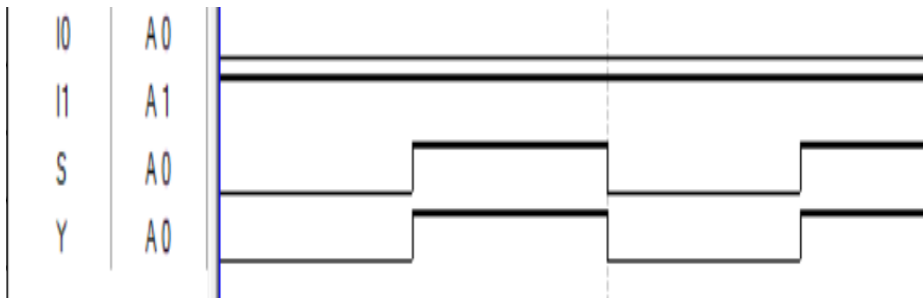
Using Data flow Model

```

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

```

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



### Behavioral modelling for 2:1 Mux

#### Using Case Statement

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

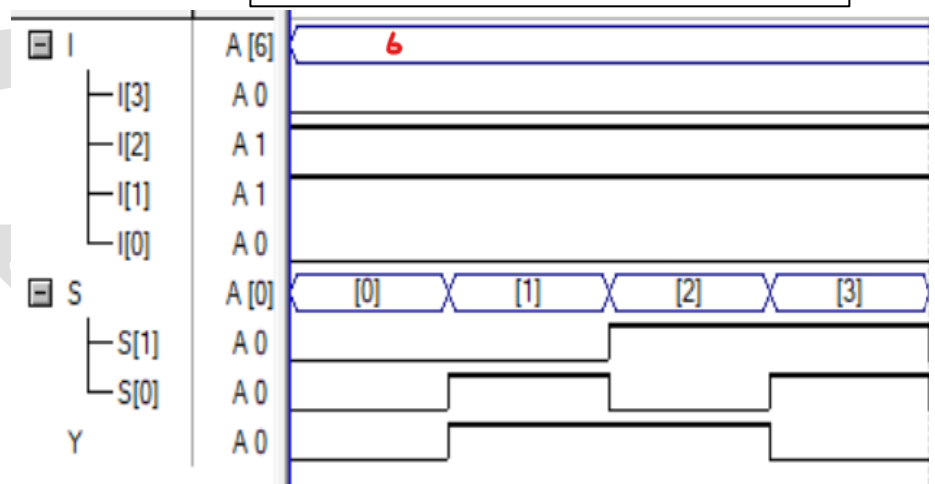
#### using If else statement

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

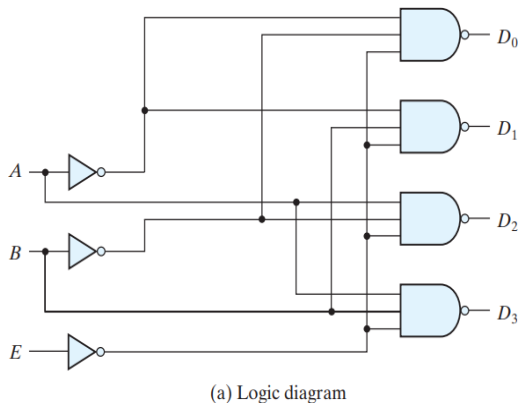
### Write Verilog program for 4:1mux using CASE STATEMENT

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

Timing Diagram



Write a Verilog code for below figure



<i>E</i>	<i>A</i>	<i>B</i>	<i>D</i> <sub>0</sub>	<i>D</i> <sub>1</sub>	<i>D</i> <sub>2</sub>	<i>D</i> <sub>3</sub>
1	<i>X</i>	<i>X</i>	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	1

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

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

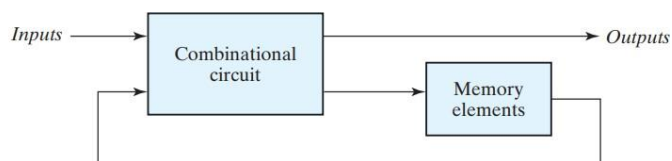
```

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

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

## Sequential Logic

- Sequential logic refers to a type of digital logic circuit that uses memory elements to store information.
- It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information.
- a sequential circuit is specified by a time sequence of inputs, outputs, and internal states.

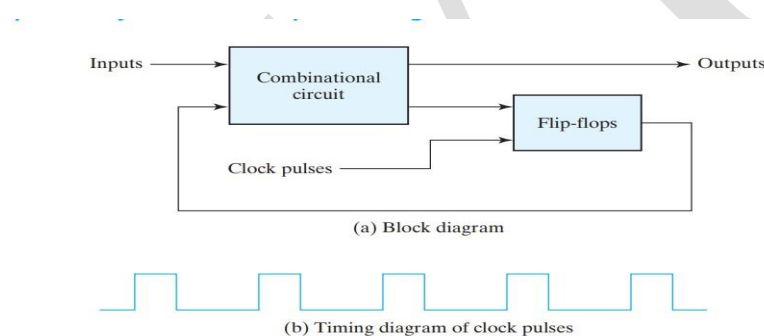


**FIGURE 5.1**  
 Block diagram of sequential circuit

## Differentiate between combinational logic and sequential logic

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

- The storage elements (memory) used in clocked sequential circuits are called flipflops.
- A flip-flop is a binary storage device capable of storing one bit of information.



**FIGURE 5.2**  
Synchronous clocked sequential circuit

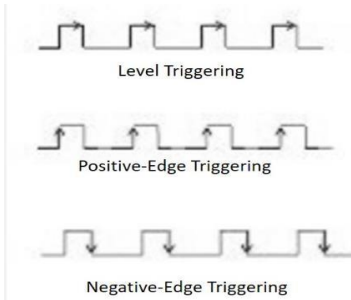
### Storage Elements:

#### 1)Latches:

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

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

- Storage elements that operate with signal **levels** (rather than signal transitions) are referred to as **latches** ; those **controlled by a clock transition** are **flip-flops**. Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices. The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed.

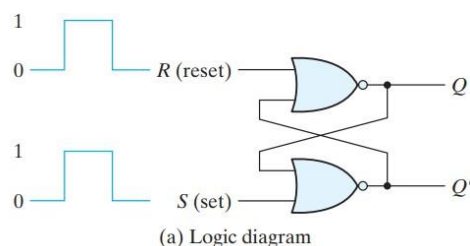


### SR Latch (Set-Reset Latch):

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

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

### SR Latch with nor gates



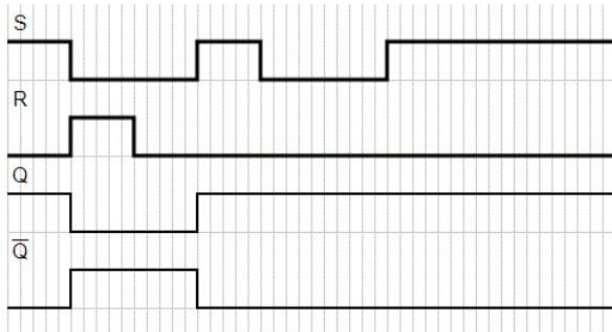
**FIGURE 5.3**  
SR latch with NOR gates

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

(b) Function table

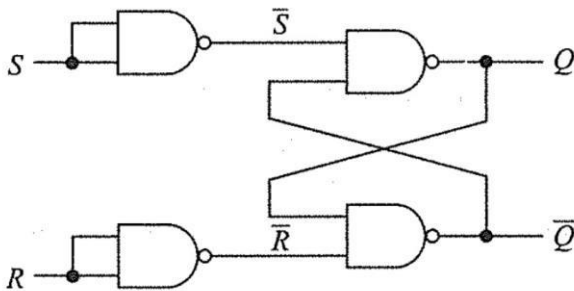
where S and R stand for set and reset. It can be constructed from a pair of cross-coupled NOR logic gates. The stored bit is present on the output marked Q.

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



Timing Diagram of SR latch

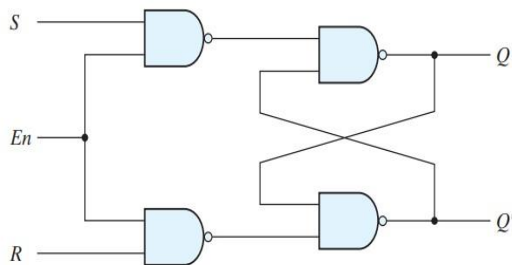
## SR latch with NAND gates



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

(b) Function table

## SR latch with control input



(a) Logic diagram

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

(b) Function table

**FIGURE 5.5**  
SR latch with control input

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

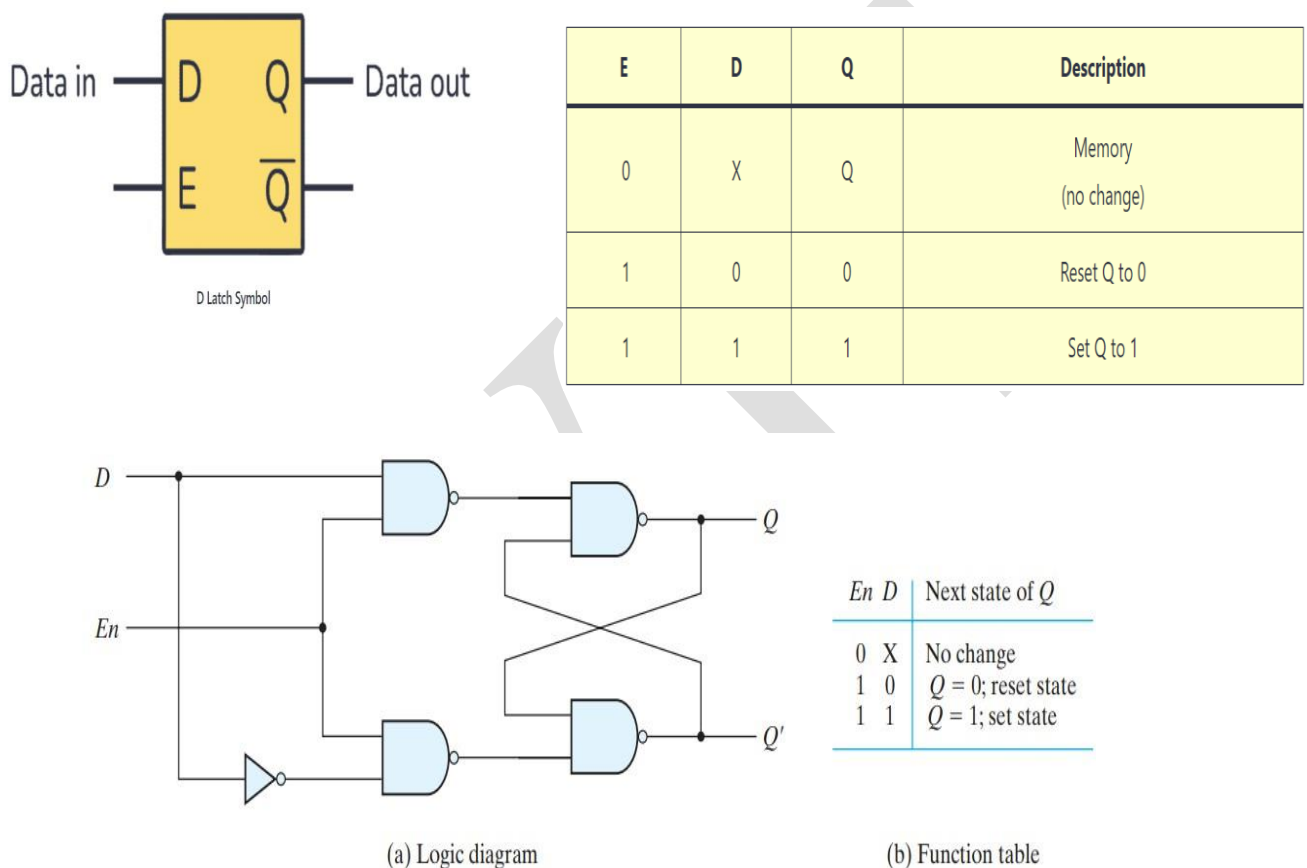


not change. These conditions are listed in the function table accompanying the diagram.

### D latch(transparent latch)

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

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



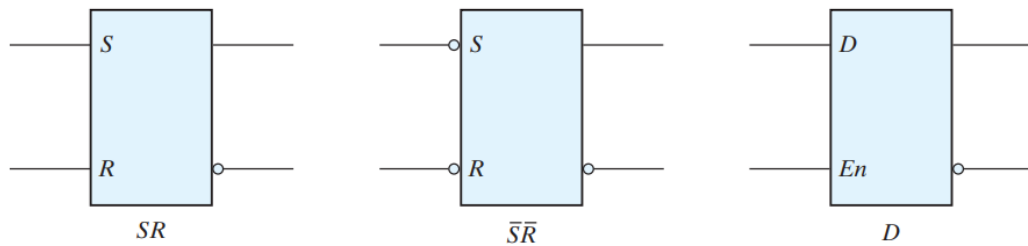
**FIGURE 5.6**  
D latch

One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs S and R are never equal to 1 at the same time. This is done in the D latch, shown in Fig. 5.6 . This latch has only two inputs: D (data) and En (enable). The D input goes directly to the S input, and its complement is applied to the R input. As long as the enable input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of D . The D input is sampled when En = 1.



If  $D = 1$ , the  $Q$  output goes to 1, placing the circuit in the set state. If  $D = 0$ , output  $Q$  goes to 0, placing the circuit in the reset state.

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

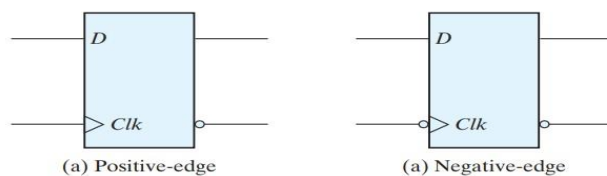


**FIGURE 5.7**  
Graphic symbols for latches

## STORAGE ELEMENTS : FLIP - FLOPS

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

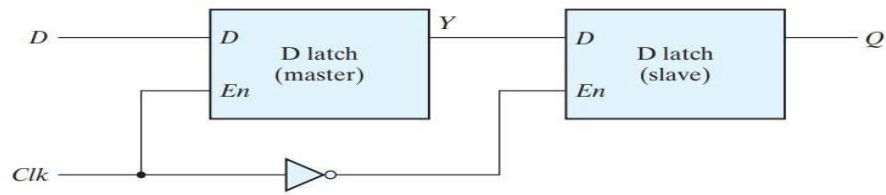
### Edge triggered DFF



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

Table of truth:

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



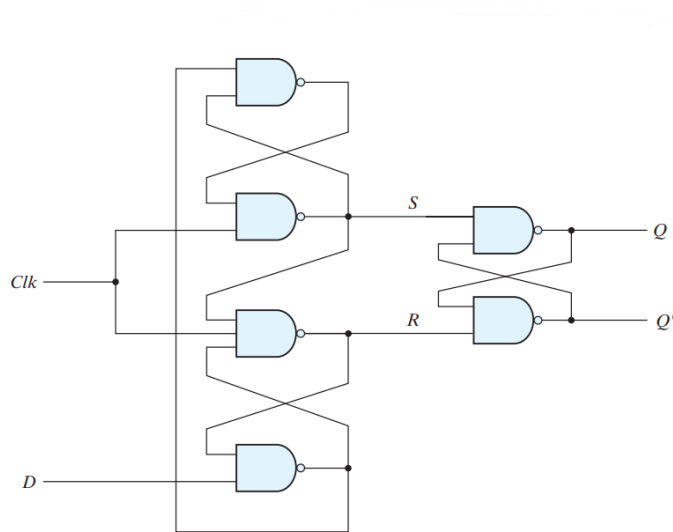
**FIGURE 5.9**  
Master-slave D flip-flop

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

### Comparison between Latch and Flipflop

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

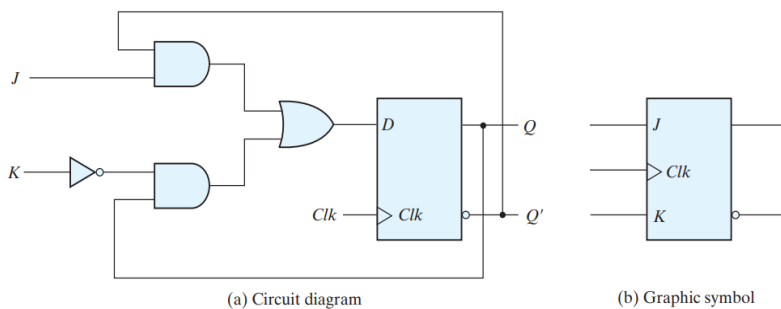
construction of an **positive edge-triggered D flip-flop** uses three SR latches



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

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

## JK FLIPFLOP



**FIGURE 5.12**  
JK flip-flop

**Table 5.1**  
Flip-Flop Characteristic Tables

JK Flip-Flop		
J	K	Q(t + 1)
0	0	Q(t) No change
0	1	0 Reset
1	0	1 Set
1	1	Q'(t) Complement

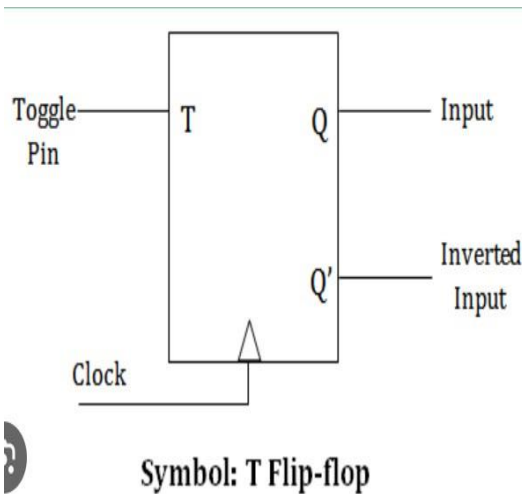
When  $J = 1$  and  $K = 0$ ,  $D = Q' + Q = 1$ , so the next clock edge sets the output to 1.

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

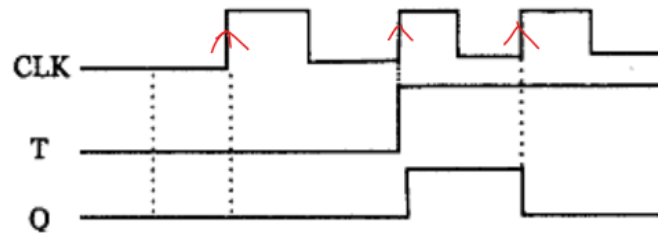
When both  $J = K = 1$  and  $D = Q$ , the next clock edge complements the output.

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

## T Flipflop

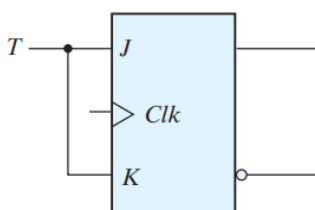


Inputs		Outputs	
CLK	T	$Q_{n+1}$	Action
0	X	$Q_n$	No change
1	0	$Q_n$	No change
1	1	$\overline{Q_n}$	Toggle



### T Flipflop using JK Flipflop

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



(a) From JK flip-flop

**FIGURE 5.13**  
T flip-flop

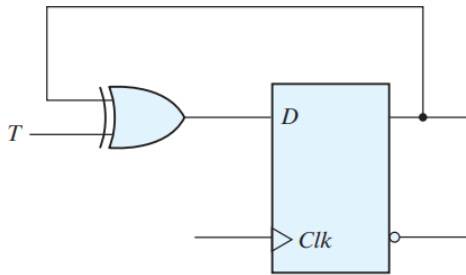
Clk	T	Q	Q'
↑	0	No Change	
↑	1	toggle	

### Implementation of TFF using DFF

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

$$D = T \oplus Q$$

$$D = T'Q + TQ'$$



(b) From D flip-flop

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

**Table 5.1**  
Flip-Flop Characteristic Tables

<b>J/K Flip-Flop</b>		
<b>J</b>	<b>K</b>	<b><math>Q(t + 1)</math></b>
0	0	$Q(t)$ No change
0	1	0 Reset
1	0	1 Set
1	1	$Q'(t)$ Complement

<b>D Flip-Flop</b>		
<b>D</b>	<b><math>Q(t + 1)</math></b>	
0	0	Reset
1	1	Set

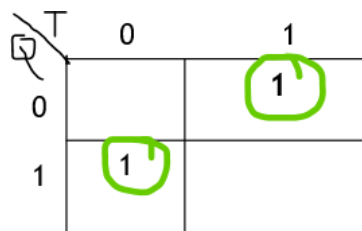
<b>T Flip-Flop</b>		
<b>T</b>	<b><math>Q(t + 1)</math></b>	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

### Characteristic equation

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

#### T FF

<b>Q</b>	<b>T</b>	<b><math>Q(t+1)</math></b>
0	0	0
0	1	1
1	0	1
1	1	0



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

## DFF

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

Q \ D	0	1
0		1
1		1

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

## JKFF

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

Q \ JK	00	01	11	10
0			1	1
1	1			1

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

## Write Verilog code for Flipflops

### SR flipflop

```

module sr(clk,s,r,q);
  input clk,s,r;
  output q;
  reg q;
  always @(posedge clk)
    begin
      case ({s,r})
        2'b00: q <= q; // No change
        2'b01: q <= 1'b0; // reset
        2'b10: q <= 1'b1; // set
        2'b11: q <= 1'bx; // Invalid inputs
      endcase
    end
endmodule

```

### JK Flipflop

```

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

```

**D flipflop**

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

**T Flipflop**

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

-----\*\*\*\*\*-----