

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028



A T M E

College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES



**COURSE: DIGITAL DESIGN AND COMPUTER
ORGANIZATION**

COURSE CODE: 1BCS303

SEMESTER: III

COURSE COORDINATOR: Ms. Bhavana V R

(ACADEMIC YEAR 2026-2027)

INSTITUTIONAL MISSION AND VISION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torchbearers of tomorrow's society
- To strive to attain ever-higher benchmarks of educational excellence

DEPARTMENT MISSION AND VISION

Vision

To be a global leader in Computer Science and Design Engineering striving for design excellence, cultural awareness, a profound commitment to environmental stewardship, and societal responsibility.

Mission

1. To establish a technology-enabled experiential learning environment, prioritizing and cultivating problem-solving and design thinking skills among students.
2. To foster collaboration with industries, research and development organizations, jointly addressing socially relevant challenges in Computer Science with a core emphasis on design.

Programme Educational Objectives (PEOs)

PEO1: Demonstrate the proficiency and adaptability needed to seamlessly integrate with the latest advancements and changes in Computer Science and Design.

PEO2: Our graduates will possess the adaptability to explore various career paths, such as Research & Development, Manufacturing, and Entrepreneurship with independent design firms, Design Services, and Consultancies.

PEO3: Our students will be well-prepared to excel in collaborative, multidisciplinary environments, showcasing their versatility and skills.

Programme Specific Outcomes (PSOs)

PSO1: To develop stand-alone, embedded, and web-based solutions with easy-to-operate interfaces using software engineering practices and contemporary programming languages.

PSO2: Design and develop computer-based systems in various areas of Multimedia, Graphics data visualization and computer vision.

MODULE - 2

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.

Text book 1: 4.1, 4.2, 4.3, 4.4, 4.5, 4.9, 4.10, 4.11, 4.12, 5.1, 5.2, 5.3, 5.4

Text book:

1. M. Morris Mano & Michael D. Ciletti, “Digital Design with an Introduction to Verilog Design”, 5e, Pearson Education.

Chapter 4

Combinational Logic

4.1 INTRODUCTION

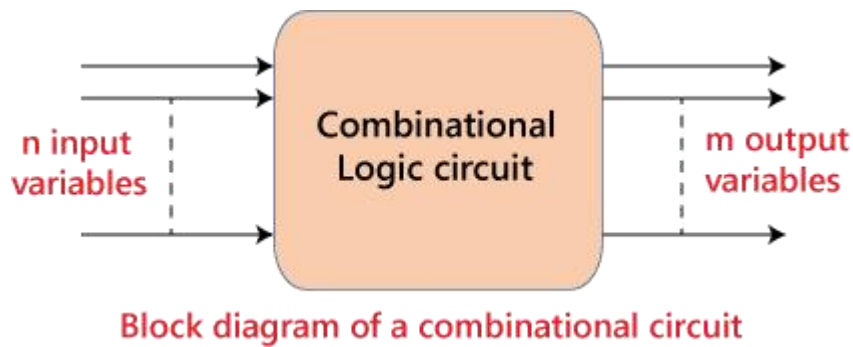
Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs. A combinational circuit performs an operation that can be specified logically by a set of Boolean functions. In contrast, 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. Sequential circuits are the building blocks of digital systems.

4.2 COMBINATIONAL CIRCUITS

The combinational logic circuits are the circuits that contain different types of logic gates. Simply, a circuit in which different types of logic gates are combined is known as a combinational logic circuit. The output of the combinational circuit is determined from the present combination of inputs, regardless of the previous input. The input variables, logic gates, and output variables are the basic components of the combinational logic circuit. There are different types of combinational logic circuits, such as Adder, Subtractor, Decoder, Encoder, Multiplexer, and De-multiplexer.

There are the following characteristics of the combinational logic circuit:

- At any instant of time, the output of the combinational circuits depends only on the present input terminals.
- The combinational circuit doesn't have any backup or previous memory. The present state of the circuit is not affected by the previous state of the input.
- The n number of inputs and m number of outputs are possible in combinational logic circuits.



The 'n' input variable comes from the external source while the 'm' output variable goes to the external destination. In many applications, the source or destinations are storage registers.

4.4 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).

A truth table for a combinational circuit consists of input columns and output columns. The input columns are obtained from the 2^n binary numbers for the n input variables. The binary values for the outputs are determined from the stated specifications. The output functions specified in the truth table give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly in the truth table, as they are often incomplete, and any wrong interpretation may result in an incorrect truth table. The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program. Frequently, there is a variety of simplified expressions from which to choose. In a particular application, certain criteria will serve as a guide in the process of choosing an implementation. A practical design must consider such constraints as the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits. Since the importance of each constraint is dictated by the particular application, it is difficult to make a general statement about what constitutes an acceptable implementation. In most cases, the simplification begins by satisfying an elementary objective, such as producing the simplified Boolean functions in a standard form. Then the simplification proceeds with further steps to meet other performance criteria.

Example: The design procedure will be illustrated by an example that converts binary coded decimal (BCD) to the excess-3 code for the decimal digits

Excess-3 binary code is an **unweighted self-complementary BCD code**. Self-Complementary property means that the 1's complement of an excess-3 number is the excess-3 code of the 9's complement of the corresponding decimal number.

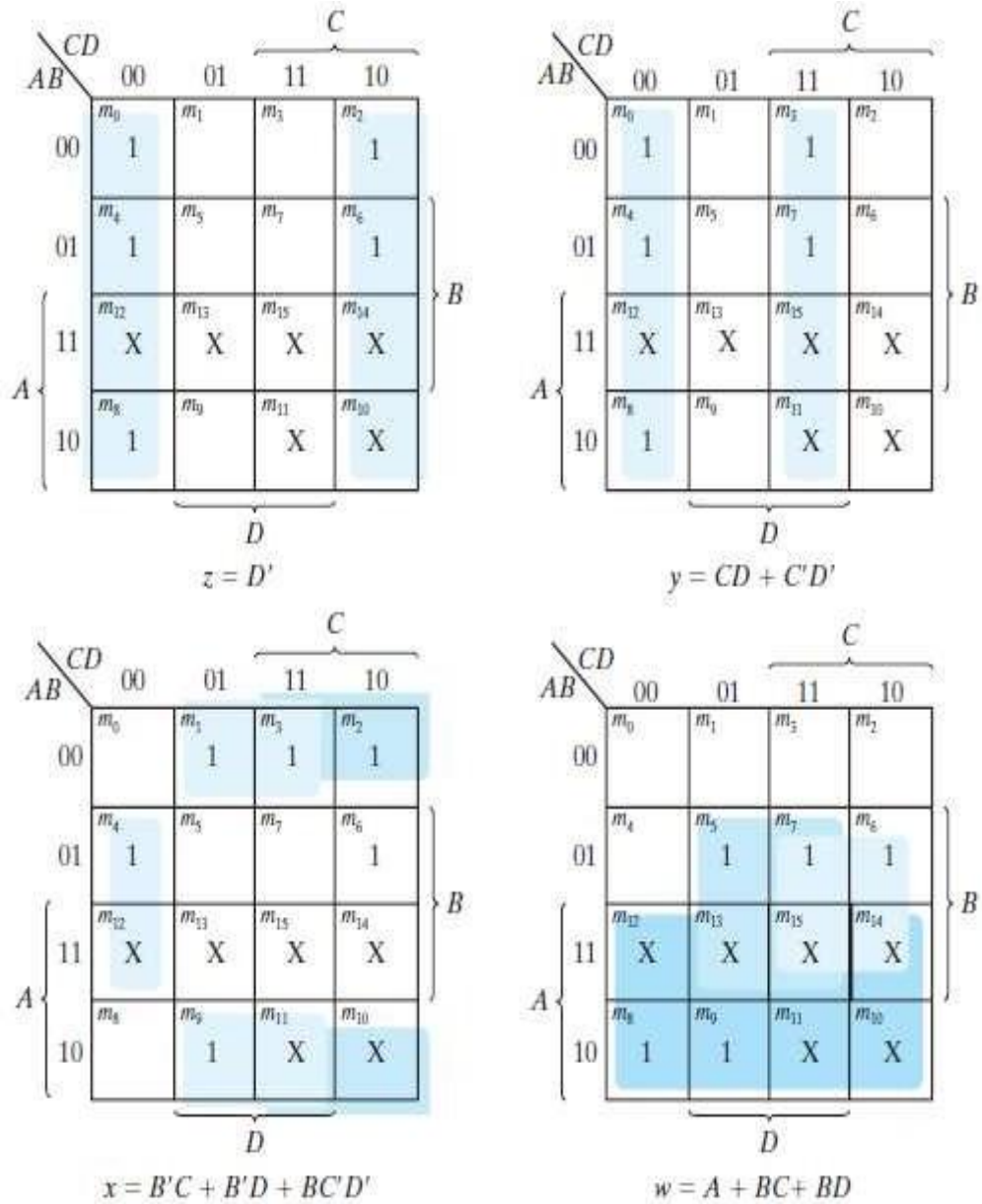
The process of converting BCD to Excess-3 is quite simple from other conversions. The Excess-3 code can be calculated by adding 3, i.e., 0011 to each four-digit BCD code.

Below is the truth table for the conversion of BCD to Excess-3 code. In the below table, the variables A, B, C, and D represent the bits of the binary numbers. The variable 'D' represents the LSB, and the variable 'A' represents the MSB. In the same way, the variables w, x, y, and z represent the bits of the Excess-3 code. The variable 'z' represents the LSB, and the variable 'w' represents the MSB. The 'don't care conditions' is expressed by the variable 'X'.

Decimal Number	BCD Code				Excess-3 Code			
	A	B	C	D	w	x	y	z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
10	1	0	1	0	X	X	X	X
11	1	0	1	1	X	X	X	X
12	1	1	0	0	X	X	X	X
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

Now, we will use the K-map method to design the logical circuit for the conversion of BCD to Excess-3 code as:

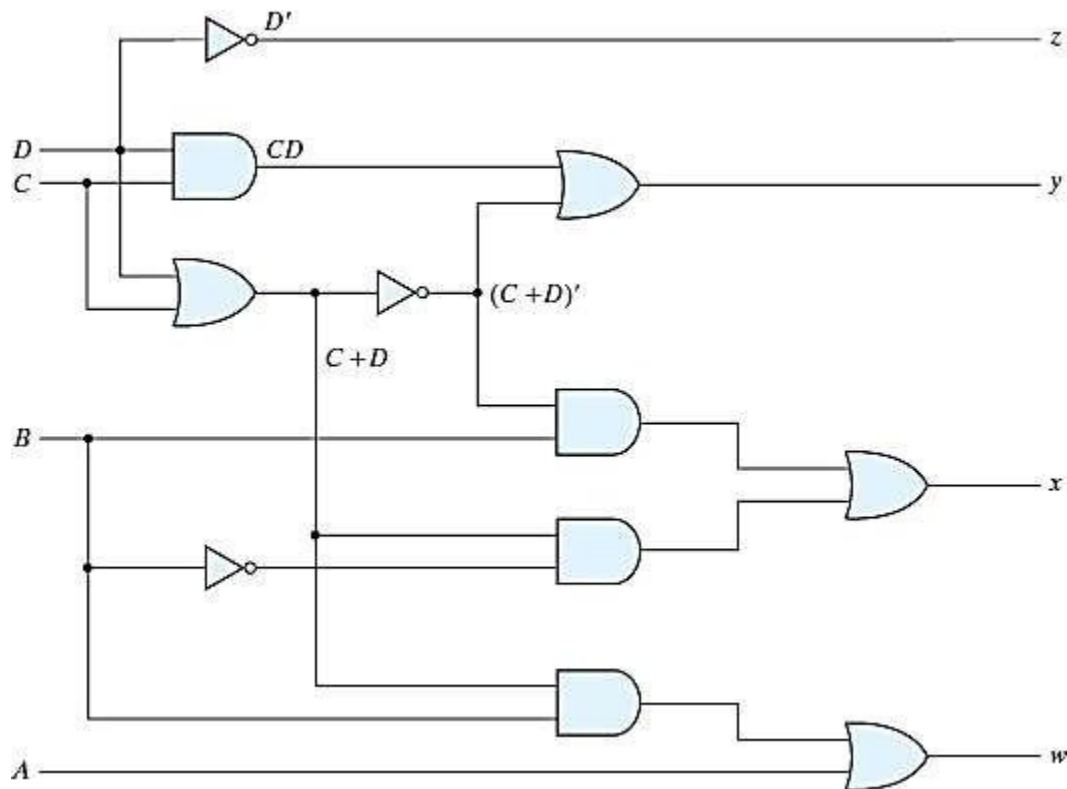
K-Maps for BCD-to-excess-3 code converter:



So,

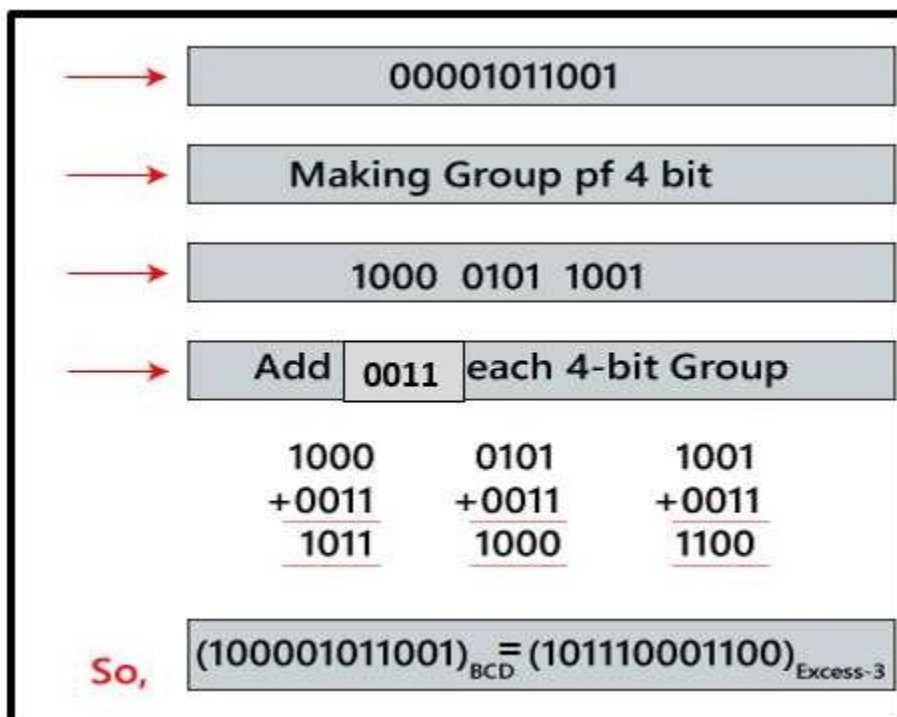
$$\begin{aligned} w &= A + BC + BD \\ x &= B'C + B'D + BC'D' \\ y &= CD + C'D' \\ z &= D' \end{aligned}$$

Logic diagram for BCD-to-excess-3 code converter:



Example: $(100001011001)_{\text{BCD}}$

To find the Excess-3 code of the given Excess-3 code, first, we will make the group of 4 bits from right to left. Then, we will add 0011 in each group of 4 bits in order to get the excess-3 code.



Excess-3 to BCD conversion

The process of converting Excess-3 to BCD is opposite to the process of converting BCD to Excess-3. The BCD code can be calculated by subtracting 3, i.e., 0011 from each four-digit Excess-3 code. Below is the truth table for the conversion of Excess-3 code to BCD. In the below table, the variables

w, x, y, and z represent the bits of the Excess-3 code. The variable 'z' represents the LSB, and the variable 'w' represents the MSB. In the same way, the variables A, B, C, and D represent the bits of the binary numbers. The variable 'D' represents the LSB, and the variable 'A' represents the MSB. The 'don't care conditions' is defined by the variable 'X'.

Decimal Number	Excess-3 Code				BCD Code			
	A	B	C	D	w	x	y	z
0	0	0	0	0	X	X	X	X
1	0	0	0	1	X	X	X	X
2	0	0	1	0	X	X	X	X
3	0	0	1	1	0	0	0	0
4	0	1	0	0	0	0	0	1
5	0	1	0	1	0	0	1	0
6	0	1	1	0	0	0	1	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	0	1	0	1
9	1	0	0	1	0	1	1	0
10	1	0	1	0	0	1	1	1
11	1	0	1	1	1	0	0	0
12	1	1	0	0	1	0	0	1
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

Now, we will use the K-map method to design the logical circuit for the conversion of Excess-3 code to BCD as:

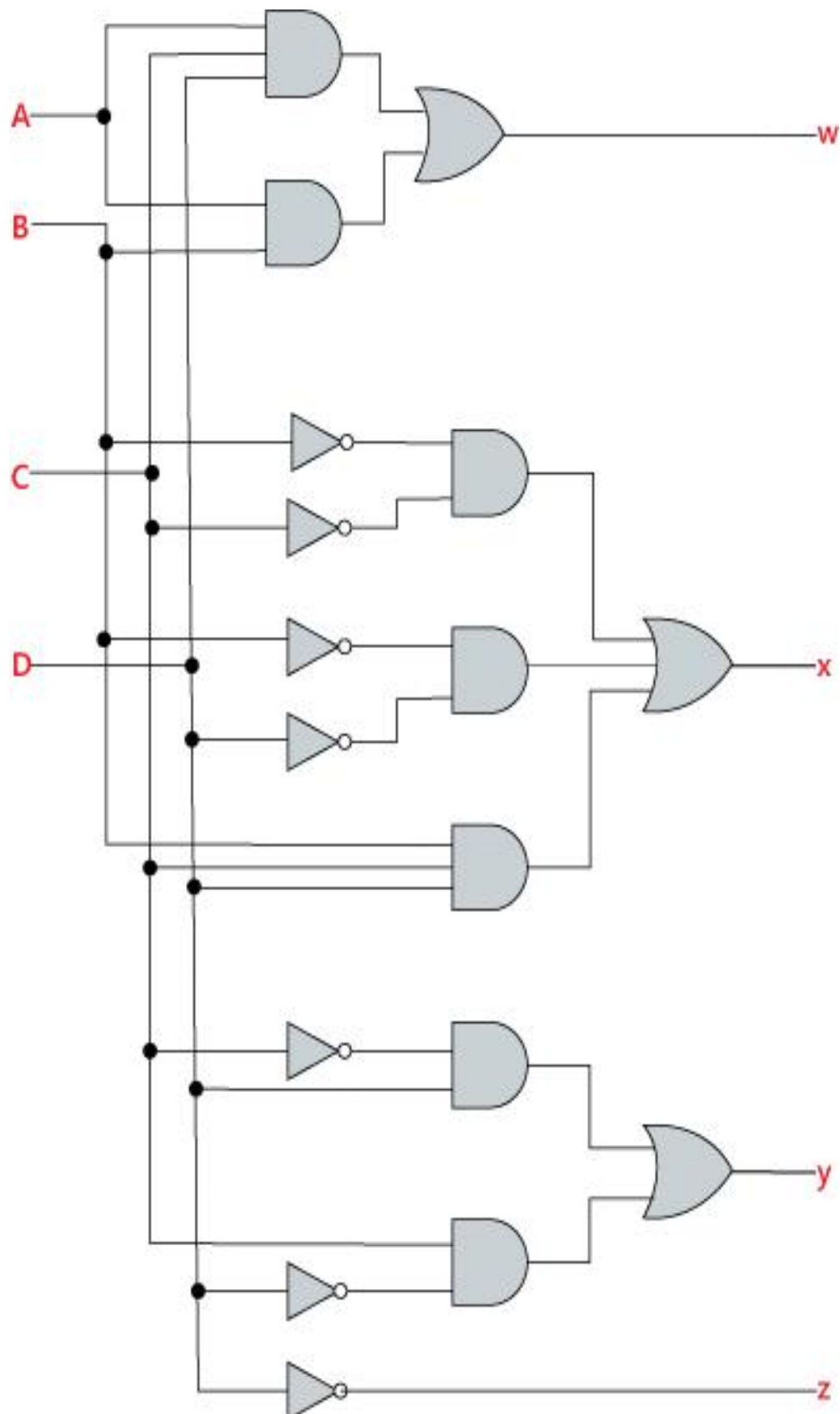
So,

$$w = AB + ACD$$

$$x = A' B' + A' D' + ABC$$

$$y = B' D + BD'$$

$$z=D'$$



4.5 BINARY ADDER-SUBTRACTOR

Addition and Subtraction are two basic Arithmetic Operations that must be performed by any Digital Computer. If both these operations can be properly implemented, then Multiplication and Division tasks become easy (as multiplication is repeated addition and division is repeated subtraction).

Consider the operation of adding two binary numbers, which is one of the fundamental tasks performed by a digital computer. The four basic addition operations two single bit binary numbers are:

- $0 + 0 = 0$
- $1 + 0 = 1$
- $0 + 1 = 1$
- $1 + 1 = (\text{Carry})1\ 0$

1	1	0	0
<u>+1</u>	<u>+0</u>	<u>+1</u>	<u>+0</u>
(carry)1 0	1	1	0

The first three operations produce a sum of one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a carry. The logic circuits which are designed to perform the addition of two binary numbers are called as Binary Adder Circuits. Depending on how they handle the output of the '1+1' addition, they are divided into:

1. Half Adder
2. Full Adder

1. Half Adder

A logic circuit used for adding two 1-bit numbers or simply two bits is called as a Half Adder circuit. This circuit has two inputs and two outputs. The inputs are the two 1-bit binary numbers (known as Augend and Addend) and the outputs are Sum and Carry.

Truth Table:

INPUT		OUTPUT	
x	y	Sum (S)	Carry (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

K-map simplification:

	y	
	0	0
x	0	1

$C = xy$

	y	
	0	1
x	1	0

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

Boolean functions:

$$\text{Sum: } S = x'y + xy'$$

Carry: $C = xy$

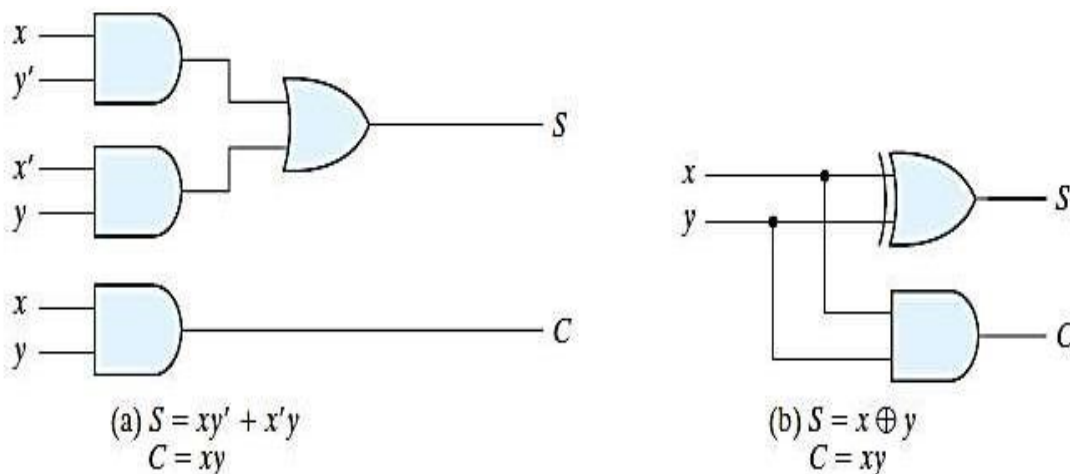


FIGURE 4.5
Implementation of half adder

Half adder is mainly used for addition of augend and addend of first order binary numbers i.e., 1-bit binary numbers. We cannot add binary numbers with more than one bit as the Half Adder cannot include the 'Carry' information from the previous sum.

Due to this limitation, Half Adder is practically not used in many applications, especially in multi-digit addition. In such applications, carry of the previous digit addition must be added along with two bits; hence it is a three bit addition.

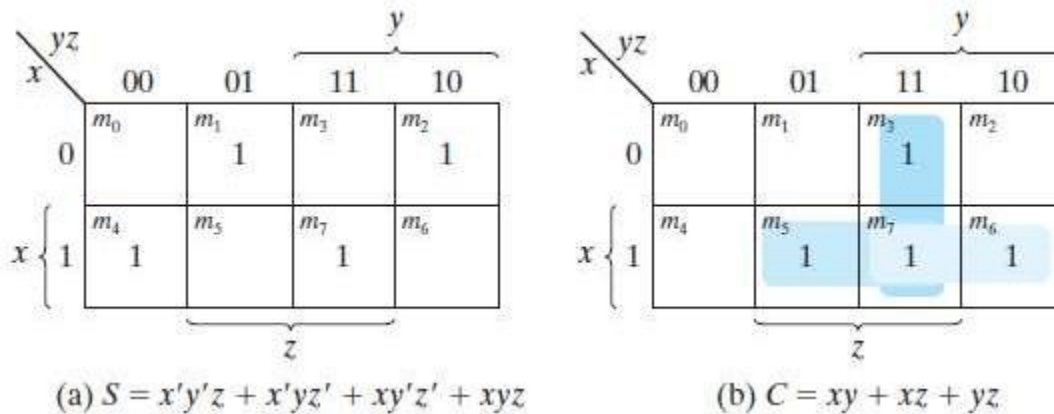
2. Full Adder

A Full Adder is a combinational logic circuit which performs addition on three bits and produces two outputs: a Sum and a Carry. As we have seen that the Half Adder cannot respond to three inputs and hence the full adder is used to add three digits at a time.

It consists of three inputs, of which two are input variables representing the two significant bits to be added, whereas the third input terminal is the carry from the previous addition. The two outputs are a Sum and Carry outputs.

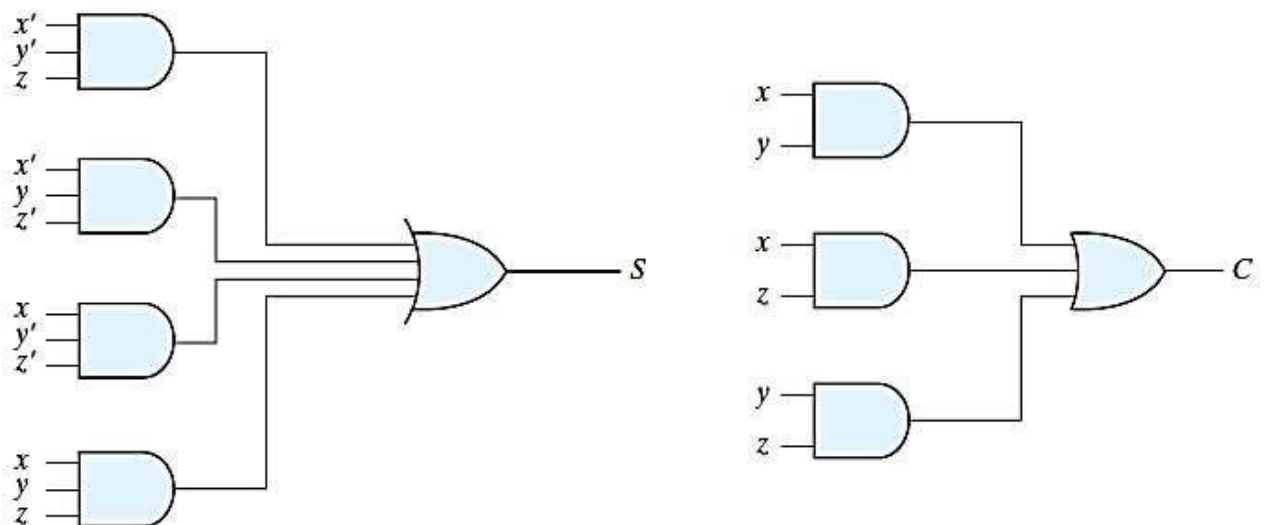
Truth Table:

INPUT			OUTPUT	
x	y	z	Sum(S)	Carry(C)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	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

K-Map:**Boolean Function:**

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

$$C = xy + xz + yz$$

Implementation of full adder in sum-of-products form:**Implementation of full adder with two half adders and an OR gate**

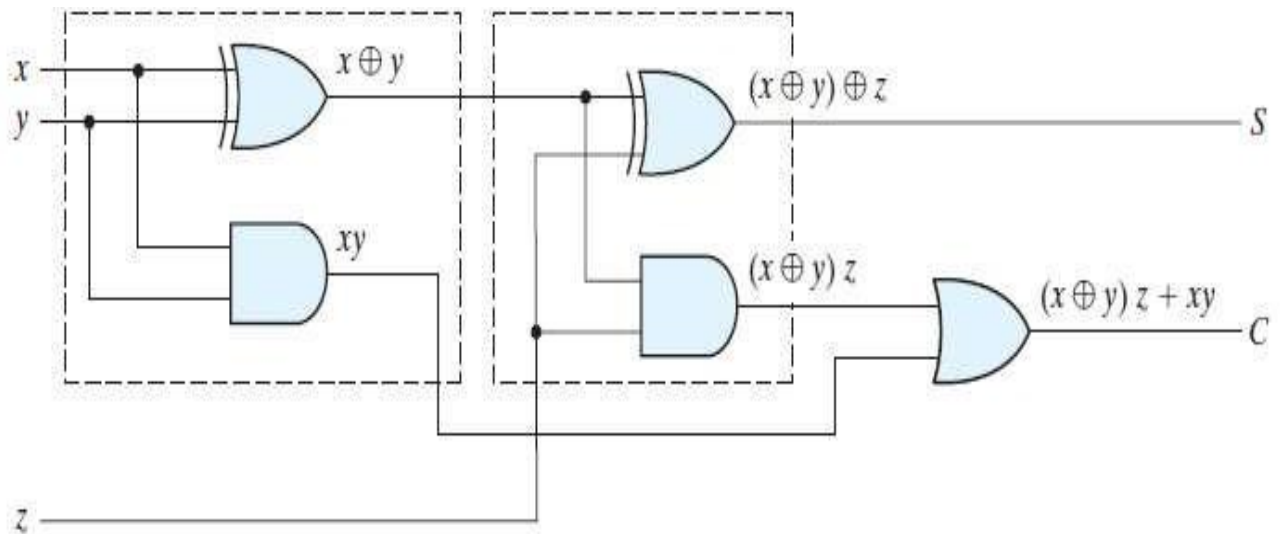
Full Adder can be easily implemented using two Half Adders and an OR Gate.

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

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

The carry output is given by

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



3) Parallel Binary Adders

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.

Four-bit adder/ four-bit binary ripple carry adder:

The input carry to the least significant position is fixed at 0. Figure below shows the 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 . The S outputs generate the required sum bits. An n -bit adder requires n full adders, with each output carry connected to the input carry of the next higher order full 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}

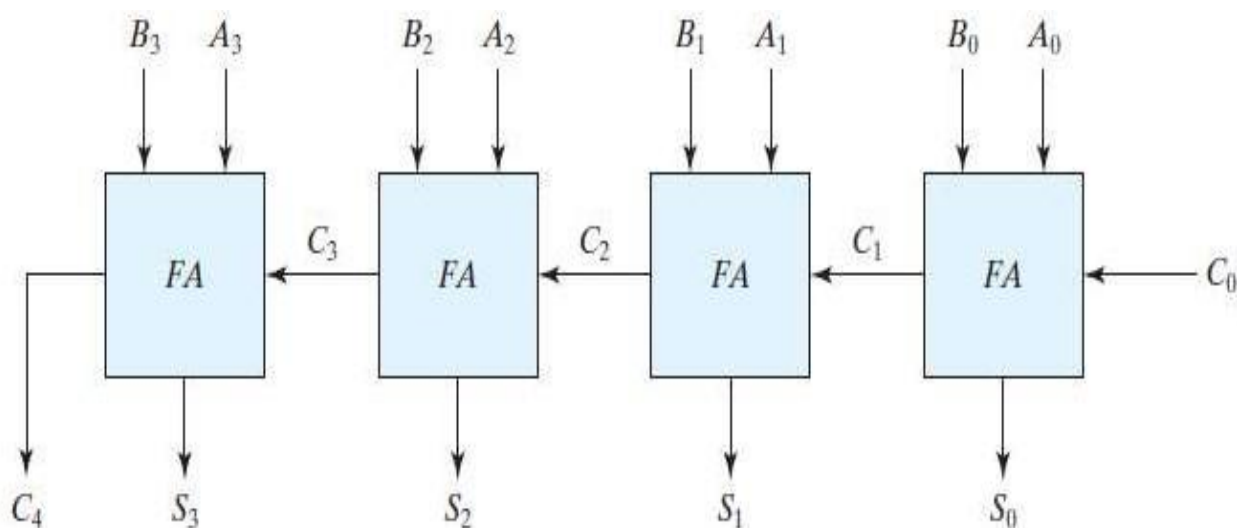


Figure: Four-Bit Adder

The bits are added with full adders, starting from the least significant position (subscript 0), to form the sum bit and carry bit. 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. This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated. All the carries must be generated for the correct sum bits to appear at the outputs.

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. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit. The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders. Since each bit of the sum output depends on the value of the input carry, the value of S_i at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated.

If there are four full adders in the adder, the output carry C_4 would have $2 * 4 = 8$ gate levels from C_0 to C_4 . For an n -bit adder, there are $2n$ gate levels for the carry to propagate from input to output.

The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added. Although the adder—or, for that matter, any combinational circuit—will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is critical. An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. However, physical circuits have a limit to their capability. Another solution is to increase the complexity of the equipment in such a way that the carry delay time is reduced. 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.

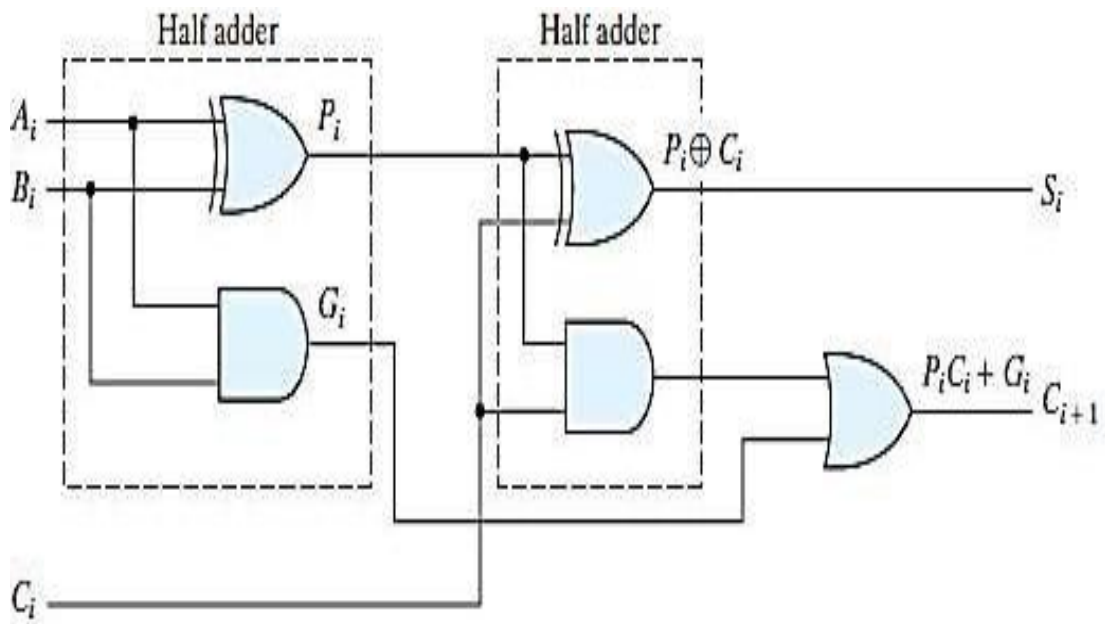


Figure: Full adder with P and G shown

Consider the circuit of the full adder shown in Figure above. If we define two new binary variables

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$

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

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 $i + 1$ (i.e., whether an assertion of C_i will propagate to an assertion of C_{i+1}). We now write the Boolean functions for the carry outputs of each stage and substitute the value of each C_i from the previous equations:

$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$

Since the Boolean function for each output carry is expressed in sum-of-products form, each function can be implemented with one level of AND gates followed by an OR gate (or by a two-level NAND). The three Boolean functions for C_1 , C_2 , and C_3 are implemented in the carry lookahead generator shown in Figure below.

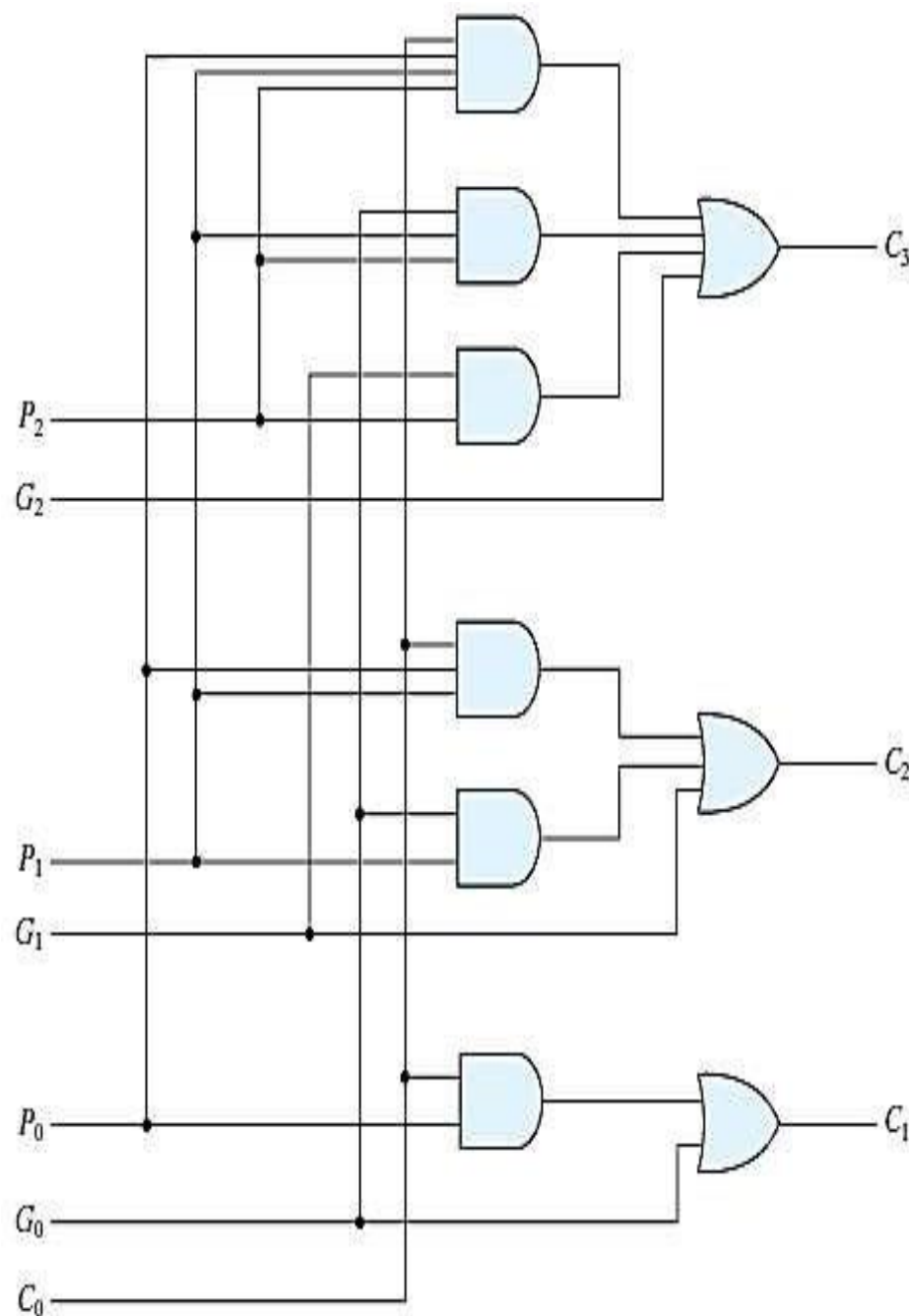


Figure: Logic diagram of carry lookahead generator

Four-bit adder with carry lookahead:

The construction of a four-bit adder with a carry lookahead scheme is shown in Figure below.

Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. The carries are propagated through the carry lookahead generator and applied as inputs to the second exclusive-OR gate. All output carries are generated after a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times. The two-level circuit for the output carry C_4 is not shown. This circuit can easily be derived by the equation-substitution method.

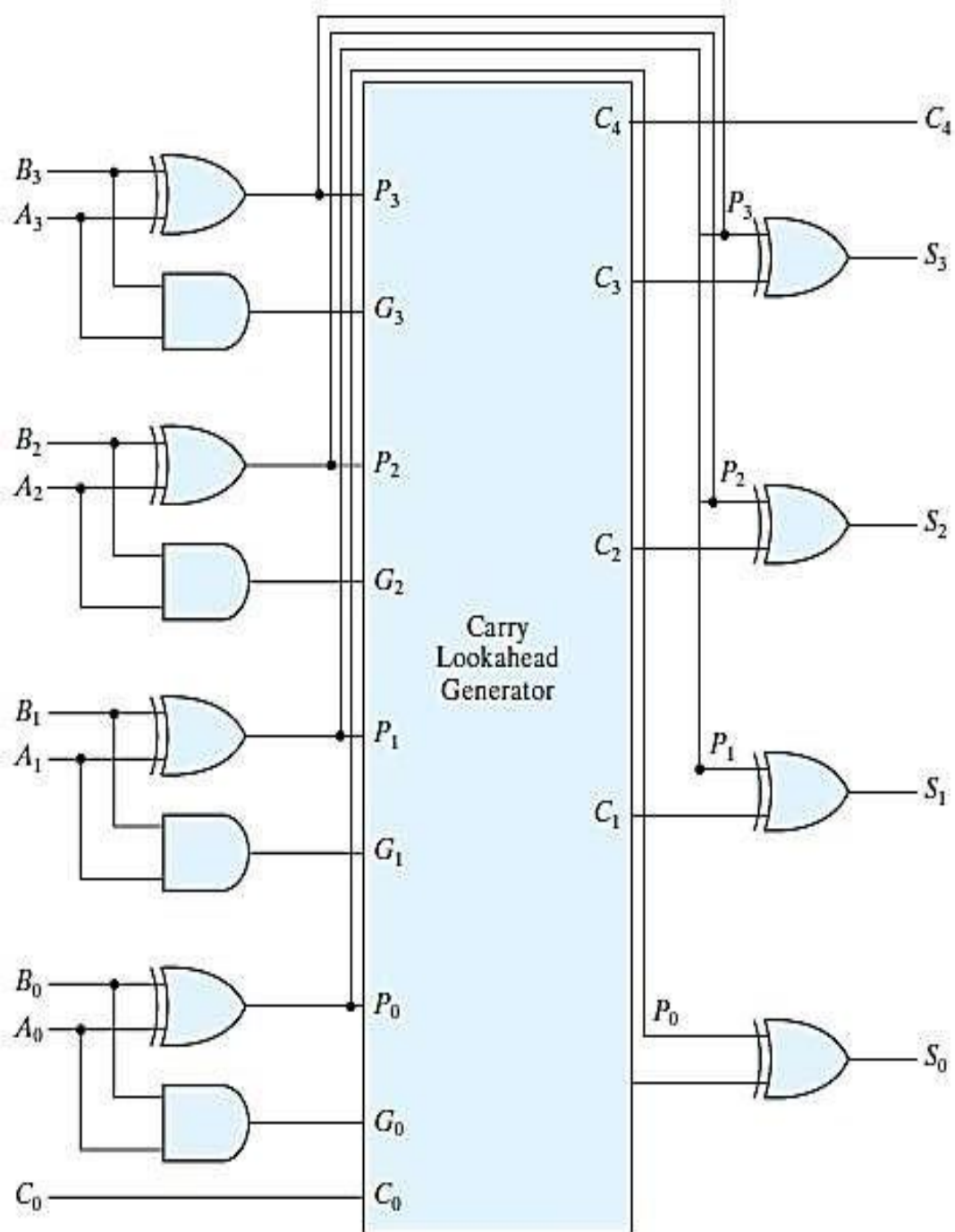


Figure: Four-Bit Adder with Carry Lookahead

Binary Subtraction Circuits

Another basic arithmetic operation to be performed by Digital Computers is the Subtraction. Subtraction is a mathematical operation in which one integer number is deducted from another to obtain the equivalent quantity. The number from which other number is to be deducted is called as 'Minuend' and the number subtracted from the minuend is called 'Subtrahend'.

Similar to the binary addition, binary subtraction is also has four possible basic operations. They are:

$$\begin{aligned}
 0 - 0 &= 0 \\
 0 - 1 &= (\text{Borrow}) 1 \ 1 \\
 1 - 0 &= 1 \\
 1 - 1 &= 0
 \end{aligned}$$

To perform the subtraction “-B”, we can use the *2's complements*, so the subtraction can be converted to addition.

2's complement can be obtained by taking the *1's complement* and adding 1 to the *LSD* bit.

1) *1's complement* can be implemented with inverters.

2) 1 can be added to the sum through the input carry.

The circuit for subtracting A-B consists of an adder with inverters placed between each data input B, and the corresponding input of the full adder. The input carry C_0 must be equal to 1.

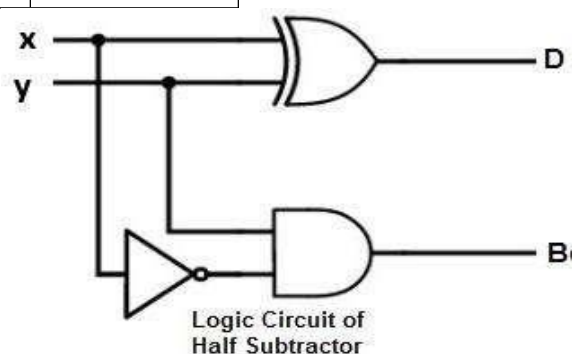
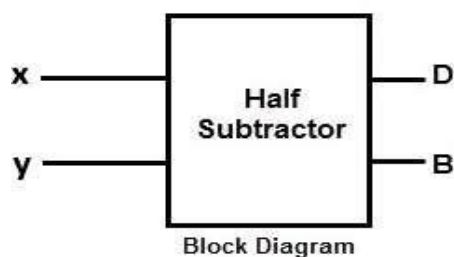
Similar to the adder circuits, basic subtraction circuits are also of two types:

3. Half Subtractor

A Half Subtractor is a multiple output Combinational Logic Circuit that does the subtraction of two 1-bit binary numbers. It has two inputs and two outputs. The two inputs correspond to the two 1-bit binary numbers and the two outputs correspond to the Difference bit and Borrow bit (in contrast to Sum and Carry in Half Adder).

Truth Table:

INPUT		OUTPUT	
x	y	Difference (D)	Borrow (B)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



K- Map:

	y'	y
x'	0	1
x	1	0

$D = x'y + xy'$

	y'	y
x'	0	1
x	0	0

$B = x'y$

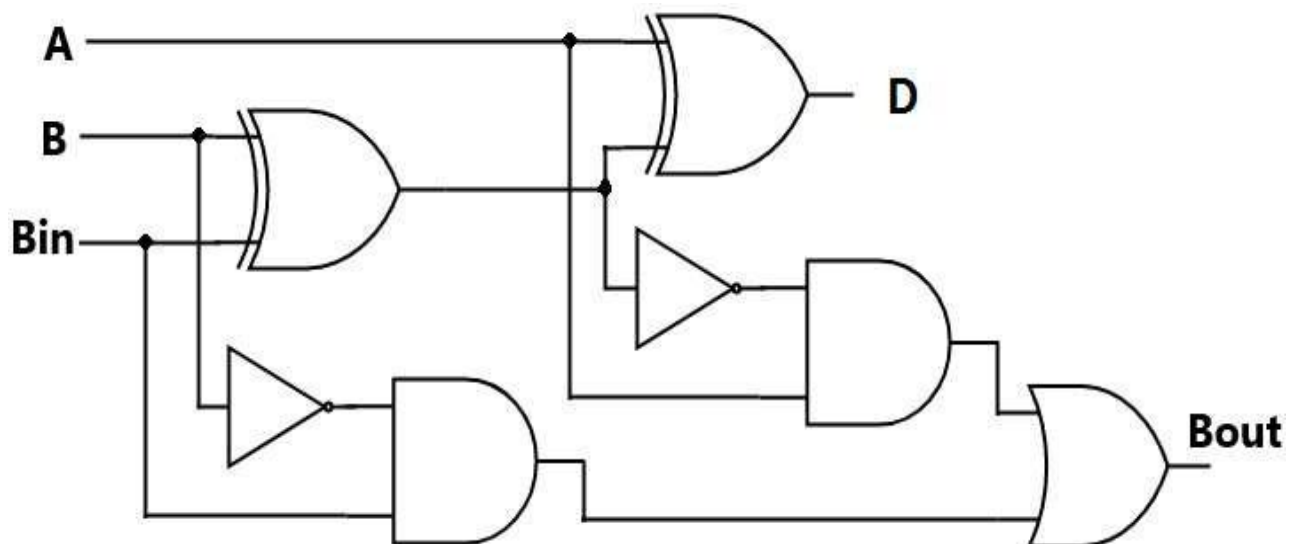
4. Full Subtractor

A Full Subtractor is a combinational logic circuit which performs a subtraction between the two 1-bit binary numbers and it also considers the borrow of the previous bit i.e., whether 1 has been borrowed by the previous minuend bit.

So, a Full Subtractor has three inputs, in which two inputs corresponding to the two bits to be subtracted (minuend A and subtrahend B), and a borrow bit, usually represented as B_{IN} , corresponding to the borrow operation. There are two outputs, one corresponds to the difference D output and the other Borrow output B_O .

The following image shows the block diagram of a full subtractor.

INPUT			OUTPUT	
A	B	B _{IN}	D	B _{OUT}
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



		B Bin			
		00	01	11	10
A	0	0	1	1	1
	1	0	0	1	0

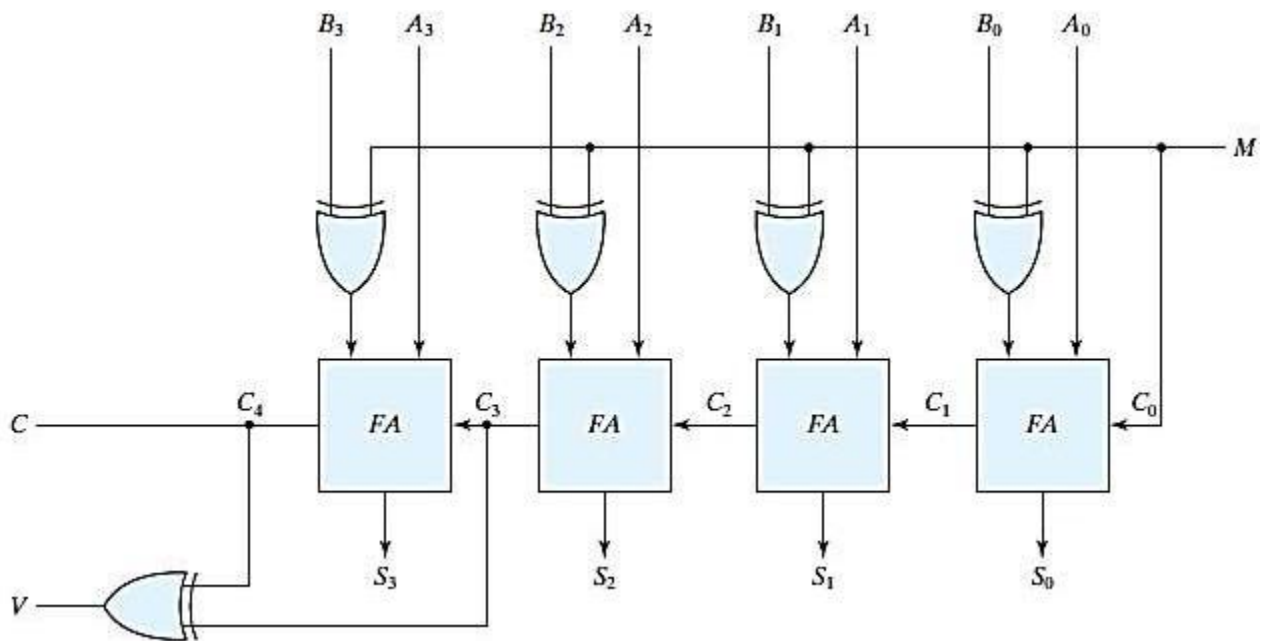
$$Bout = A'Bin + A'B + BBin$$

		B Bin			
		00	01	11	10
A	0	0	1	0	1
	1	1	0	1	0

$$D = A'B'Bin + AB'Bin' + A'BBin' + ABBin$$

Four-bit adder–subtractor (with overflow detection)

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.



The mode input M controls the operation as the following:

If $M=0 \rightarrow$ the circuit is an adder

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

The binary adder-subtractor circuit with outputs C and V is shown in Fig. 4.13. If the two binary numbers are considered to be unsigned, then the C bit detects a carry after addition or a borrow after subtraction. If the numbers are considered to be signed, then the V bit detects an overflow. If $V = 0$ after an addition or subtraction, then no overflow occurred and the n -bit result is correct. If $V = 1$, then the result of the operation contains $n + 1$ bits, but only the rightmost n bits of the number fit in the space available, so an overflow has occurred. The $1n + 12$ th bit is the actual sign and has been shifted out of position.

4.9 Decoders and Encoders

- A decoder is a combinational circuit which has n inputs and 2^n outputs, such that only one output line is activated for each one of the possible combinations of inputs.
- Fig.3.16 shows the general block diagram of the decoder.

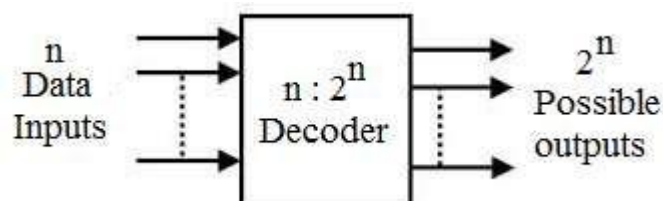


Fig.3.16: General block diagram of decoder

- One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. The outputs of the decoder are nothing but the min terms of 'n' input variables (lines), when it is enabled.

2:4 Decoder

- 2:4 Decoder has two inputs (A & B) and four outputs (Y_3 , Y_2 , Y_1 & Y_0) as shown in the fig.2.4.

Block Diagram:

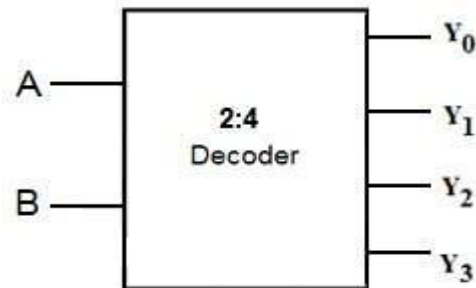


Fig.3.17: Block diagram of 2:4 Decoder

Truth Table:

Inputs		Outputs			
A	B	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

- From Truth table, **Boolean functions** for each output can be written as

$$Y_0 = \bar{A}\bar{B}$$

$$Y_1 = \bar{A}B$$

$$Y_2 = A\bar{B}$$

$$Y_3 = AB$$

Logic Diagram:

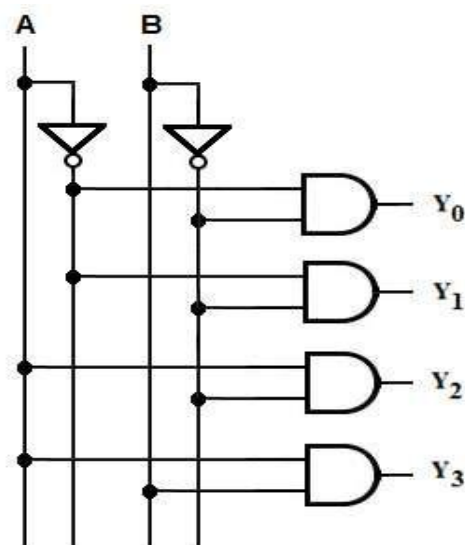


Fig.3.18: Logic diagram of 2:4 Decoder

3:8 Decoder

- 3:8 Decoder has three inputs (A, B and C) and eight outputs ($Y_7, Y_6, Y_5, Y_4, Y_3, Y_2, Y_1$ & Y_0) as shown in the fig.3.19.

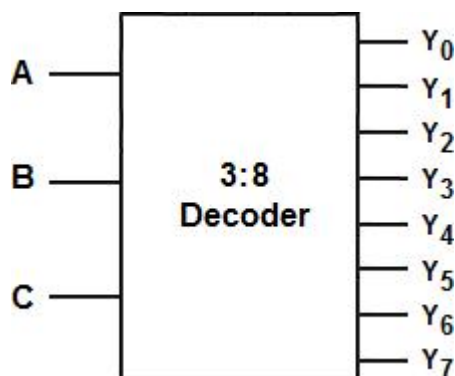


Fig.3.19: A 3-to-8 Line Decoder

Truth Table:

Inputs			Outputs							
A	B	C	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

- From Truth table, **Boolean functions** for each output line can be written as

$$Y_0 = \bar{A}\bar{B}\bar{C}$$

$$Y_1 = \bar{A}\bar{B}C$$

$$Y_2 = \bar{A}B\bar{C}$$

$$Y_3 = \bar{A}BC$$

$$Y_4 = A\bar{B}\bar{C}$$

$$Y_5 = A\bar{B}C$$

$$Y_6 = AB\bar{C}$$

$$Y_7 = ABC$$

- From the above expression it is clear that a 3-to-8 decoder generates all minterms of the three input variables.
- Exactly one of the output lines will be logic 1 for each combination values of the input variables.

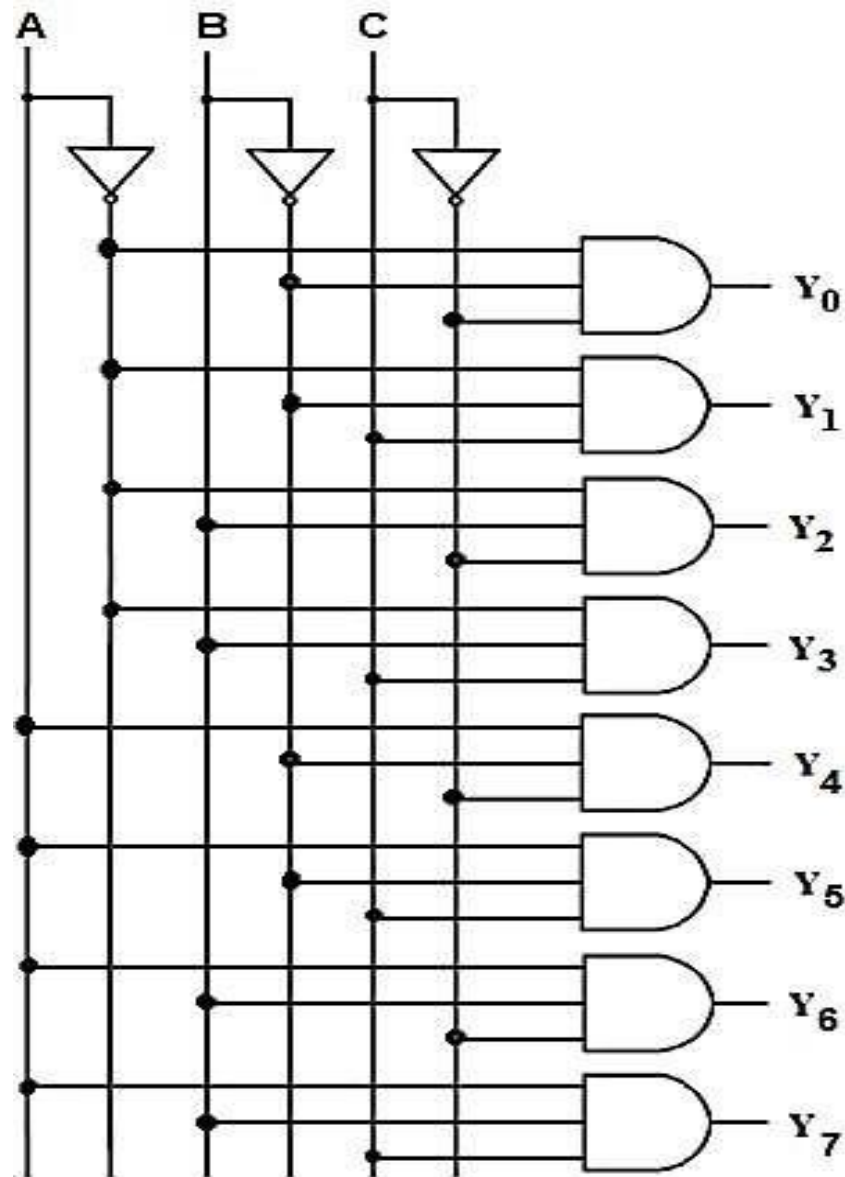


Fig.3.20: Logic diagram of 3-to-8 Decoder

BCD-to-Decimal Decoder / 4-to-10 Line Decoder

- The BCD-to-decimal decoder is also called a 4-to-10 line decoder. It has 4 inputs & 10 outputs as shown in the fig.3.21. Only one output line is active at a time.

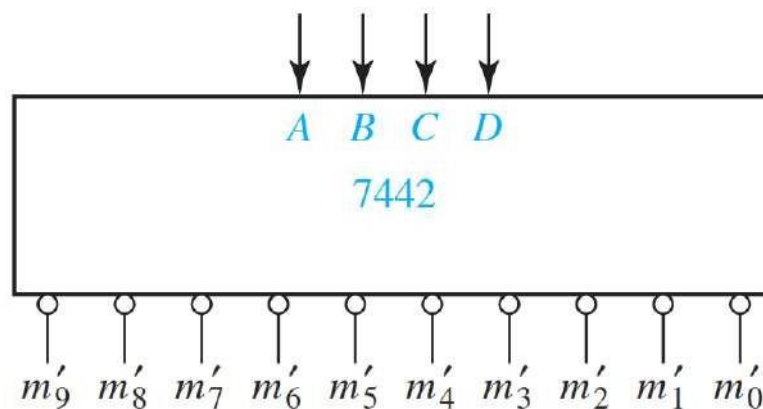


Fig.3.21: 4-to-10 line decoder IC

- A BCD code applied to the 4 inputs, A, B, C, and D, results in a low level at the selected 1-of-10 decimal decoded outputs.

Inputs				Outputs									
A	B	C	D	Y ₉	Y ₈	Y ₇	Y ₆	Y ₅	Y ₄	Y ₃	Y ₂	Y ₁	Y ₀
0	0	0	0	1	1	1	1	1	1	1	1	1	0
0	0	0	1	1	1	1	1	1	1	1	1	0	1
0	0	1	0	1	1	1	1	1	1	1	0	1	1
0	0	1	1	1	1	1	1	1	1	0	1	1	1
0	1	0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	1	1	1	1	1	0	1	1	1	1	1
0	1	1	0	1	1	1	0	1	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1	1	1	1
1	0	0	0	1	0	1	1	1	1	1	1	1	1
1	0	0	1	0	1	1	1	1	1	1	1	1	1

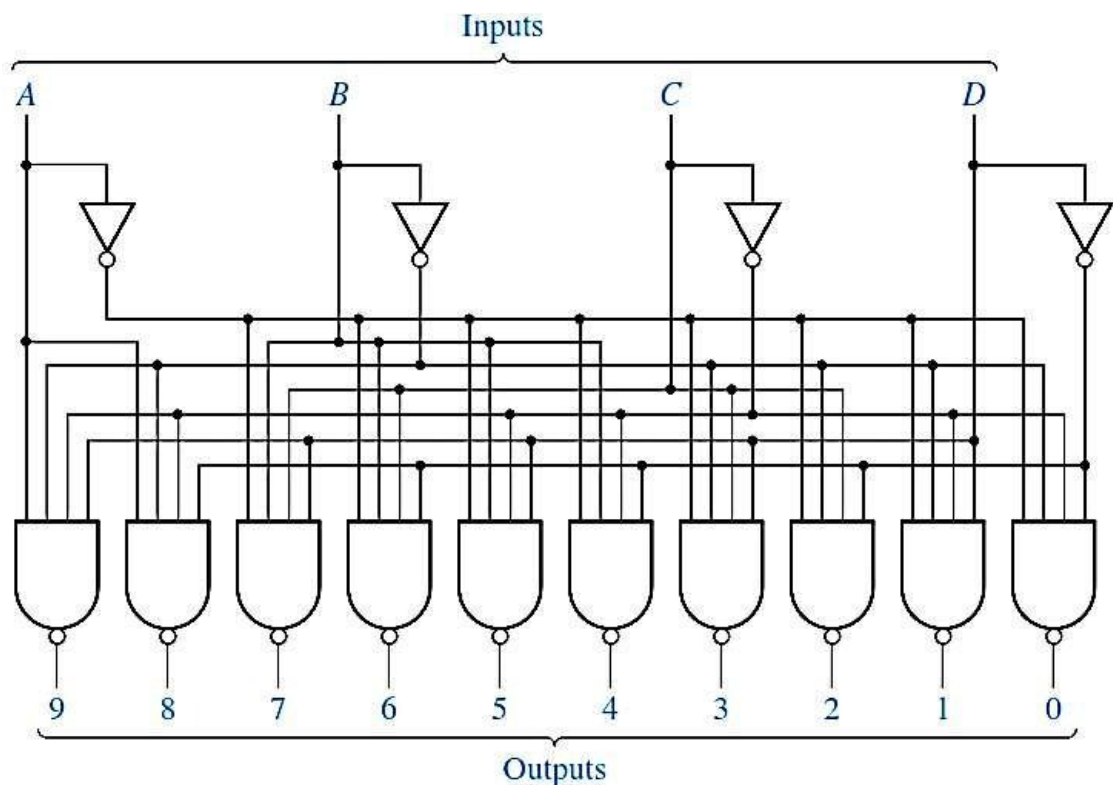


Fig.3.22: Logic diagram of 4-to-10 line decoder

- In general, an n -to- 2^n line decoder generates all 2^n minterms (or maxterms) of the n input variables. The outputs are defined by the equations

$$y_i = m_i, \quad i = 0 \text{ to } 2^n - 1 \text{ (noninverted outputs)}$$

or

$$y_i = \bar{\bar{m}}_i = M_i, \quad i = 0 \text{ to } 2^n - 1 \text{ (inverted outputs)}$$

where m_i is a minterm of the n input variables and M_i is a maxterm.

Realization of boolean function using decoders

- The combination of decoder and external logic gates can be used to implement any boolean functions.
- Since an n-input decoder generates all of the minterms of n variables, n-variable functions can be realized by ORing the selected minterm outputs of the decoder.
- If the decoder outputs are inverted (**active low outputs**), then NAND gates can be used to generate the functions.

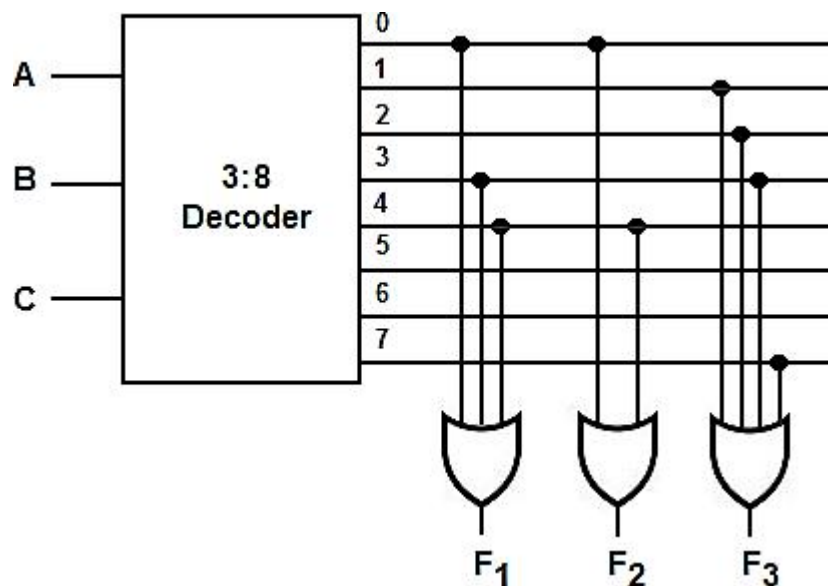
Problems:

1. Implement the following Boolean function using a 3-to-8 decoder and multi-input OR gates simultaneously.

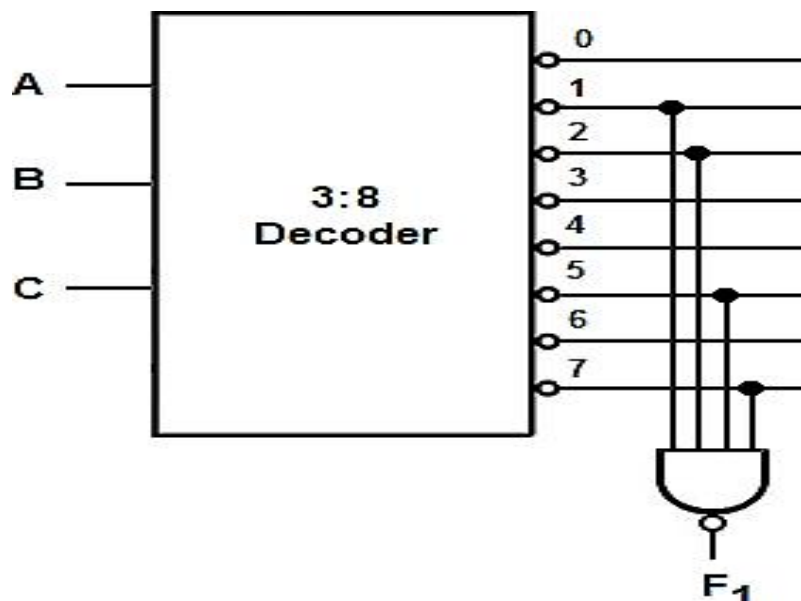
$$F_1(A, B, C) = \sum m(0, 3, 4)$$

$$F_2(A, B, C) = \sum m(0, 5)$$

$$F_3(A, B, C) = \sum m(1, 2, 3, 7)$$

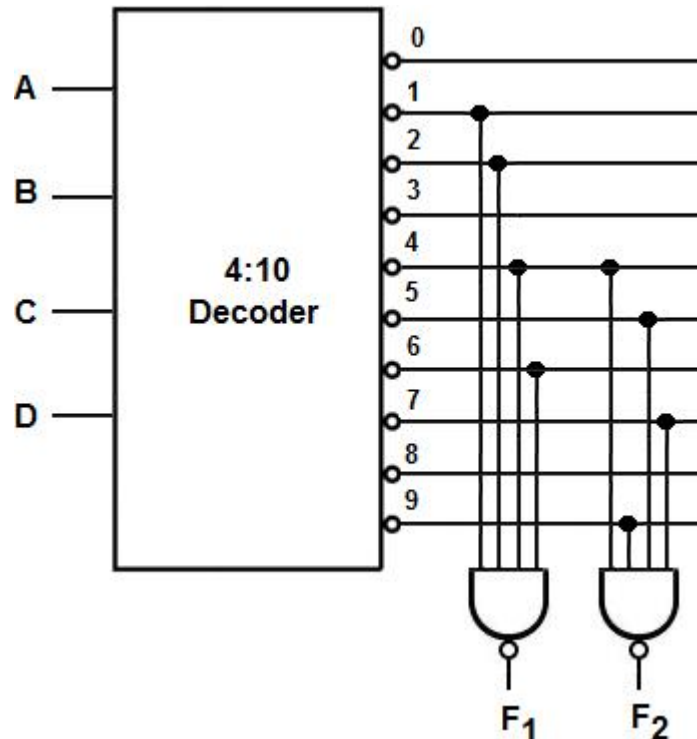


2. Implement the following function $f(A, B, C) = \sum m(1, 2, 5, 7)$ using 3 to 8 decoder with active low outputs.



3. Realize the following functions using a 4-to-10 active low output decoder.

$$F_1(a, b, c, d) = m_1 + m_2 + m_4 + m_6 \text{ and } F_2(a, b, c, d) = m_4 + m_5 + m_7 + m_9$$

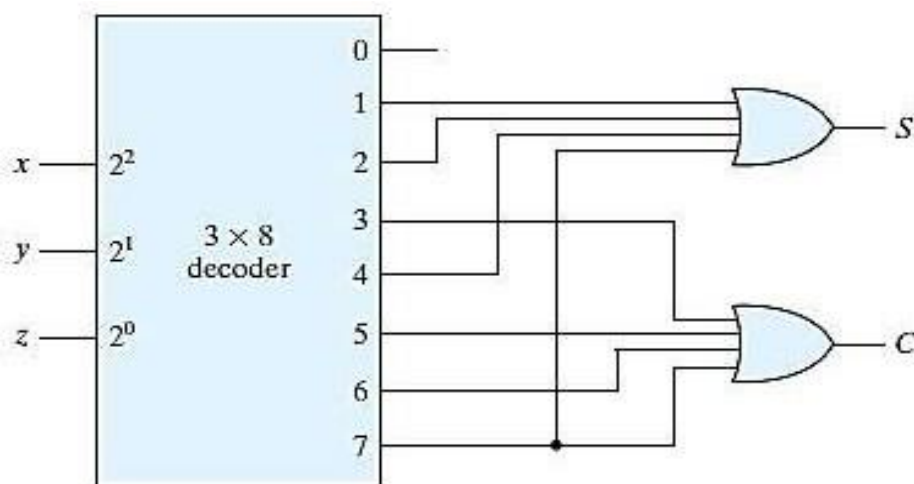


4. Implementation of a full adder with a decoder

From the truth table of the full adder, we obtain the functions for the Combinational circuit in sum-of-minterms form:

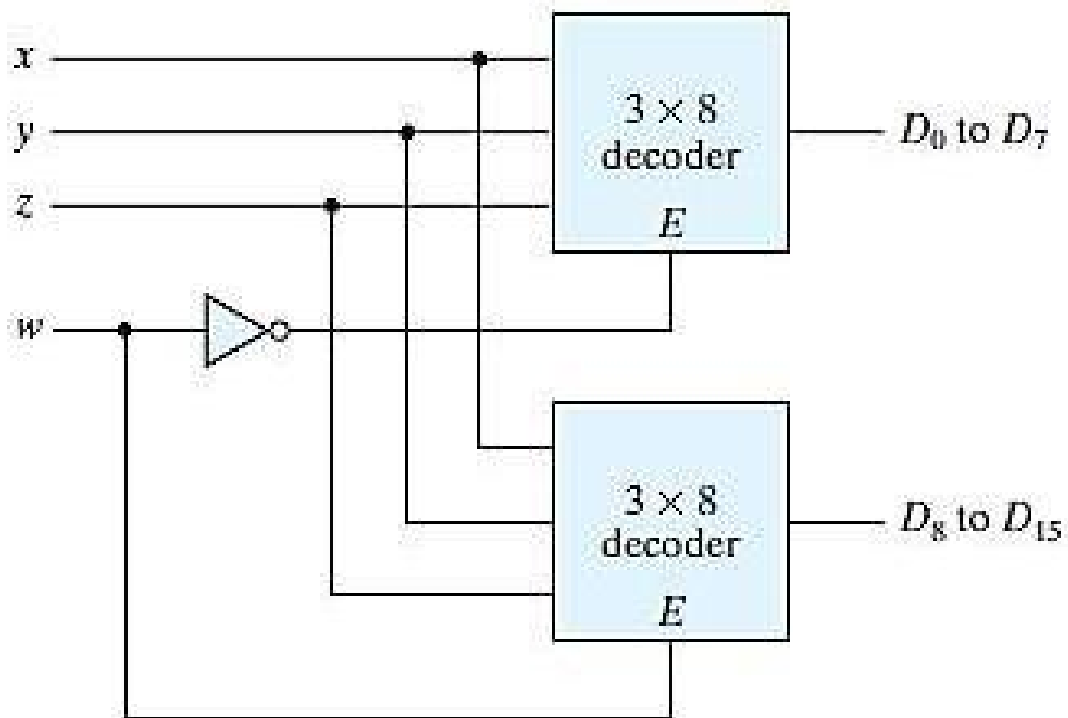
$$S(x, y, z) = \sum (1, 2, 4, 7)$$

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



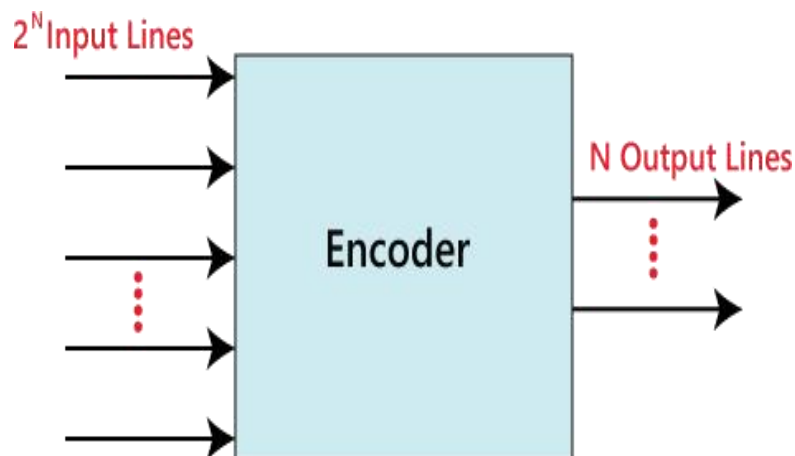
5. Realize 4 * 16 decoder with two 3 * 8 decoders

- 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, while the outputs of the top decoder are all 0's.



4.10 ENCODERS:

- The combinational circuits that change the binary information into N output lines are known as Encoders. The binary information is passed in the form of 2^N input lines. The output lines define the N-bit code for the binary information. In simple words, the Encoder performs the reverse operation of the Decoder. At a time, only one input line is activated for simplicity. The produced N-bit output code is equivalent to the binary information.

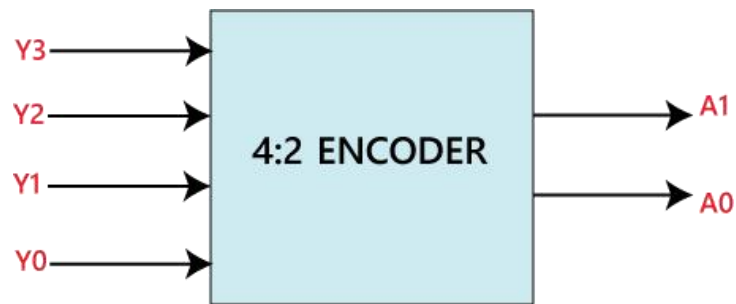


- There are various types of encoders which are as follows:

4 to 2 line Encoder:

In 4 to 2 line encoder, there are total of four inputs, i.e., Y_0 , Y_1 , Y_2 , and Y_3 , and two outputs, i.e., A_0 and A_1 . In 4-input lines, one input-line is set to true at a time to get the respective binary code in the output side. Below are the block diagram and the truth table of the 4 to 2 line encoder.

Block Diagram:



Truth Table:

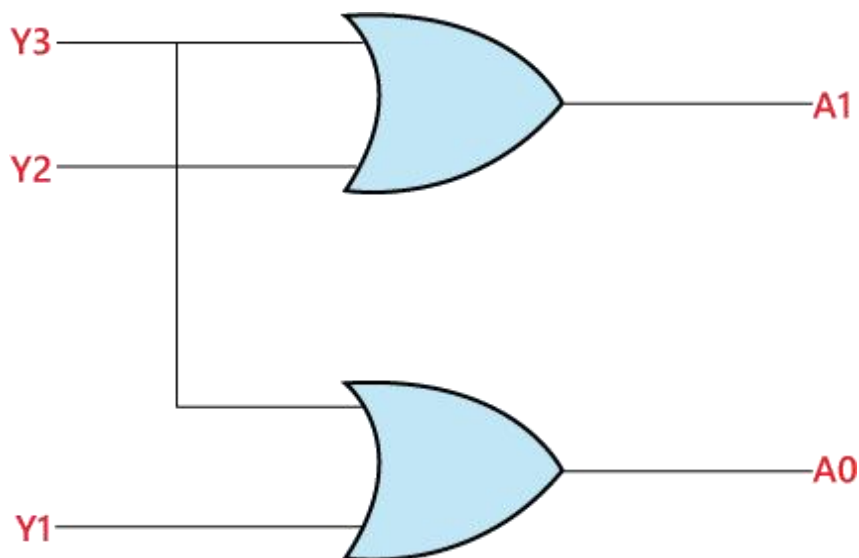
INPUTS				OUTPUTS	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

The logical expression of the term A_0 and A_1 is as follows:

$$A_1 = Y_3 + Y_2$$

$$A_0 = Y_3 + Y_1$$

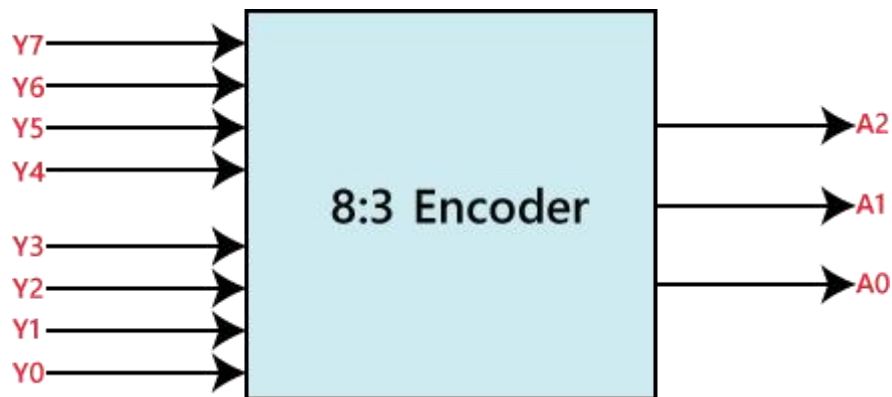
Logical circuit of the above expressions is given below:



8 to 3 line Encoder/ Octal-to-Binary Encoder:

The 8 to 3 line Encoder is also known as **Octal to Binary Encoder**. In 8 to 3 line encoder, there is a total of eight inputs, i.e., $Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6$, and Y_7 and three outputs, i.e., A_0, A_1 , and A_2 . In 8-input lines, one input-line is set to true at a time to get the respective binary code in the output side. Below are the block diagram and the truth table of the 8 to 3 line encoder.

Block Diagram:



Truth Table:

INPUTS								OUTPUTS		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

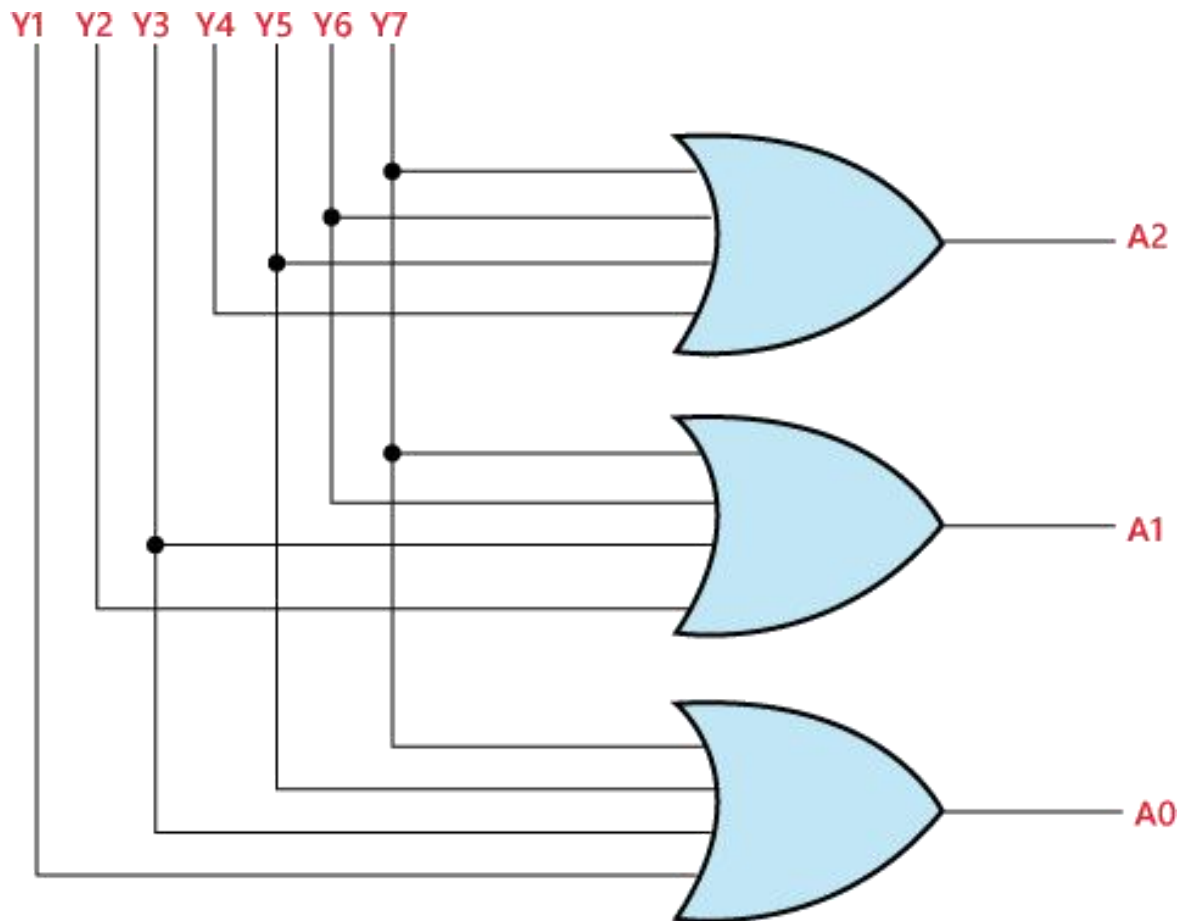
The logical expression of the term A_0, A_1 , and A_2 are as follows:

$$A_2 = Y_4 + Y_5 + Y_6 + Y_7$$

$$A_1 = Y_2 + Y_3 + Y_6 + Y_7$$

$$A_0 = Y_7 + Y_5 + Y_3 + Y_1$$

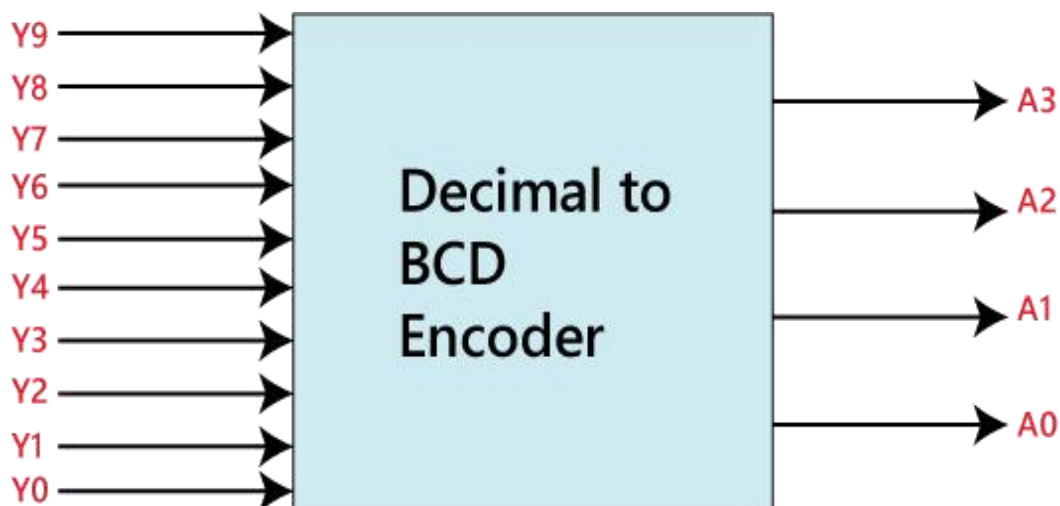
Logical circuit of the above expressions is given below:



Decimal to BCD Encoder

The Octal to Binary Encoder is also known as **10 to 4 line Encoder**. In 10 to 4 line encoder, there are total of ten inputs, i.e., $Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7, Y_8$, and Y_9 and four outputs, i.e., A_0, A_1, A_2 , and A_3 . In 10-input lines, one input-line is set to true at a time to get the respective **BCD code** in the output side. The block diagram and the truth table of the decimal to BCD encoder are given below.

Block Diagram:



Truth Table:

INPUTS										OUTPUTS			
Y_9	Y_8	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0	1

The logical expression of the term A_0 , A_1 , A_2 , and A_3 is as follows:

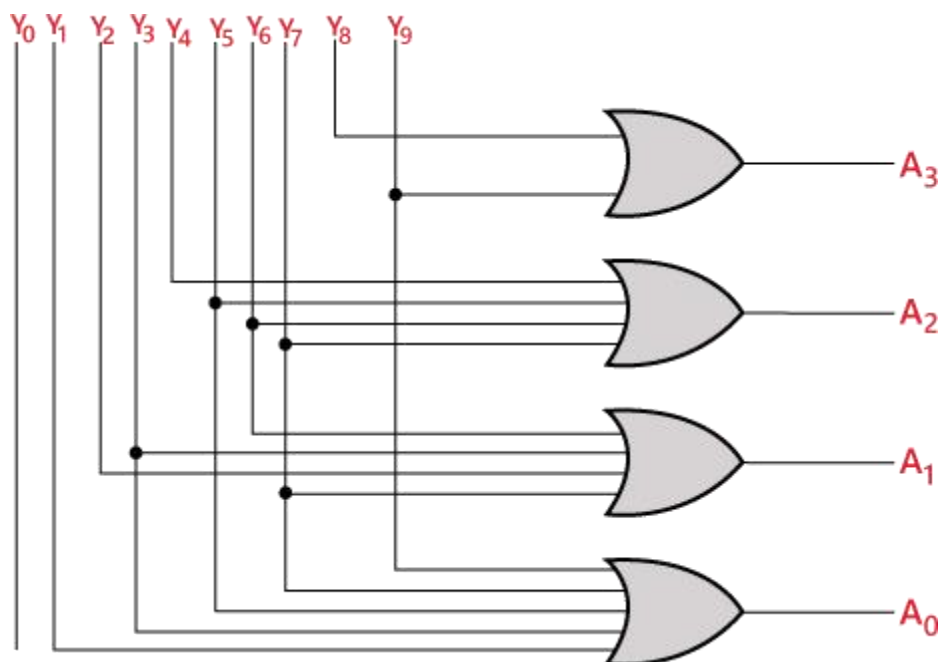
$$A_3 = Y_9 + Y_8$$

$$A_2 = Y_7 + Y_6 + Y_5 + Y_4$$

$$A_1 = Y_7 + Y_6 + Y_3 + Y_2$$

$$A_0 = Y_9 + Y_7 + Y_5 + Y_3 + Y_1$$

Logical circuit of the above expressions is given below:



Priority Encoder:

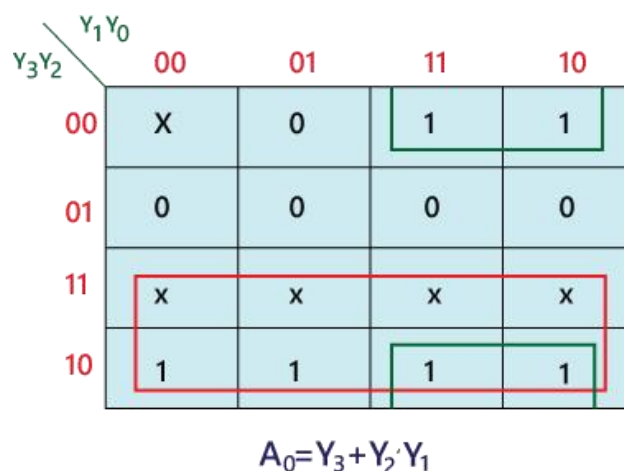
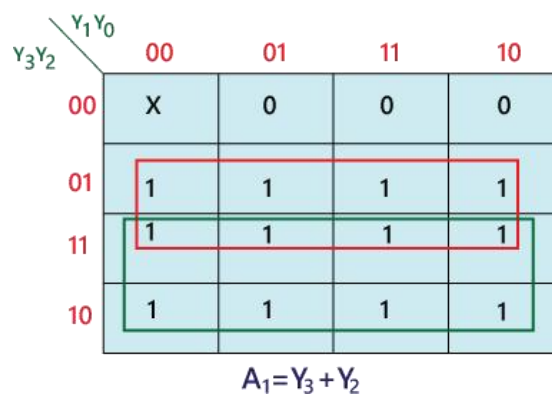
4 to 2 line Priority Encoder:

In this priority encoder, there are total of 4 inputs, i.e., Y_0 , Y_1 , Y_2 , and Y_3 , and two outputs, i.e., A_0 and A_1 . The Y_3 has high and Y_0 has low priority inputs. When more than one input is '1' at the same time, the output will be the (binary) code corresponding to the higher priority input. Below is the truth table of the 4 to 2 line priority encoder.

Truth Table:

INPUTS				OUTPUTS		
Y_3	Y_2	Y_1	Y_0	A_1	A_0	V
0	0	0	0	X	x	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	x	X	1	0	1
1	X	x	X	1	1	1

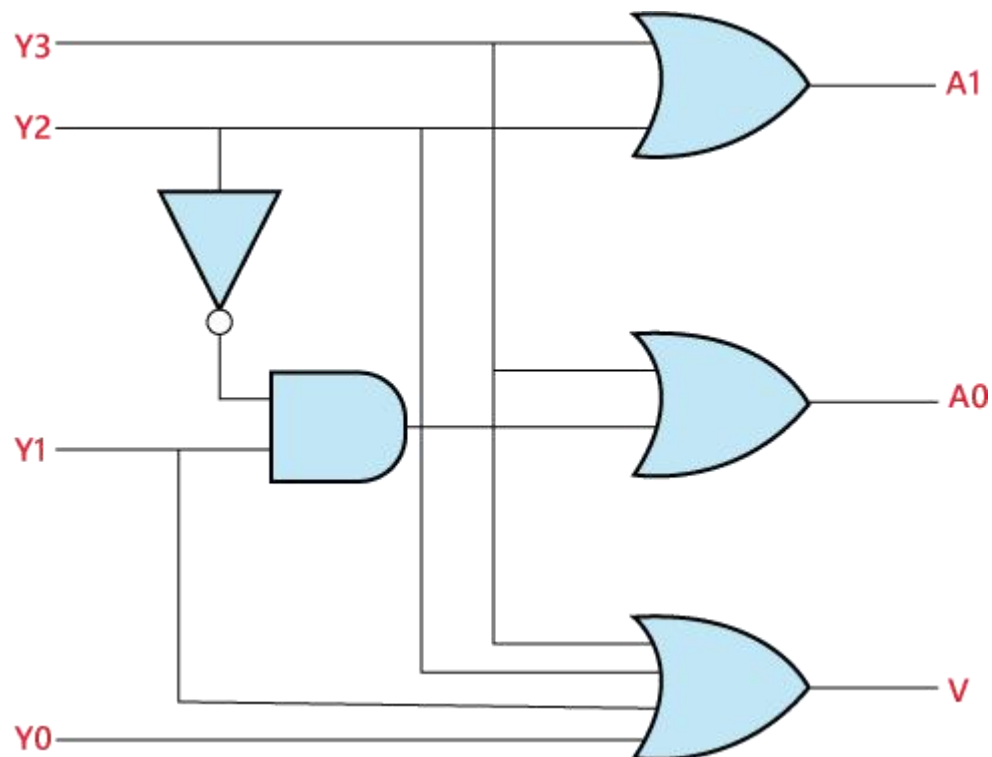
The logical expression of the term A_0 and A_1 can be found using **K-map** as:



$$A_1 = Y_3 + Y_2$$

$$A_0 = Y_3 + Y_2' \cdot Y_1$$

Logical circuit of the above expressions is given below:



8 to 3 line Priority Encoder / Octal-to-Binary Priority Encoder

- An encoder performs the inverse function of a decoder. Fig.3.23 shows an 8-to-3 priority encoder with eight inputs Y_0 through Y_7 and three outputs A, B and C.

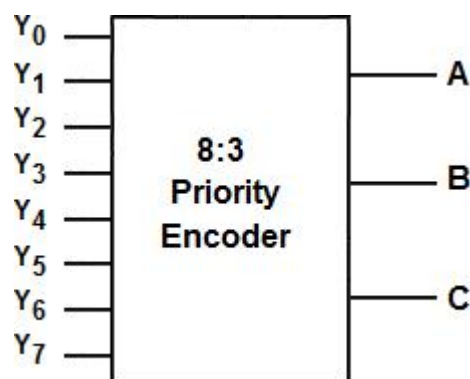


Fig.3.23: 8:3 Priority Encoder

Inputs								Outputs		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A	B	C
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	1	X	X	0	1	0

0	0	0	0	1	X	X	X	0	1	1
0	0	0	1	X	X	X	X	1	0	0
0	0	1	X	X	X	X	X	1	0	1
0	1	X	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	X	1	1	1

- If input Y_i is 1 and the other inputs are 0, then the ABC outputs represent a binary number equal to i . For example, if $Y_3 = 1$, then $ABC = 011$.
- If more than one input is 1 at the same time, then the output can be defined using a priority scheme.
- If more than one input is 1, the highest numbered input determines the output. For example, if inputs Y_1 , Y_4 , and Y_5 are 1, the output is $ABC = 101$.
- The X's in the table are don't-cares; for example, if Y_5 is 1, the inputs Y_0 through Y_4 are don't cares and it can take a value 0 or 1.

4.11 MULTIPLEXERS:

- A multiplexer (or data selector, abbreviated as MUX) has a group of data inputs (2^n) and a group of control inputs (n) (also called as select inputs).
- The control inputs are used to select one of the data inputs and connect it to the output terminal.

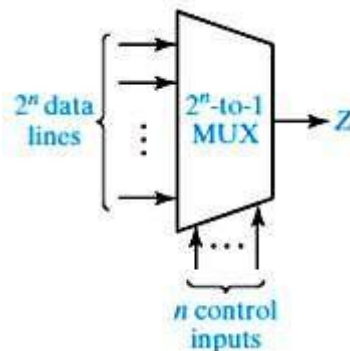


Fig.3.6: General block diagram of $2^n : 1$ multiplexer

- A general block diagram of $2^n : 1$ multiplexer is shown in the fig.3.6.
- A 2 to 1 multiplexer requires 1 select input, 4 to 1 multiplexers require 2 select inputs and 8 to 1 multiplexers require 3 select inputs.

2:1 Multiplexer

- A 2 to 1 multiplexer has 2 data inputs, 1 select input and 1 output.

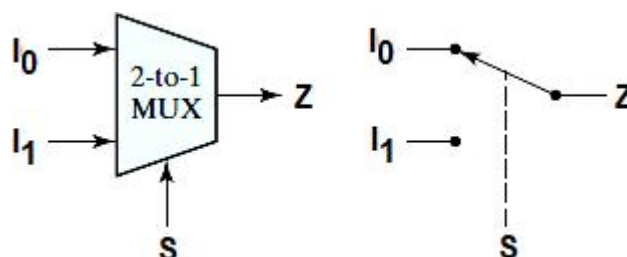


Fig.3.7: 2 to 1 Multiplexer and Switch Analog

- When the select (control) input A is 0, the switch is in the upper position and the MUX output is $Z = I_0$.
- When the select (control) input A is 1, the switch is in the lower position and the MUX output is $Z = I_1$.
- In other words, a MUX acts like a switch that selects one of the data inputs (I_0 or I_1) and transmits it to the output.

Truth Table:

Select (S)	Output (Z)
0	I_0
1	I_1

- The logic equation for the 2-to-1 MUX can be written as:

$$Z = SI_0 + SI_1$$

Logic Diagram:

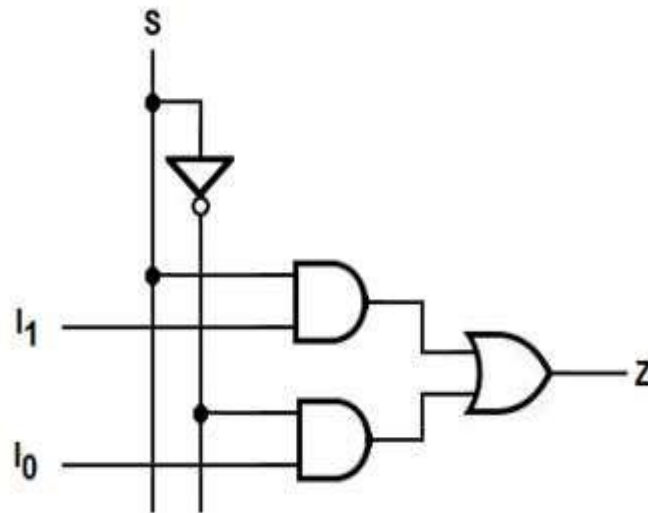


Fig.3.8: Logic diagram of 2:1 Multiplexer

4:1 Multiplexer

- A 4 to 1 multiplexer has 4 data inputs, 2 select inputs and 1 output.
- The 4 to 1 MUX acts like a four-position switch that transmits one of the four inputs to the output.

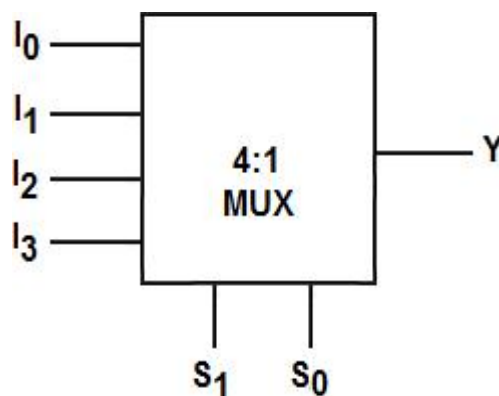


Fig.3.9: 4 to 1 Multiplexer

- Two select (control) inputs (S_1 and S_0) are needed to select one of the four inputs. If the control inputs are $S_1S_0 = 00$, the output is I_0 ; similarly, for the control inputs 01, 10, and 11 give outputs of I_1 , I_2 , and I_3 , respectively.

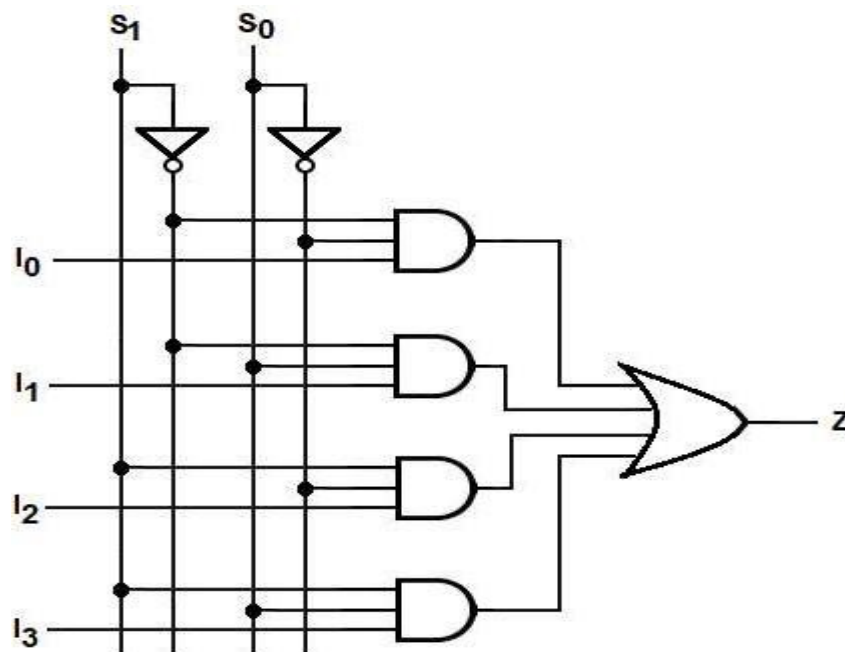
Truth Table:

Select (S_1)	Select (S_0)	Output (Z)
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

- The logic equation for 4-to-1 multiplexer can be written as

$$Z = S_1 S_0 I_0 + S_1 S_0 I_1 + S_1 S_0 I_2 + S_1 S_0 I_3$$

Logic Diagram:



8:1 Multiplexer

- An 8 to 1 multiplexer has 8 data inputs, 3 select inputs and 1 output. 8 to 1 MUX selects one of eight data inputs using three select inputs

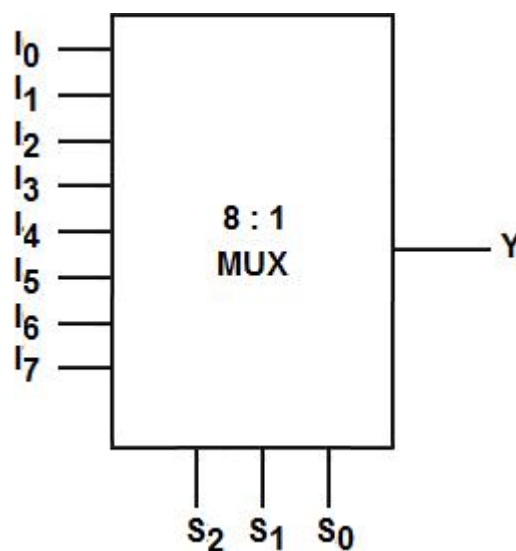


Fig.3.10: 8 to 1 Multiplexe

- The 8-to-1 MUX acts like an eight-position switch that transmits one of the eight inputs to the output.

Truth Table:

Select (S_2)	Select (S_1)	Select (S_0)	Output (Z)
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

- The logic equation for the 8 to 1 MUX can be written as:

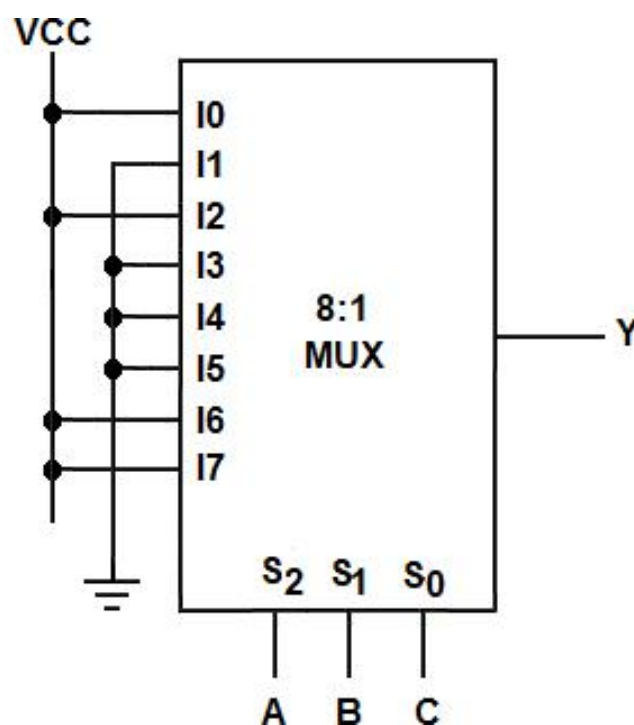
$$Z = S_2S_1S_0I_0 + S_2S_1S_0I_1 + S_2S_1S_0I_2 + S_2S_1S_0I_3 + S_2S_1S_0I_4 + S_2S_1S_0I_5 + S_2S_1S_0I_6 + S_2S_1S_0I_7$$

Problems:

1. Implement the given Boolean function using 8: 1 multiplexer

$$F(A, B, C) = \sum m(0, 2, 6, 7)$$

- Select inputs A, B and C as select lines and connect A, B and C to the select lines of 8:1 multiplexer.
- Connect data input terminals corresponding to the minterms present to logic 1 and connect the remaining data input terminals to logic 0.



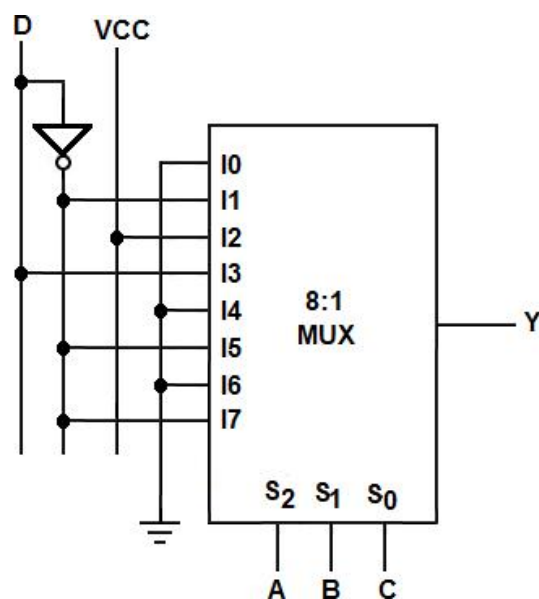
2. Implement the following Boolean function using 8 to 1 multiplexer.

$$Y = F(A, B, C, D) = \sum m(2, 4, 5, 7, 10, 14)$$

- Select inputs A, B and C as select lines and connect A, B and C to the select lines of 8:1 multiplexer.
- Select input D as data input and express output Y in terms of D input as shown in the table.

Truth Table:

A	B	C	D	Y	Data inputs for 8:1 MUX
0	0	0	0	0	0
0	0	0	1	0	
0	0	1	0	1	$\bar{D}$
0	0	1	1	0	
0	1	0	0	1	1
0	1	0	1	1	
0	1	1	0	0	D
0	1	1	1	1	
1	0	0	0	0	0
1	0	0	1	0	
1	0	1	0	1	$\bar{D}$
1	0	1	1	0	
1	1	0	0	0	0
1	1	0	1	0	
1	1	1	0	1	$\bar{D}$
1	1	1	1	0	



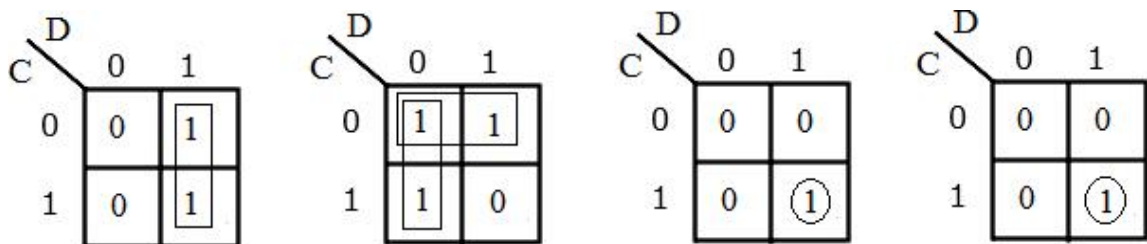
3. Implement the following Boolean function using 4 to 1 multiplexer with A and B as select lines.

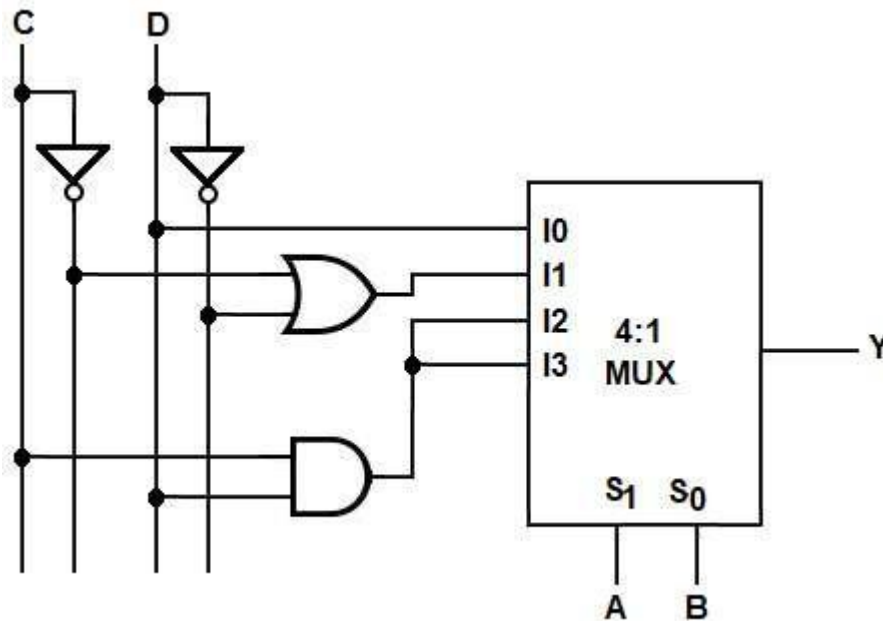
$$Y = F(A, B, C, D) = \sum m(1, 3, 4, 5, 6, 11, 15)$$

- Select input C and D as data inputs and express output Y in terms of C and D inputs.

A	B	C	D	Y	Data Inputs for 4:1 MUX
0	0	0	0	0	D
0	0	0	1	1	
0	0	1	0	0	
0	0	1	1	1	
0	1	0	0	1	$\bar{C} + \bar{D}$
0	1	0	1	1	
0	1	1	0	1	
0	1	1	1	0	
1	0	0	0	0	CD
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	1	
1	1	0	0	0	CD
1	1	0	1	0	
1	1	1	0	0	
1	1	1	1	1	

- Using K-map obtain expressions for data inputs in terms of inputs C and D





4. Realize $Y = AB + BC + ABC$ using an 8 to 1 multiplexer. Also, realize the same with a 4 to 1 multiplexer.

- Convert the expression into decimal notation

$$Y = AB + BC + ABC$$

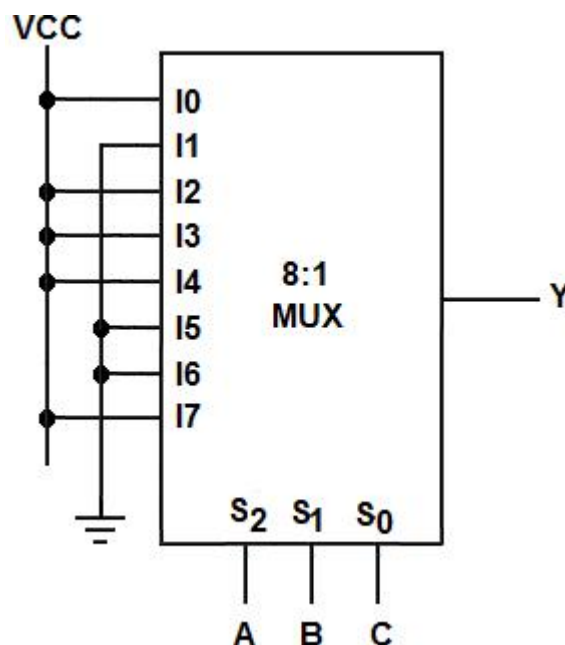
$$Y = AB(C + \bar{C}) + BC(A + \bar{A}) + ABC$$

$$Y = ABC + \bar{A}BC + ABC + AB\bar{C} + ABC$$

$$Y = \sum m(0, 2, 3, 4, 7)$$

8 to 1 implementation:

- Select inputs A, B and C as select lines and connect A, B and C to the select lines of 8:1 multiplexer.
- Connect data input terminals corresponding to the minterms (0, 2, 3, 4, 7) present to logic 1 and connect the remaining data input terminals to logic 0.

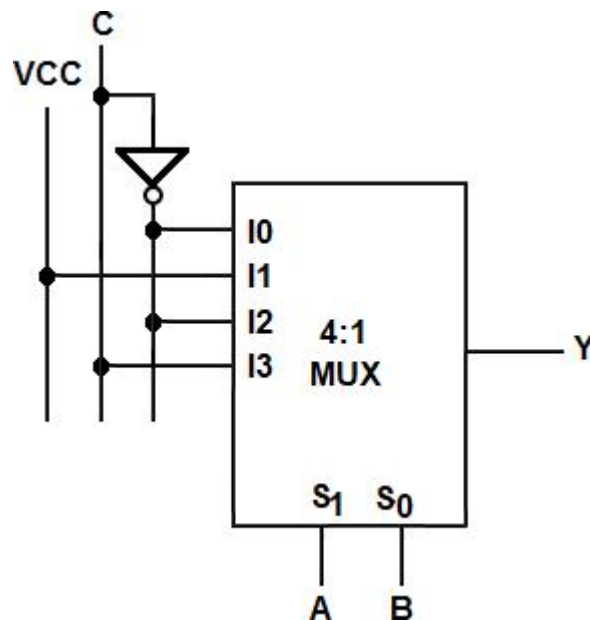


4 to 1 implementation:

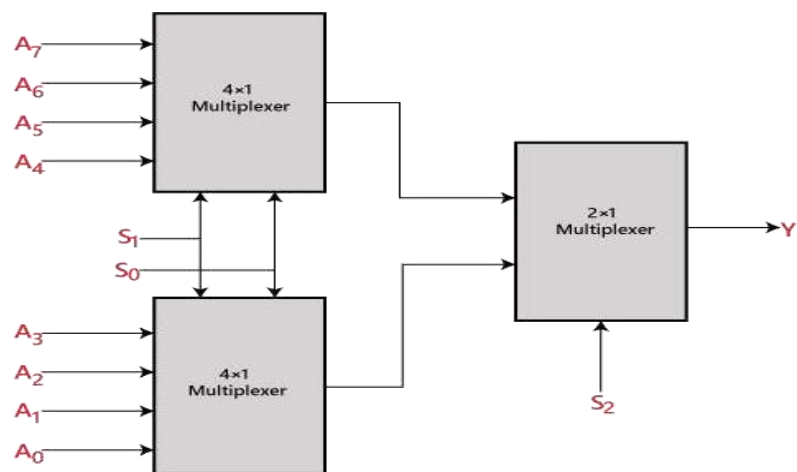
- Select inputs A and B as select lines and connect A and B to the select lines of 4:1 multiplexer.

- Select input C as data input and express output Y in terms of C input.

A	B	C	Y	Data Inputs for 4:1 MUX
0	0	0	1	$\bar{C}$
0	0	1	0	
0	1	0	1	1
0	1	1	1	
1	0	0	1	$\bar{C}$
1	0	1	0	
1	1	0	0	C
1	1	1	1	



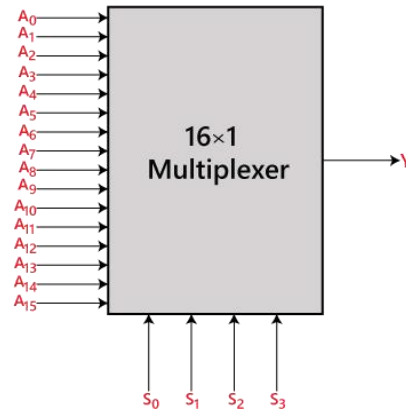
8 × 1 multiplexer using 4 × 1 and 2 × 1 multiplexer



16 × 1 multiplexer

In the 16 to 1 multiplexer, there are total of 16 inputs, i.e., $A_0, A_1, \dots, A_{15}$, 4 selection lines, i.e., S_0, S_1, S_2 , and S_3 and single output, i.e., Y . On the basis of the combination of inputs that are present at the selection lines S_0, S_1 , and S_2 , one of these 16 inputs will be connected to the output. The block diagram and the truth table of the 16×1

Block Diagram:



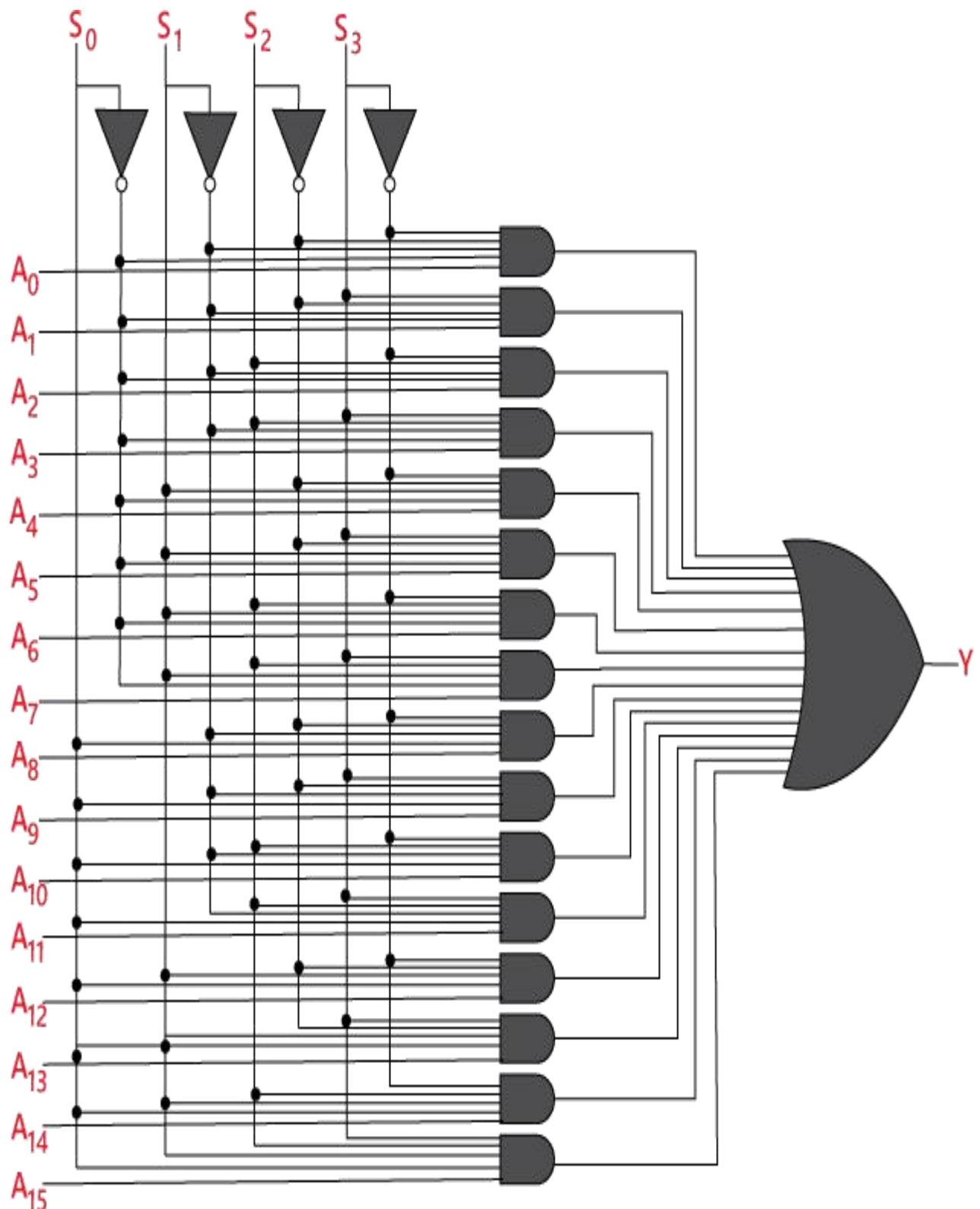
Truth Table:

INPUTS				Output
S_0	S_1	S_2	S_3	Y
0	0	0	0	A_0
0	0	0	1	A_1
0	0	1	0	A_2
0	0	1	1	A_3
0	1	0	0	A_4
0	1	0	1	A_5
0	1	1	0	A_6
0	1	1	1	A_7
1	0	0	0	A_8
1	0	0	1	A_9
1	0	1	0	A_{10}
1	0	1	1	A_{11}
1	1	0	0	A_{12}
1	1	0	1	A_{13}
1	1	1	0	A_{14}
1	1	1	1	A_{15}

The logical expression of the term Y is as follows:

$$Y = A_0.S_0'.S_1'.S_2'.S_3' + A_1.S_0'.S_1'.S_2'.S_3 + A_2.S_0'.S_1'.S_2.S_3' + A_3.S_0'.S_1'.S_2.S_3 + A_4.S_0'.S_1.S_2'.S_3' + A_5.S_0'.S_1.S_2'.S_3 + A_6.S_1.S_2.S_3' + A_7.S_0'.S_1.S_2.S_3 + A_8.S_0.S_1'.S_2'.S_3' + A_9.S_0.S_1'.S_2'.S_3 + A_{10}.S_0.S_1'.S_2.S_3' + A_{11}.S_0.S_1'.S_2.S_3 + A_{12}.S_0.S_1.S_2'.S_3' + A_{13}.S_0.S_1.S_2'.S_3 + A_{14}.S_0.S_1.S_2.S_3' + A_{15}.S_0.S_1.S_2.S_3$$

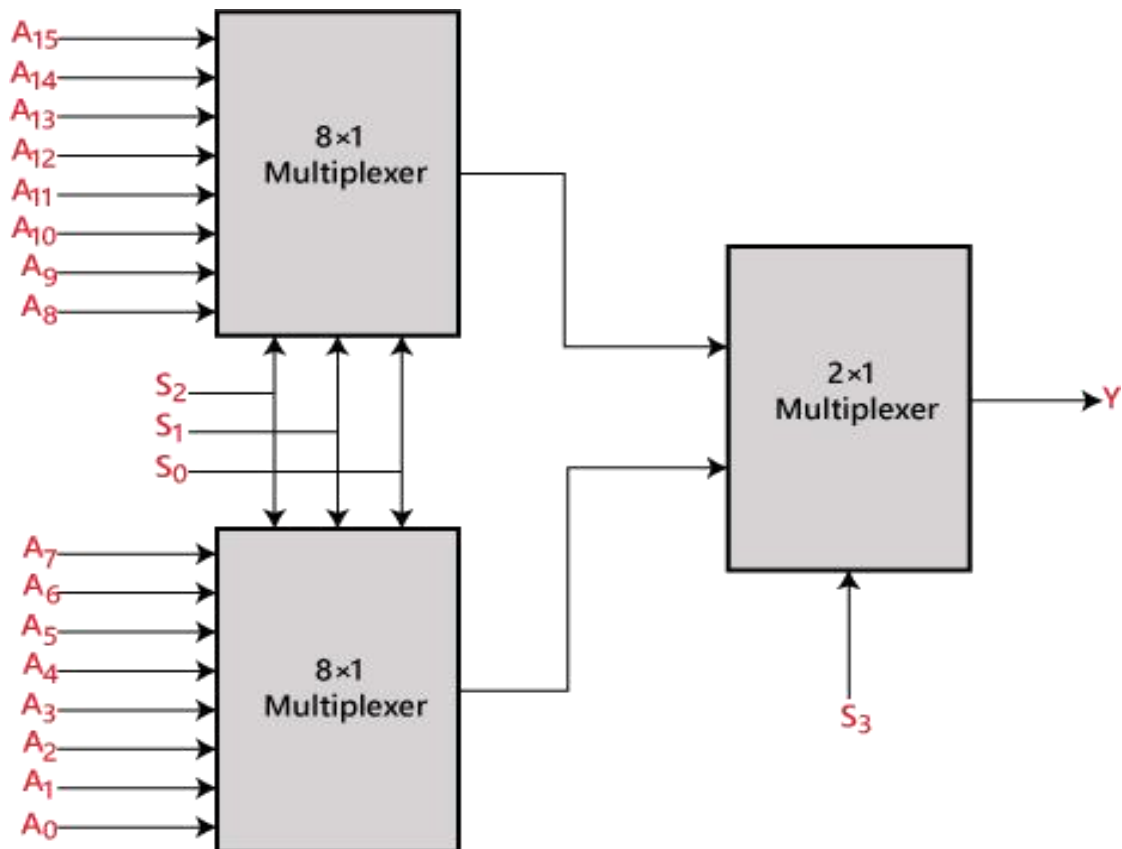
Logical circuit of the above expression is given below:



16×1 multiplexer using 8×1 and 2×1 multiplexer

We can implement the 16×1 multiplexer using a lower order multiplexer. To implement the 8×1 multiplexer, we need two 8×1 multiplexers and one 2×1 multiplexer. The 8×1 multiplexer has 3 selection lines, 4 inputs, and 1 output. The 2×1 multiplexer has only 1 selection line.

For getting 16 data inputs, we need two 8 × 1 multiplexers. The 8×1 multiplexer produces one output. So, in order to get the final output, we need a 2×1 multiplexer. The block diagram of 16×1 multiplexer using 8×1 and 2×1 multiplexer is given below.



De-multiplexer

A De-multiplexer is a combinational circuit that has only 1 input line and 2^N output lines. Simply, the multiplexer is a single-input and multi-output combinational circuit. The information is received from the single input lines and directed to the output line. On the basis of the values of the selection lines, the input will be connected to one of these outputs. De-multiplexer is opposite to the multiplexer.

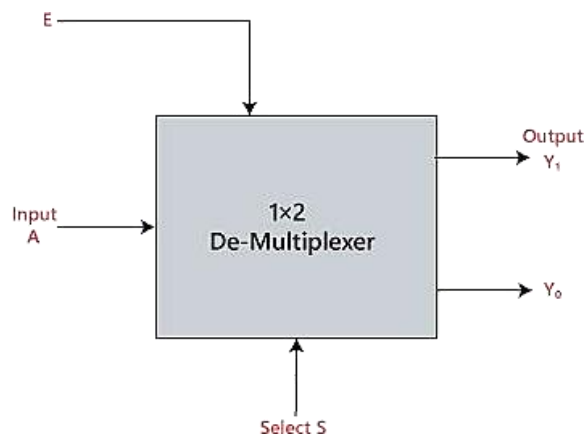
Unlike encoder and decoder, there are n selection lines and 2^n outputs. So, there is a total of 2^n possible combinations of inputs. De-multiplexer is also treated as **De-mux**.

There are various types of De-multiplexer which are as follows:

1×2 De-multiplexer:

In the 1 to 2 De-multiplexer, there are only two outputs, i.e., Y₀, and Y₁, 1 selection lines, i.e., S₀, and single input, i.e., A. On the basis of the selection value, the input will be connected to one of the outputs. The block diagram and the truth table of the 1×2 multiplexer are given below.

Block Diagram:



Truth Table:

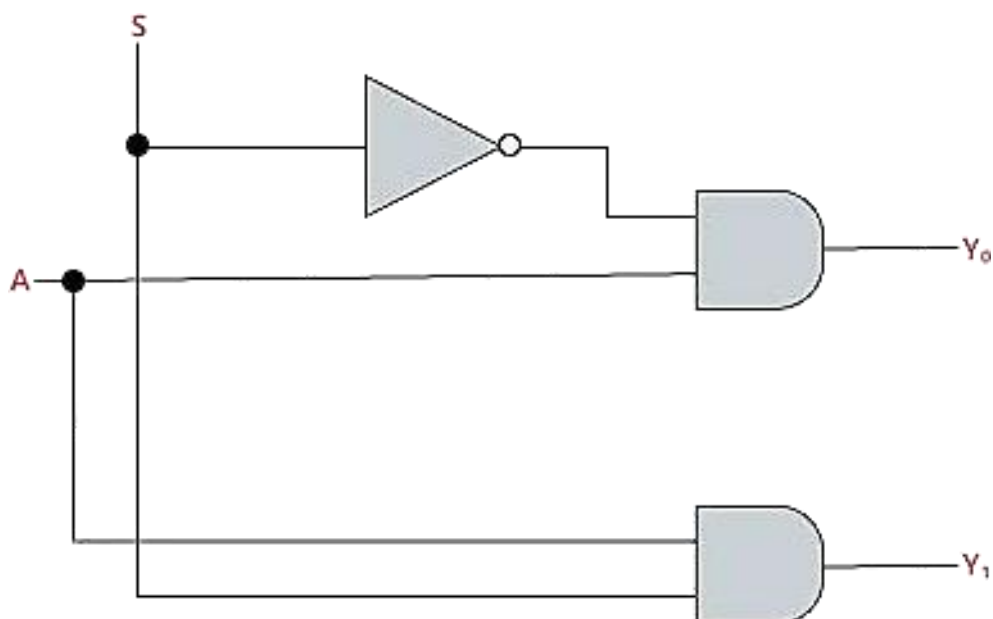
INPUTS		Output	
S ₀		Y ₁	Y ₀
0		0	A
1		A	0

The logical expression of the term Y is as follows:

$$Y_0 = S_0' \cdot A$$

$$Y_1 = S_0 \cdot A$$

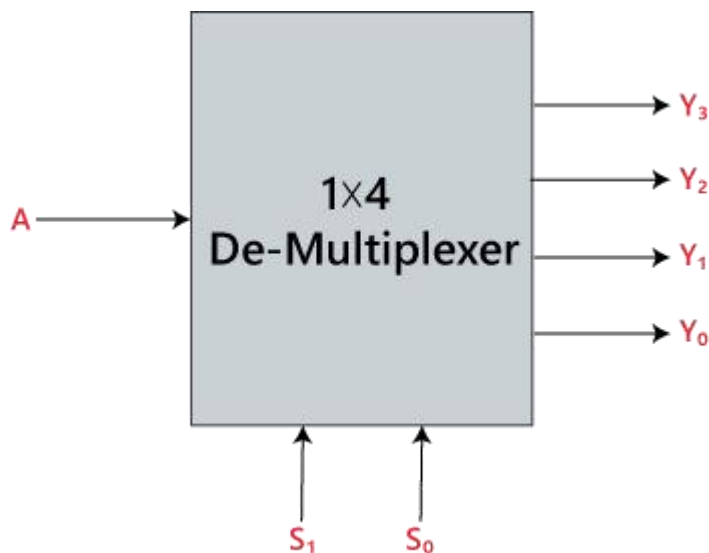
Logical circuit of the above expressions is given below:



1×4 De-multiplexer:

In 1 to 4 De-multiplexer, there are total of four outputs, i.e., Y_0 , Y_1 , Y_2 , and Y_3 , 2 selection lines, i.e., S_0 and S_1 and single input, i.e., A . On the basis of the combination of inputs which are present at the selection lines S_0 and S_1 , the input be connected to one of the outputs. The block diagram and the truth table of the 1×4 multiplexer are given below.

Block Diagram:



Truth Table:

INPUTS		Output			
S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	A
0	1	0	0	A	0
1	0	0	A	0	0
1	1	A	0	0	0

the term Y is as follows:

$$Y_0 = S_1' S_0' A$$

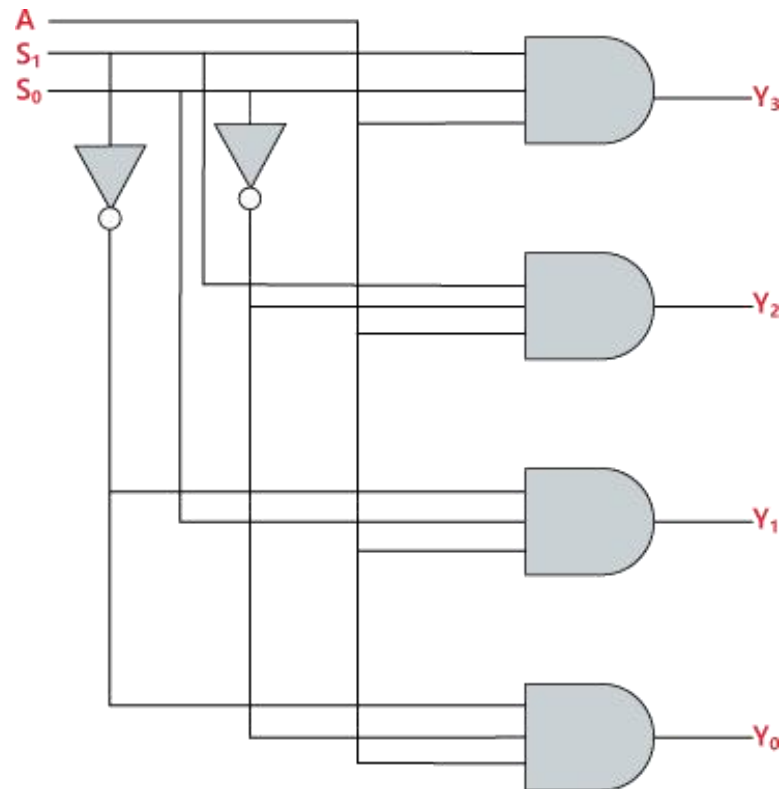
$$Y_1 = S_1' S_0 A$$

$$Y_2 = S_1 S_0' A$$

$$Y_3 = S_1 S_0 A$$

Logical circuit of the above expressions is given below:

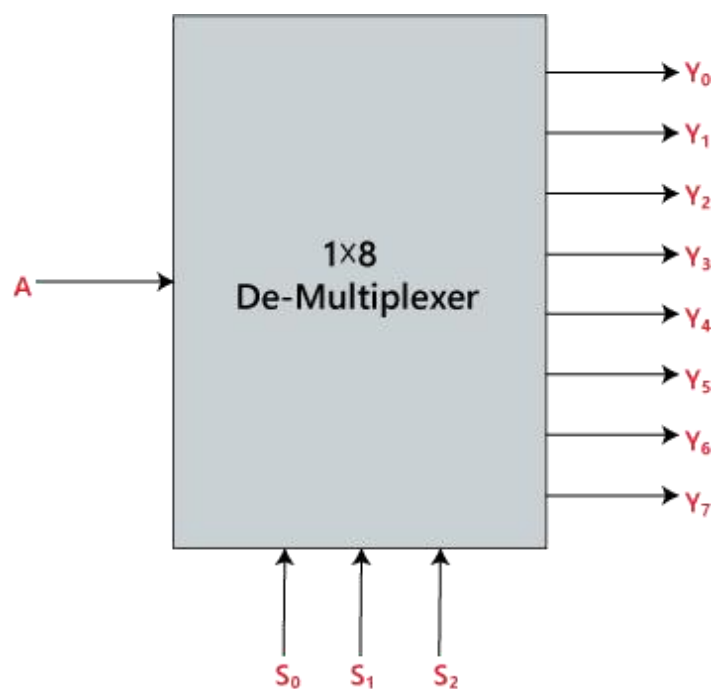
The logical expression of



1×8 De-multiplexer

In 1 to 8 De-multiplexer, there are total of eight outputs, i.e., Y_0 , Y_1 , Y_2 , Y_3 , Y_4 , Y_5 , Y_6 , and Y_7 , 3 selection lines, i.e., S_0 , S_1 and S_2 and single input, i.e., A . On the basis of the combination of inputs which are present at the selection lines S_0 , S_1 and S_2 , the input will be connected to one of these outputs. The block diagram and the truth table of the 1×8 de-multiplexer are given below.

Block Diagram:



Truth Table:

INPUTS			Output							
S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	A
0	0	1	0	0	0	0	0	0	A	0
0	1	0	0	0	0	0	0	A	0	0
0	1	1	0	0	0	0	A	0	0	0
1	0	0	0	0	0	A	0	0	0	0
1	0	1	0	0	A	0	0	0	0	0
1	1	0	0	A	0	0	0	0	0	0
1	1	1	A	0	0	0	0	0	0	0

The logical expression of the term Y is as follows:

$$Y_0 = S_0' \cdot S_1' \cdot S_2' \cdot A$$

$$Y_1 = S_0 \cdot S_1' \cdot S_2' \cdot A$$

$$Y_2 = S_0' \cdot S_1 \cdot S_2' \cdot A$$

$$Y_3 = S_0 \cdot S_1 \cdot S_2' \cdot A$$

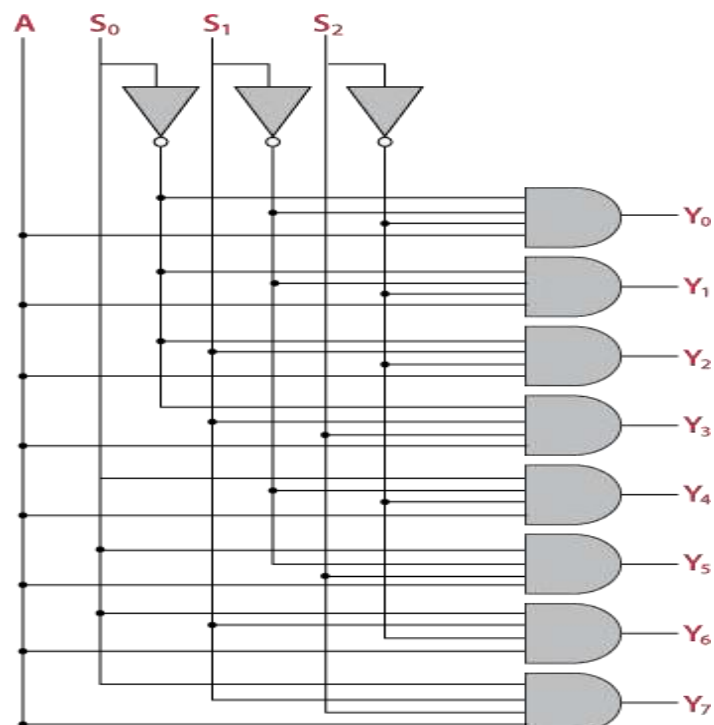
$$Y_4 = S_0' \cdot S_1' \cdot S_2 \cdot A$$

$$Y_5 = S_0 \cdot S_1' \cdot S_2 \cdot A$$

$$Y_6 = S_0' \cdot S_1 \cdot S_2 \cdot A$$

$$Y_7 = S_0 \cdot S_1 \cdot S_2 \cdot A$$

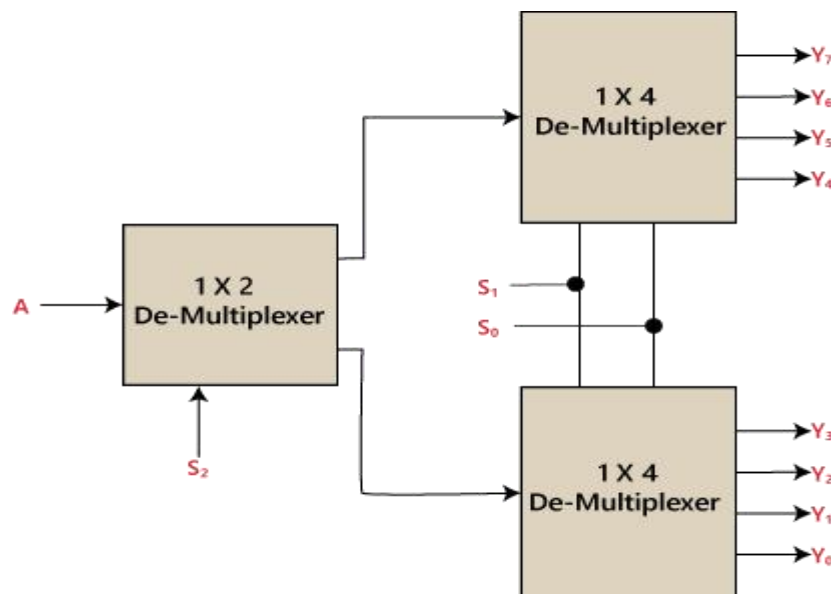
Logical circuit of the above expressions is given below:



1×8 De-multiplexer using 1×4 and 1×2 de-multiplexer

We can implement the 1×8 de-multiplexer using a lower order de-multiplexer. To implement the 1×8 de-multiplexer, we need two 1×4 de-multiplexer and one 1×2 de-multiplexer. The 1×4 multiplexer has 2 selection lines, 4 outputs, and 1 input. The 1×2 de-multiplexer has only 1 selection line.

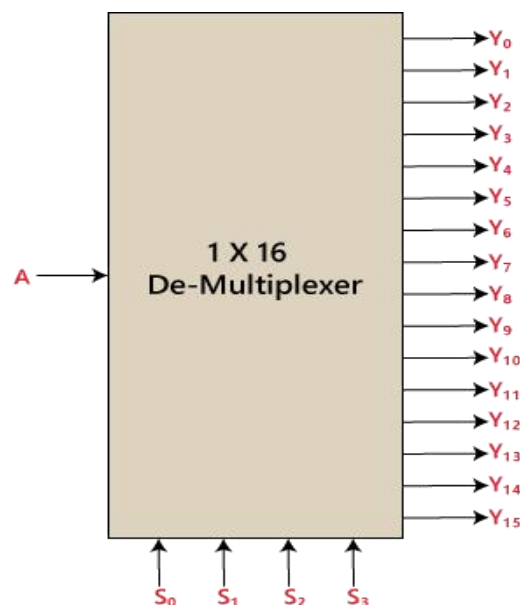
For getting 8 data outputs, we need two 1×4 de-multiplexer. The 1×2 de-multiplexer produces two outputs. So, in order to get the final output, we have to pass the outputs of 1×2 de-multiplexer as an input of both the 1×4 de-multiplexer. The block diagram of 1×8 de-multiplexer using 1×4 and 1×2 de-multiplexer is given below.



1 x 16 De-multiplexer

In 1×16 de-multiplexer, there are total of 16 outputs, i.e., $Y_0, Y_1, \dots, Y_{15}$, 4 selection lines, i.e., S_0, S_1, S_2 , and S_3 and single input, i.e., A . On the basis of the combination of inputs which are present at the selection lines S^0, S^1 , and S_2 , the input will be connected to one of these outputs. The block diagram and the truth table of the 1×16 de-multiplexer are given below.

Block Diagram:



Truth Table:

INPUTS				OUTPUTS															
S_3	S_2	S_1	S_0	Y_{15}	Y_{14}	Y_{13}	Y_{12}	Y_{11}	Y_{10}	Y_9	Y_8	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The logical expression of the term Y is as follows:

$$Y_0 = A.S_0'.S_1'.S_2'.S_3'$$

$$Y_1 = A.S_0'.S_1'.S_2'.S_3$$

$$Y_2 = A.S_0'.S_1'.S_2.S_3'$$

$$Y_3 = A.S_0'.S_1'.S_2.S_3$$

$$Y_4 = A.S_0'.S_1.S_2'.S_3'$$

$$Y_5 = A.S_0'.S_1.S_2'.S_3$$

$$Y_6 = A.S_0'.S_1.S_2.S_3'$$

$$Y_7 = A.S_0'.S_1.S_2.S_3$$

$$Y_8 = A.S_0.S_1'.S_2'.S_3'$$

$$Y_9 = A.S_0.S_1'.S_2'.S_3$$

$$Y_{10} = A.S_0.S_1'.S_2.S_3'$$

$$Y_{11} = A.S_0.S_1'.S_2.S_3$$

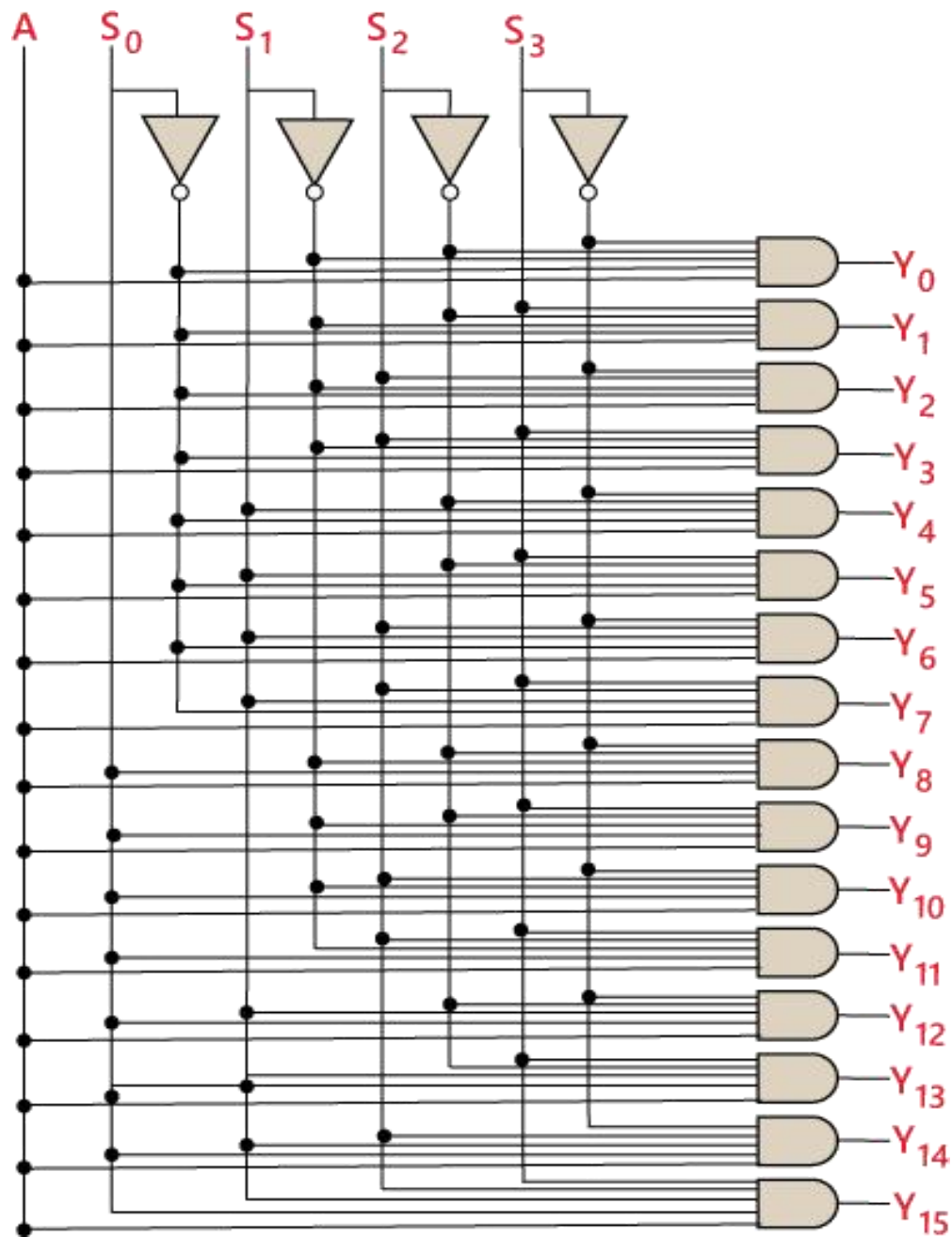
$$Y_{12} = A.S_0.S_1.S_2'.S_3'$$

$$Y_{13} = A.S_0.S_1.S_2'.S_3$$

$$Y_{14} = A \cdot S_0 \cdot S_1 \cdot S_2 \cdot S_3'$$

$$Y_{15} = A \cdot S_0 \cdot S_1 \cdot S_2' \cdot S_3$$

Logical circuit of the above expressions is given below:

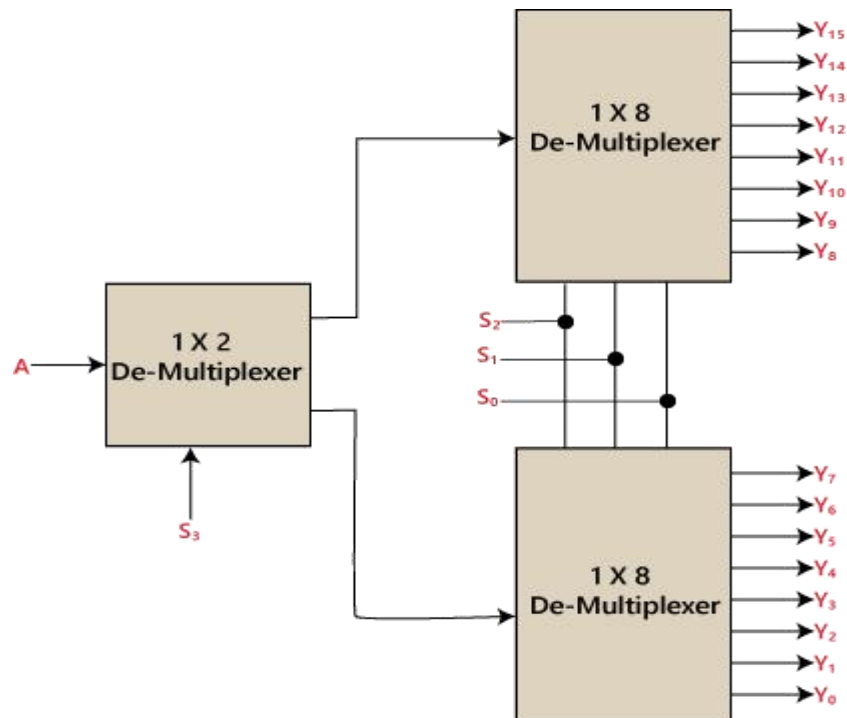


1×16 de-multiplexer using 1×8 and 1×2 de-multiplexer

We can implement the 1×16 de-multiplexer using a lower order de-multiplexer. To implement the 1×16 de-multiplexer, we need two 1×8 de-multiplexer and one 1×2 de-multiplexer. The 1×8 multiplexer has 3 selection lines, 1 input, and 8 outputs. The 1×2 de-multiplexer has only 1 selection line.

For getting 16 data outputs, we need two 1×8 de-multiplexer. The 1×8 de-multiplexer produces eight outputs. So, in order to get the final output, we need a 1×2 de-multiplexer to

produce two outputs from a single input. Then we pass these outputs into both the de-multiplexer as an input. The block diagram of 1×16 de-multiplexer using 1×8 and 1×2 de-multiplexer is given below.



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 shown in Fig. 4.29. It is distinguished from a normal buffer by an input control line entering the bottom of the symbol. 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.

- Fig.3.12 shows a three-state buffer and its logical equivalent. Three-state buffers are also called tri-state buffers.

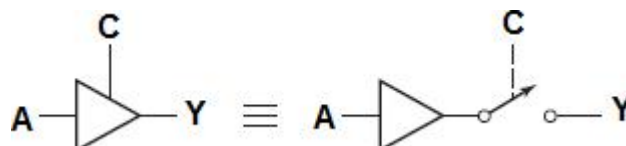


Fig. 3.12: Three-State Buffer

- When the enable input C is 1, the output Y equals A; when C is 0, the output Y acts like an open circuit.
- When C is 0, the output Y is effectively disconnected from the buffer output so that no current can flow.
- This is often referred to as a Hi-Z (high-impedance) state of the output because the circuit offers a very high resistance or impedance to the flow of current.
- Fig.3.13 shows four different types of three-state buffers. In fig.3.13(a) and (b), the enable input C is not inverted, so the buffer output is enabled when C = 1 and disabled when C = 0.
- In fig.3.13 (b), the buffer output is inverted so that output Y = $\bar{A}$ when the buffer is enabled.
- The buffers in fig.3.13(c) and (d) operate the same as in (a) and (b) except that the enable input is inverted, so the buffer is enabled when C = 0.

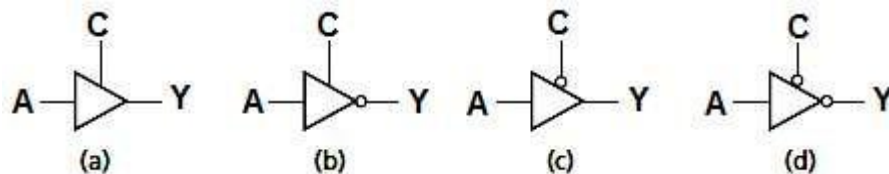


Fig.3.13: Four Kinds of Three-State Buffers

C	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

C	A	Y
0	0	Z
0	1	Z
1	0	1
1	1	0

C	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

C	A	Y
0	0	1
0	1	0
1	0	Z
1	1	Z

- Two three-state buffers can be used to implement 2-to-1 multiplexer as shown in the fig.3.14.
- When B = 0, the top buffer is enabled, so that D = A. When B = 1, the lower buffer is enabled, so that D = C.
- Therefore,

$$D = BA + BC$$

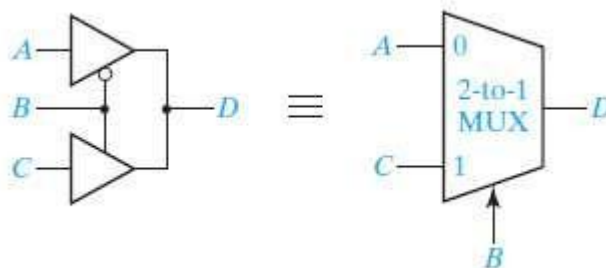


Fig.3.14: Data Selection Using Three-State Buffers

- When two three-state buffer outputs are connected together, as shown in fig.3.15, if one of the buffers is disabled (output = Z), the combined output F is the same as the other buffer output. If both buffers are disabled, the output is Z. If both buffers are enabled, a conflict can occur.

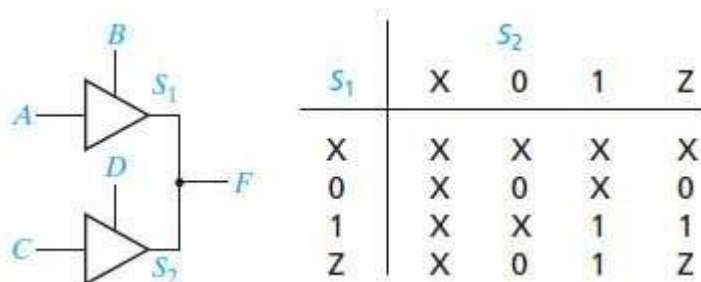


Fig.3.15: Circuit with Two Three-State Buffers

- If $A = 0$ and $C = 1$, then F output is unknown (X). If one of the buffer inputs is unknown, the F output will also be unknown.
- 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 . The construction of a four-to-one-line multiplexer is shown in Fig. 4.30(b). The outputs of 4 three-state buffers are connected together to form a single output line. The control inputs to the buffers determine which one of the four normal inputs I_0 through I_3 will be connected to the output line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only 1 three state buffer has access to the output while all other buffers are maintained in a high impedance state. 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. Careful investigation reveals that this circuit is another way of constructing a four-to-one-line multiplexer.

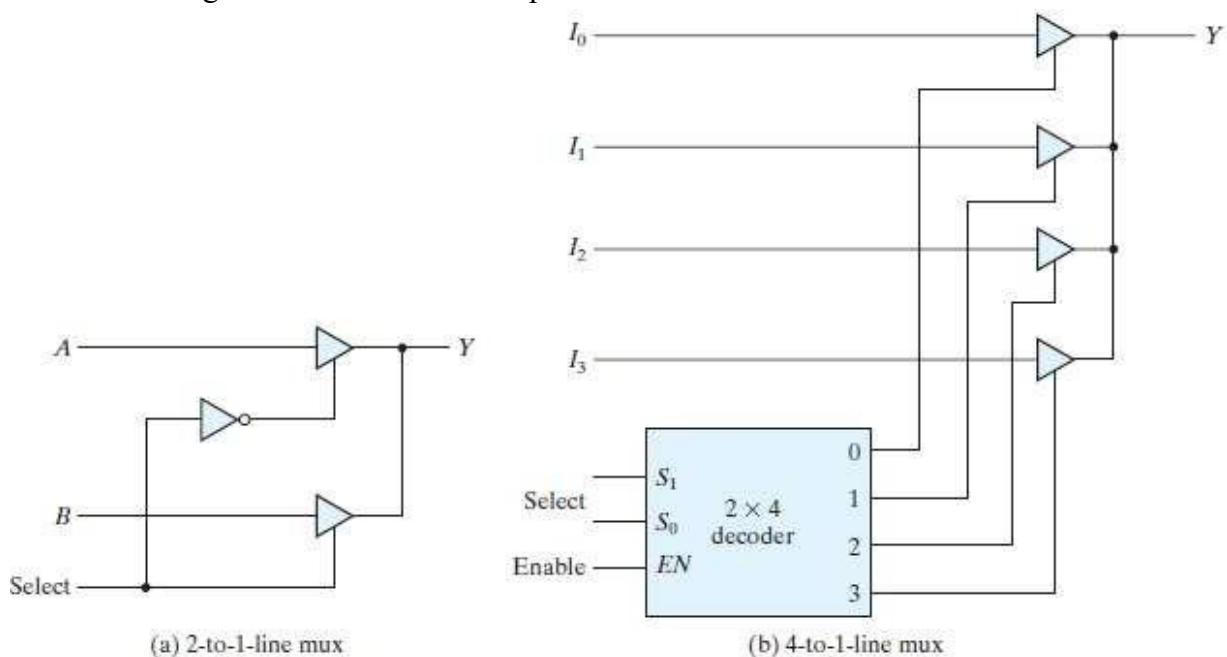


FIGURE 4.30
Multiplexers with three-state gates

4.12 HDL MODELS OF COMBINATIONAL CIRCUITS

As mentioned previously, the module is the basic building block for modeling hardware with the Verilog HDL. The logic of a module can be described in any one (or a combination) of the following modeling styles:

- **Gate-level modeling** using instantiations of predefined and user-defined primitive gates.
 - Gate-level (structural) modeling describes a circuit by specifying its gates and how they are connected with each other.

- **Dataflow modeling** using continuous assignment statements with the keyword `assign`.
 - Dataflow modeling is used mostly for describing the Boolean equations of combinational logic.
- **Behavioral modeling** using procedural assignment statements with the keyword `always`.
 - behavioral modeling that is used to describe combinational and sequential circuits at a higher level of abstraction.

Combinational logic can be designed with truth tables, Boolean equations, and schematics; Verilog has a construct corresponding to each of these “classical” approaches to design: user-defined primitives, continuous assignments, and primitives, as shown in Fig. 4.31. There is one other modeling style, called switch-level modeling. It is sometimes used in the simulation of MOS transistor circuit models, but not in logic synthesis. We will not consider switch-level modeling.

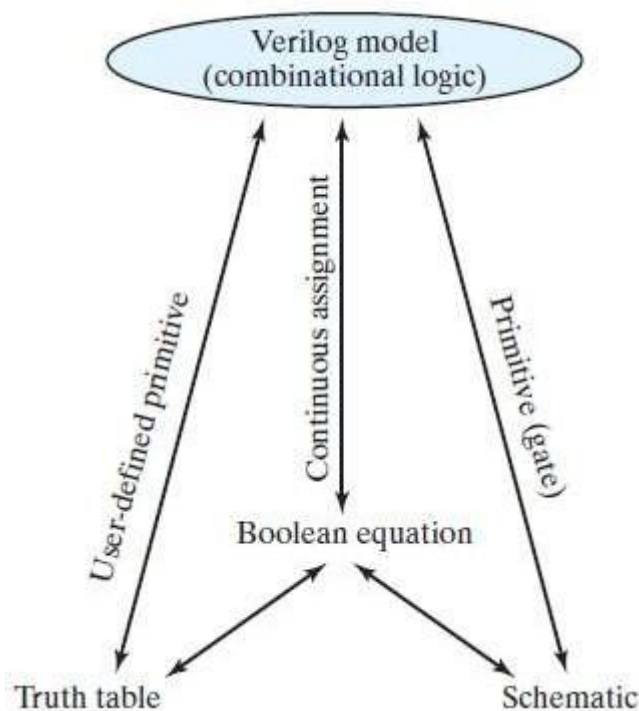


FIGURE 4.31: Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

Gate-Level Modeling

Gate-level modeling was introduced in Section 3.10 with a simple example. In this type of representation, a circuit is specified by its logic gates and their interconnections. Gate level modeling provides a textual description of a schematic diagram. The Verilog HDL includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type. The other eight are all declared with the lowercase keywords **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, and **buf**. Primitives such as **and** are n-input primitives. They can have any number of scalar inputs (e.g., a three-input and primitive). The **buf** and **not** primitives are n-output primitives. A single input can drive multiple output lines distinguished by their identifiers.

The Verilog language includes a functional description of each type of gate, too. The logic of each gate is based on a **four-valued system**. When the gates are simulated, the simulator assigns one value to the output of each gate at any instant. In addition to the **two logic values of 0 and 1**, there are two other values: **unknown** and **high impedance**.

- An unknown value is denoted by `x` and a high impedance by `z`.

- An unknown value is assigned during simulation when the logic value of a signal is ambiguous—for instance, if it cannot be determined whether its value is 0 or 1 (e.g., a flip-flop without a reset condition).
- A high-impedance condition occurs at the output of three-state gates that are not enabled or if a wire is inadvertently left unconnected.
- The four-valued logic truth tables for the **and**, **or**, **xor**, and **not** primitives are shown in Table 4.9. The truth table for the other four gates is the same, except that the outputs are complemented. Note that for the **and** gate, the output is 1 only when both inputs are 1 and the output is 0 if any input is 0. Otherwise, if one input is x or z, the output is x. The output of the **or** gate is 0 if both inputs are 0, is 1 if any input is 1, and is x otherwise.

Table 4.9
Truth Table for Predefined Primitive Gates

and	0	1	x	z	or	0	1	x	z
0	0	0	0	0	0	0	1	x	x
1	0	1	x	x	1	1	1	1	1
x	0	x	x	x	x	x	1	x	x
z	0	x	x	x	z	x	1	x	x

xor	0	1	x	z	not	input	output
0	0	1	x	x		0	1
1	1	0	x	x		1	0
x	x	x	x	x		x	x
z	x	x	x	x		z	x

Instantiate: When a primitive gate is listed in a module, we say that it is instantiated in the module. In general, component instantiations are statements that reference lower level components in the design, essentially creating unique copies (or instances) of those components in the higher level module. Thus, a module that uses a gate in its description is said to *instantiate* the gate. Think of instantiation as the HDL counterpart of placing and connecting parts on a circuit board.

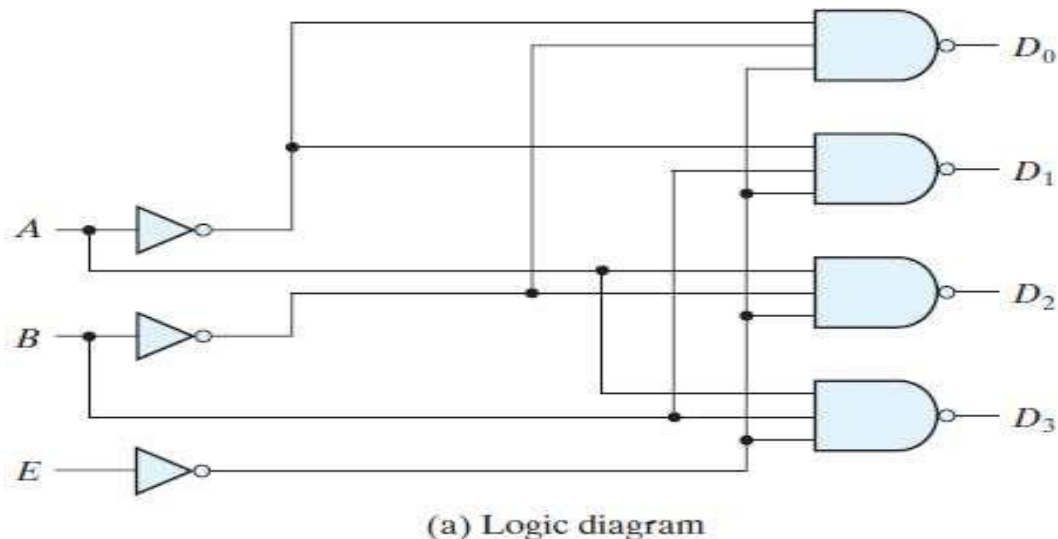
Vectors: We now present two examples of gate-level modeling. Both examples use identifiers having multiple bit widths, called vectors. The syntax specifying a vector includes within square brackets two numbers separated with a colon. The following Verilog statements specify two vectors:

```
output [0: 3] D;
wire [7: 0] SUM;
```

The first statement declares an output vector D with four bits, 0 through 3. The second declares a wire vector SUM with eight bits numbered 7 through 0. (Note: The first (leftmost) number (array index) listed is always the most significant bit of the vector.) The individual bits are specified within square brackets, so D[2] specifies bit 2 of D. It is also possible to address parts (contiguous bits) of vectors. For example, SUM[2: 0] specifies the three least significant bits of vector SUM.

Wire: The wire declaration is for internal connections

Example 4.1: HDL shows the gate-level description of a two-to-four-line decoder.



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
```

Basic types of design methodologies: Two or more modules can be combined to build a hierarchical description of a design. There are two basic types of design methodologies: top down and bottom up. In a top-down design, the top-level block is defined and then the sub blocks necessary to build the top-level block are identified. In a bottom-up design, the building blocks are first identified and then combined to build the top-level block. Take, for example, the binary adder of Fig. 4.9. It can be considered as a top-block component built with four full-adder blocks, while each full adder is built with two half-adder blocks. In a top-down design, the four-bit adder is defined first, and then the two adders are described. In a bottom-up design, the half adder is defined, then each full adder is constructed, and then the four-bit adder is built from the full adders.

A bottom-up hierarchical description of a four-bit adder is shown in HDL Example 4.2. The half adder is defined by instantiating primitive gates. The next module describes the full adder by instantiating and connecting two half adders. The third module describes the four-bit adder by instantiating and connecting four full adders.

Note that the first character of an identifier cannot be a number, but can be an underscore, so the module name 4bitadder is valid. An alternative name that is meaningful, but does not require a leading

underscore, is `adder_4_bit`. The instantiation is done by using the name of the module that is instantiated together with a new (or the same) set of port names. For example, the half adder HA1 inside the full adder module is instantiated with ports S1, C1, x, and y. This produces a half adder with outputs S1 and C1 and inputs x and y.

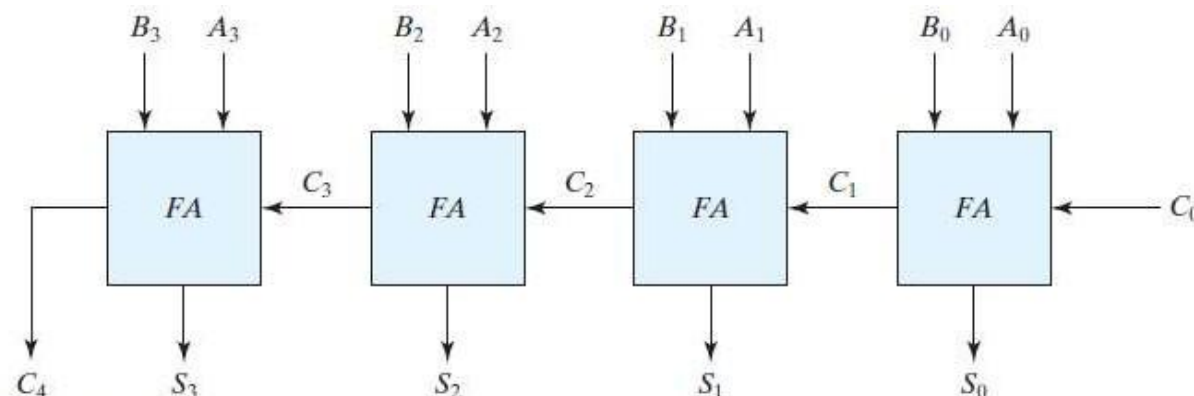


FIGURE 4.9
Four-bit adder

HDL Example 4.2 (Ripple-Carry Adder)

```
// Gate-level description of four-bit ripple carry adder
// Description of half adder ( Fig. 4.5b )
// module half_adder (S, C, x, y); // Verilog 1995 syntax
// output S, C;
// input x, y;

module half_adder ( output S, C, input x, y); // Verilog 2001, 2005 syntax
// Instantiate primitive gates
    xor (S, x, y);
    and (C, x, y);
endmodule

// Description of full adder ( Fig. 4.8 ) // Verilog 1995 syntax
// module full_adder (S, C, x, y, z);
// output S, C;
// input x, y, z;

module full_adder ( output S, C, input x, y, z); // Verilog 2001, 2005 syntax
    wire S1, C1, C2;
    // Instantiate half adders
    half_adder HA1 (S1, C1, x, y);
    half_adder HA2 (S, C2, S1, z);
    or G1 (C, C2, C1);
endmodule

// Description of four-bit adder ( Fig. 4.9 ) // Verilog 1995 syntax
// module ripple_carry_4_bit_adder (Sum, C4, A, B, C0);
// output [3: 0] Sum;
// output C4;
// input [3: 0] A, B;
// input C0;
// Alternative Verilog 2001, 2005 syntax:
```

```

module ripple_carry_4_bit_adder ( output [3: 0] Sum, output C4,
    input [3: 0] A, B, input C0);
    wire C1, C2, C3; // Intermediate carries
    // Instantiate chain of full adders
    full_adder FA0 (Sum[0], C1, A[0], B[0], C0),
    FA1 (Sum[1], C2, A[1], B[1], C1),
    FA2 (Sum[2], C3, A[2], B[2], C2),
    FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule

```

Note that modules can be instantiated (nested) within other modules, but module declarations cannot be nested; that is, a module definition (declaration) cannot be placed within another module declaration. In other words, a module definition cannot be inserted into the text between the module and endmodule keywords of another module. The only way one module definition can be incorporated into another module is by instantiating it. Instantiating modules within other modules creates a hierarchical decomposition of a design. A description of a module is said to be a structural description if it is composed of instantiations of other modules. Note also that instance names must be specified when defined modules are instantiated (such as FA0 for the first full adder in the third module), but using a name is optional when instantiating primitive gates. Module ripple_carry_4_bit_adder is composed of instantiated and interconnected full adders, each of which is itself composed of half adders and some glue logic. The top level, or parent module, of the design hierarchy is the module ripple_carry_4_bit_adder. Four copies of full_adder are its child modules, etc. C0 is an input of the cell forming the least significant bit of the chain, and C4 is the output of the cell forming the most significant bit.

Three-State Gates: A three-state gate has a control input that can place the gate into a high-impedance state. The high-impedance state is symbolized by *z* in Verilog. There are four types of three-state gates, as shown in Fig. 4.32. The **bufif1** gate behaves like a normal buffer if control = 1. The output goes to a high-impedance state *z* when control = 0. The **bufif0** gate behaves in a similar fashion, except that the high-impedance state occurs when control = 1. The two **notif** gates operate in a similar manner, except that the output is the complement of the input when the gate is not in a high-impedance state.

The gates are instantiated with the statement

gatename (output, input, control);

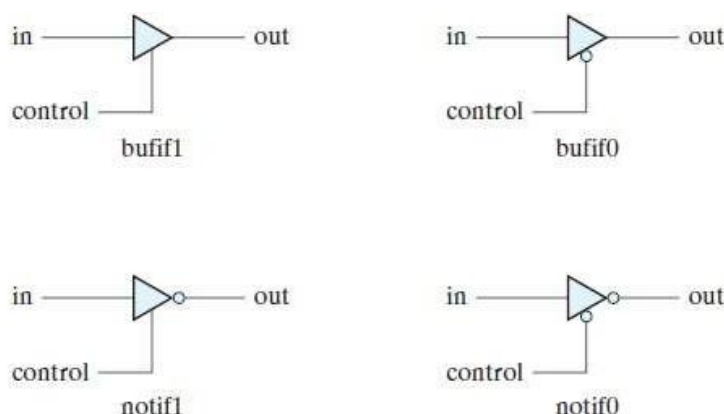


FIGURE 4.32
Three-state gates

The gate name can be that of any 1 of the 4 three-state gates. In simulation, the output can result in 0, 1, *x*, or *z*. Two examples of gate instantiation are

bufif1 (OUT, A, control);

notif0 (Y, B, enable);

In the first example, input *A* is transferred to *OUT* when *control* = 1.

OUT goes to **z** when *control* = 0.

In the second example, output *Y* = **z** when *enable* = 1 and

Output *Y* = *B* when *enable* = 0.

The HDL description must use a tri data type for the output:

The outputs of three-state gates can be connected together to form a common output line. To identify such a connection, Verilog HDL uses the keyword **tri** (for tristate) to indicate that the output has multiple drivers. As an example, consider the two-to-one-line multiplexer with three-state gates shown in Fig. 4.33.

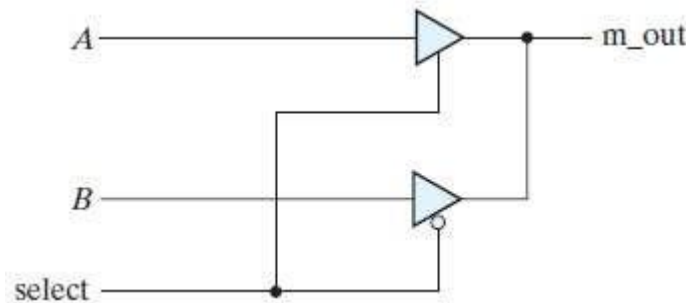


FIGURE 4.33: Two-to-one-line multiplexer with three-state buffers

The HDL description must use a **tri** data type for the output:

```
// Mux with three-state output
module mux_tri (m_out, A, B, select);
    output m_out;
    input A, B, select;
    tri m_out;
    bufif1 (m_out, A, select);
    bufif0 (m_out, B, select);
endmodule
```

The 2 three-state buffers have the same output. In order to show that they have a common connection, it is necessary to declare **m_out** with the keyword **tri**.

Nets: Keywords **wire** and **tri** are examples of a set of data types *called nets*, which represent connections between hardware elements.

- In simulation, their value is determined by a continuous assignment statement or by the device whose output they represent.
- The word **net** is not a keyword, but represents a class of data types, such as **wire**, **wor**, **wand**, **tri**, **supply1**, and **supply0**.
- The **wire** declaration is used most frequently. In fact, if an identifier is used, but not declared, the language specifies that it will be interpreted (by default) as a wire.
- The net **wor** models the hardware implementation of the wired-OR configuration (emitter-coupled logic).
- The **wand** models the wired-AND configuration (open-collector technology; see Fig. 3.26).
- The nets **supply1** and **supply0** represent power supply and ground, respectively. They are used to hardwire an input of a device to either 1 or 0.

Dataflow Modeling:

Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result. Verilog HDL provides about 30 different operators.

Table 4.10 lists some of these operators, their symbols, and the operation that they perform.

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		

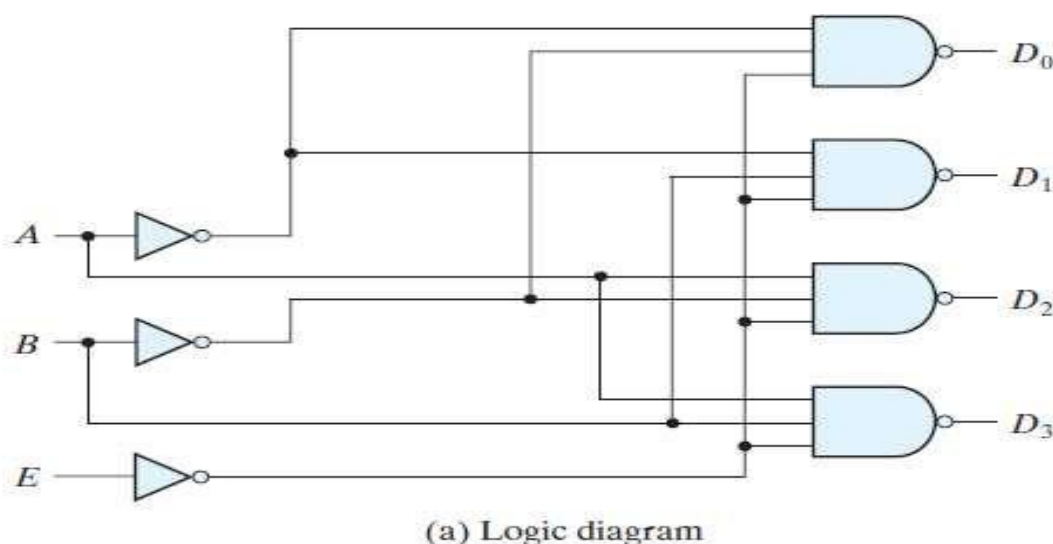
Dataflow modeling uses continuous assignments and the keyword `assign`. A continuous assignment is a statement that assigns a value to a net. The data type family `net` is used in Verilog HDL to represent a physical connection between circuit elements. A net is declared explicitly by a net keyword (e.g., `wire`) or by declaring an identifier to be an input port. The logic value associated with a net is determined by what the net is connected to. If the net is connected to an output of a gate, the net is said to be driven by the gate, and the logic value of the net is determined by the logic values of the inputs to the gate and the truth table of the gate. If the identifier of a net is the left-hand side of a continuous assignment statement or a procedural assignment statement, the value assigned to the net is specified by a Boolean expression that uses operands and operators.

As an example, assuming that the variables were declared, a two-to-one-line multiplexer with scalar data inputs `A` and `B`, select input `S`, and output `Y` is described with the continuous assignment

`assign Y = (A && S) || (B && S);`

The relationship between `Y`, `A`, `B`, and `S` is declared by the keyword `assign`, followed by the target output `Y` and an equals sign. Following the equals sign is a Boolean expression. In hardware terms, this assignment would be equivalent to connecting the output of the OR gate to wire `Y`.

Example 4.3: HDL shows the dataflow description of a two-to-four-line decoder.



HDL Example 4.3 (Dataflow: Two-to-Four Line Decoder)

// Dataflow description of two-to-four-line decoder

// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

```

module decoder_2x4_df (                                // Verilog 2001, 2005 syntax
    output [0: 3] D,
    input  A, B,
           enable
);
    assign D[0] = !((!A) && (!B) && (!enable)),
           D[1] = !(*!A) && B && (!enable)),
           D[2] = !(A && B && (!enable))
           D[3] = !(A && B && (!enable))
endmodule

```

HDL Example 4.5 (Dataflow: Four-Bit Comparator)

The algorithm is a direct application of the procedure a person uses to compare the relative magnitudes of two numbers. 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 outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$, or $A < B$.

$$(A = B) = x_3 x_2 x_1 x_0$$

$$(A > B) = A_3 B'_3 + x_3 A_2 B'_2 + x_3 x_2 A_1 B'_1 + x_3 x_2 x_1 A_0 B'_0$$

$$(A < B) = A'_3 B_3 + x_3 A'_2 B_2 + x_3 x_2 A'_1 B_1 + x_3 x_2 x_1 A'_0 B_0$$

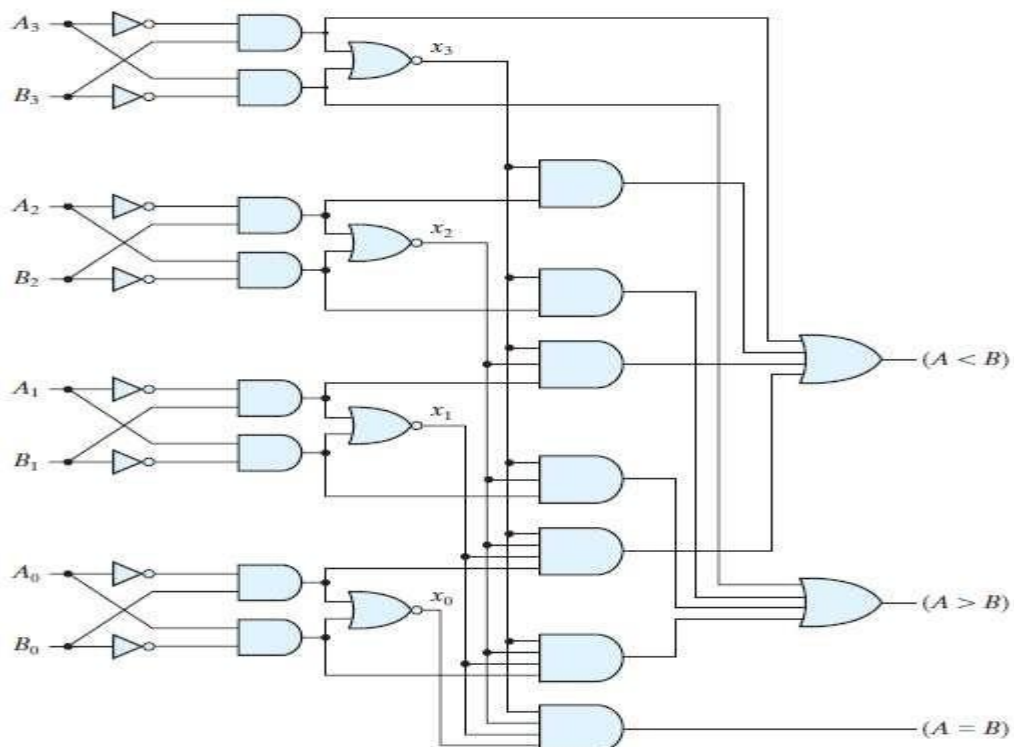


FIGURE 4.17
Four-bit magnitude comparator

```
// Dataflow description of a four-bit comparator //V2001, 2005 syntax

module mag_compare
( output          A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]     A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A == B);
endmodule
```

Conditional operator (? :) This operator takes three operands:

condition ? true-expression: false-expression;

The condition is evaluated.

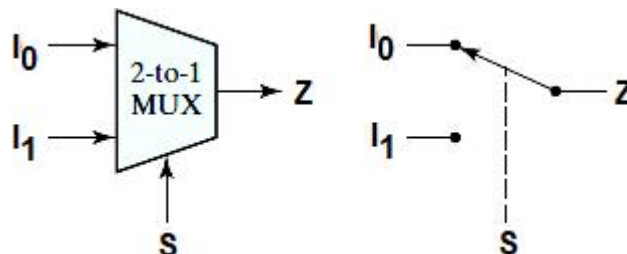
- If the result is logic 1, the true expression is evaluated and used to assign a value to the left-hand side of an assignment statement.
- If the result is logic 0, the false expression is evaluated. The two conditions together are equivalent to an if-else condition.

HDL Example 4.6 describes a two-to-one-line multiplexer using the conditional operator.

The continuous assignment

assign OUT = select ? A : B;

specifies the condition that OUT = A if select = 1, else OUT = B if select = 0.



HDL Example 4.6 (Dataflow: Two-to-One Multiplexer)

```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output      m_out;
  input       A, B;
  input       select;

  assign m_out = (select)? A : B;
endmodule
```

Behavioral Modeling:

- Behavioral modeling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.
- Behavioral descriptions use the keyword **always**, followed by an **optional event control expression** and a **list of procedural assignment statements**.
- The **event control expression** specifies when the statements will execute.
- The target output of a procedural assignment statement must be of the **reg** data type. Contrary to the wire data type, whereby the target output of an assignment may be continuously updated, a **reg** data type retains its value until a new value is assigned.

HDL Example 4.7 shows the behavioral description of a two-to-one-line multiplexer. (Compare it with HDL Example 4.6.)

- Since variable `m_out` is a target output, it must be declared as **reg** data (in addition to the output declaration).
- The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variables listed after the `@` symbol.
- Note that there is no semicolon (;) at the end of the **always** statement.
- In this case, these variables are the input variables `A`, `B`, and `select`. The statements execute if `A`, `B`, or `select` changes value.
- Note that the keyword `or`, instead of the bitwise logical OR operator "`|`", is used between variables.

The conditional statement if–else provides a decision based upon the value of the select input.

The

if statement can be written without the equality symbol:

if (select) `OUT = A`;

The statement implies that *select* is checked for logic 1.

HDL Example 4.7 (Behavioral: Two-to-One Line Multiplexer)

```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);
    output      m_out;
    input       A, B, select;
    reg         m_out;

    always      @(A or B or select)
        if (select == 1) m_out = A;
        else m_out = B;
endmodule
```

case statement:

HDL Example 4.8 describes the function of a four-to-one-line multiplexer.

- The select input is defined as a **two-bit vector**, and output `y` is declared to have type **reg**.
- The **always** statement, in this example, has a sequential block enclosed between the keywords **case** and **endcase**.
- The block is executed whenever any of the inputs listed after the `@` symbol changes in value.

- Whenever in_0, in_1, in_2, in_3 or select change, the case expression (select) is evaluated and its value compared, from top to bottom, with the values in the list of statements that follow, the so-called **case** items.
- The statement associated with the first **case** item that matches the case expression is executed. In the absence of a match, **no statement is executed**. Since select is a two-bit number, it can be equal to 00, 01, 10, or 11. The case items have an implied priority because the list is evaluated from top to bottom. The list is called a **sensitivity list** (Verilog 2001, 2005) and is equivalent to the **event control expression** (Verilog 1995) formed by “ORing” the signals. Combinational logic is reactive—when an input changes an output may change.

HDL Example 4.8 (Behavioral: Four-to-One Line Multiplexer)

```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg m_out,
  input      in_0, in_1, in_2, in_3,
  input [1: 0] select
);
always @ (in_0, in_1, in_2, in_3, select)    // Verilog 2001, 2005 syntax
  case (select)
    2'b00:      m_out = in_0;
    2'b01:      m_out = in_1;
    2'b10:      m_out = in_2;
    2'b11:      m_out = in_3;
  endcase
endmodule
```

- Binary numbers in Verilog are specified and interpreted with the letter **b** preceded by a prime. The size of the number is written first and then its value.
 - Thus, **2'b01** specifies a two-bit binary number whose value is 01.
- Numbers are stored as a bit pattern in memory, but they can be referenced in decimal, octal, or hexadecimal formats with the letters **d**', **o**', and **h**', respectively.
 - For example, **4'HA = 4'd10 = 4'b1010** and have the same internal representation in a simulator.
 - If the base of the number is not specified, its interpretation defaults to decimal. If the size of the number is not specified, the system assumes that the size of the number **is at least 32 bits**; if a host simulator has a larger word length—say, 64 bits—the language will use that value to store unsized numbers.
 - The integer data type (keyword integer) is stored in a 32-bit representation.
 - The underscore (**_**) may be inserted in a number to improve readability of the code (e.g., 16'b0101_1110_0101_0011). It has no other effect.
- The case construct has two important variations: **casex** and **casez** .

- The first will treat as don't-cares any bits of the case expression or the case item that have logic value x or z. The casez construct treats as don't-cares only the logic value z, for the purpose of detecting a match between the case expression and a case item.
- The list of case items need not be complete. If the list of case items does not include all possible bit patterns of the case expression, no match can be detected. Unlisted case items, i.e., bit patterns that are not explicitly decoded can be treated by using the default keyword as the last item in the list of case items. The associated statement will execute when no other match is found. This feature is useful, for example, when there are more possible state codes in a sequential machine than are actually used. Having a default case item lets the designer map all of the unused states to a desired next state without having to elaborate each individual state, rather than allowing the synthesis tool to arbitrarily assign the next state.

Writing a Simple Test Bench

A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit in order to test it and observe its response during simulation. Test benches can be quite complex and lengthy and may take longer to develop than the design that is tested. The results of a test are only as good as the test bench that is used to test a circuit. Care must be taken to write stimuli that will test a circuit thoroughly, exercising all of the operating features that are specified. However, the test benches considered here are relatively simple, since the circuits we want to test implement only combinational logic.

In addition to employing the **always** statement, test benches use the **initial** statement to provide a stimulus to the circuit being tested. We use the term “always statement” loosely. Actually, always is a Verilog language construct specifying how the associated statement is to execute (subject to the event control expression). The always statement executes repeatedly in a loop. The **initial** statement executes only once, starting from simulation time 0, and may continue with any operations that are delayed by a given number of time units, as specified by the symbol #.

For example, consider the **initial** block

```
initial
    begin
        A=0; B=0;
        #10 A=1;
        #20 A=0; B=1;
    end
```

The block is enclosed between the keywords **begin** and **end**. At time 0, *A* and *B* are set to 0. Ten time units later, *A* is changed to 1. Twenty time units after that (at $t=30$), *A* is changed to 0 and *B* to 1.

For example, Inputs specified by a three-bit truth table can be generated with the **initial** block:

```
initial
    begin
        D=3'b000;
        repeat (7)
            #10 D=D + 3'b001;
    end
```

When the simulator runs, the three-bit vector *D* is initialized to 000 at time = 0. The keyword **repeat** specifies a looping statement: *D* is incremented by 1 seven times, once every 10 time units. The result is a sequence of binary numbers from 000 to 111.

A stimulus module has the following form:

```

module test_module_name;
// Declare local reg and wire identifiers.
// Instantiate the design module under test.
// Specify a stopwatch, using $finish to terminate the simulation.
// Generate stimulus, using initial and always statements.
// Display the output response (text or graphics (or both)).
endmodule

```

A test module is written like any other module, but it typically has no inputs or outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local reg data type. The outputs of the design module that are displayed for testing are declared in the stimulus module as local wire data type. The module under test is then instantiated, using the local identifiers in its port list.

Figure 4.34 clarifies this relationship. The stimulus module generates inputs for the design module by declaring local identifiers t_A and t_B as **reg** type and checks the output of the design unit with the **wire** identifier t_C . The local identifiers are then used to instantiate the design module being tested. The simulator associates the (actual) local identifiers within the test bench, t_A , t_B , and t_C , with the formal identifiers of the module (A, B, C). The association shown here is based on position in the port list, which is adequate for the examples that we will consider. The reader should note, however, that Verilog provides a more flexible name association mechanism for connecting ports in larger circuits.

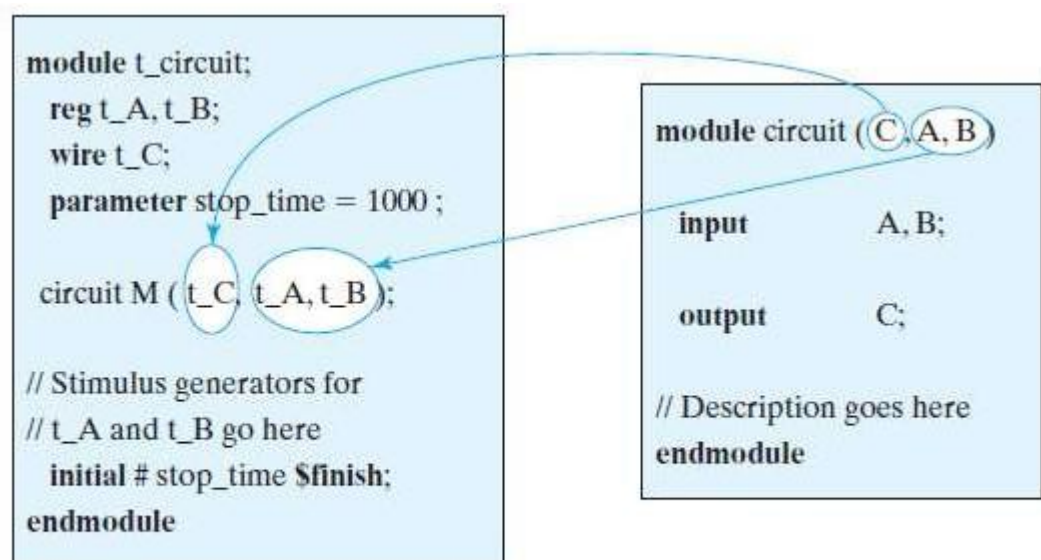


FIGURE 4.34
Interaction between stimulus and design modules

The response to the stimulus generated by the **initial** and **always** blocks will appear in text format as standard output and as waveforms (timing diagrams) in simulators having graphical output capability. Numerical outputs are displayed by using Verilog system tasks. These are built-in system functions that are recognized by keywords that begin with the symbol \$. Some of the system tasks that are useful for display are

- \$display** —display a one-time value of variables or strings with an end-of-line return,
- \$write** —same as \$display, but without going to next line,
- \$monitor** —display variables whenever a value changes during a simulation run,
- \$time** —display the simulation time,

\$finish —terminate the simulation.

The syntax for **\$display**, **\$write**, and **\$monitor** is of the form

Task-name (format specification, argumentlist);

The format specification uses the symbol % to specify the radix of the numbers that are displayed and may have a string enclosed in quotes ("). The base may be binary, decimal, hexadecimal, or octal, identified with the symbols %b, %d, %h, and %o, respectively (%B, %D, %H, and %O are valid too).

For example, the statement

\$display ("%d %b %b", C, A, B);

are separated by a comma, and that the argument list has commas between the variables. An example that specifies a string enclosed in quotes may look like the statement

\$display ("time= %0d A= %b", \$time, A, B);

and will produce the display time=3 A=10 B=1

where (*time* =), (*A* =), and (*B* =) are part of the string to be displayed. The format specifiers %0d, %b, and %b specify the base for **\$time**, *A*, and *B*, respectively. In displaying time values, it is better to use the format %0d instead of %d. This provides a display of the significant digits without the leading spaces that %d will include. (%d will display about 10 leading spaces because time is calculated as a 32-bit number.)

An example of a stimulus module is shown in HDL Example 4.9.

The circuit to be tested is the two-to-one-line multiplexer described in Example 4.6. The module *t_mux_2x1_df* has no ports. The inputs for the mux are declared with a **reg** keyword and the outputs with a **wire** keyword. The mux is instantiated with the local variables. The **initial** block specifies a sequence of binary values to be applied during the simulation. The output response is checked with the **\$monitor** system task. Every time a variable in its argument changes value, the simulator displays the inputs, output, and time. The result of the simulation is listed under the simulation log in the example. It shows that *m_out_A* when *select_1* and *m_out_B* when *select_0* verifying the operation of the multiplexer.

HDL Example 4.9 (Test Bench)

```
// Test bench with stimulus for mux_2x1_df

module t_mux_2x1_df;
  wire      t_mux_out;
  reg       t_A, t_B;
  reg       t_select;
  parameter stop_time = 50;

  mux_2x1_df M1 (t_mux_out, t_A, t_B, t_select);           // Instantiation of circuit to be tested

  initial # stop_time $finish;

  initial begin                                           // Stimulus generator
    t_select = 1; t_A = 0; t_B = 1;
    #10 t_A = 1; t_B = 0;
    #10 t_select = 0;
    #10 t_A = 0; t_B = 1;
  end

  initial begin                                           // Response monitor
    // $display (" time Select A B m_out");
    // $monitor ($time,, " %b %b %b %b ", t_select, t_A, t_B, t_m_out);
```

```

    $monitor ("time =", $time,, "select = %b A = %b B = %b OUT = %b",
    t_select, t_A, t_B, t_mux_out);
end
endmodule

// Dataflow description of two-to-one-line multiplexer

// from Example 4.6
module mux_2x1_df (m_out, A, B, select);
    output      m_out;
    input       A, B;
    input       select;

    assign m_out = (select)? A : B;
endmodule

Simulation log:
select = 1 A = 0 B = 1 OUT = 0 time = 0
select = 1 A = 1 B = 0 OUT = 1 time = 10
select = 0 A = 1 B = 0 OUT = 0 time = 20
select = 0 A = 0 B = 1 OUT = 1 time = 30

```

Logic simulation is a fast and accurate method of verifying that a model of a combinational circuit is correct. There are two types of verification: functional and timing. In functional verification, we study the circuit logical operation independently of timing considerations. This can be done by deriving the truth table of the combinational circuit. In timing verification, we study the circuit's operation by including the effect of delays through the gates. This can be done by observing the waveforms at the outputs of the gates when they respond to a given input.

A **\$monitor** system task displays the output caused by the given stimulus.

A commented alternative statement having a **\$display** task would create a header that could be used with a **\$monitor** statement to eliminate the repetition of names on each line of output.

HDL Example 4.10 (Gate-Level Circuit)

The circuit has three inputs, two outputs, and nine gates. The description of the circuit follows the interconnections between the gates according to the schematic diagram of Fig. 4.2 . The stimulus for the circuit is listed in the second module. The inputs for simulating the circuit are specified with a three-bit **reg** vector *D* . *D*[2] is equivalent to input *A* , *D*[1] to input *B* , and *D*[0] to input *C* . The outputs of the circuit *F1* and *F2* are declared as **wire**. The complement of *F2* is named *F2_b* to illustrate a common industry practice for designating the complement of a signal (instead of appending *not*). This procedure follows the steps outlined in Fig. 4.34 . The repeat loop provides the seven binary numbers after 000 for the truth table. The result of the simulation generates the output truth table displayed with the example. The truth table listed shows that the circuit is a full adder.

HDL Example 4.10 (Gate-Level Circuit)

```
// Gate-level description of circuit of Fig. 4.2

module Circuit_of_Fig_4_2 (A, B, C, F1, F2);
    input A, B, C;
    output F1, F2;
    wire T1, T2, T3, F2_b, E1, E2, E3;
    or g1 (T1, A, B, C);
    and g2 (T2, A, B, C);
    and g3 (E1, A, B);
    and g4 (E2, A, C);
    and g5 (E3, B, C);
    or g6 (F2, E1, E2, E3);
    not g7 (F2_b, F2);
    and g8 (T3, T1, F2_b);
    or g9 (F1, T2, T3);
endmodule

// Stimulus to analyze the circuit

module test_circuit;
    reg [2:0] D;
    wire F1, F2;
    Circuit_of_Fig_4_2 (D[2], D[1], D[0], F1, F2);
    initial
        begin
            D = 3'b000;
            repeat (7) #10 D = D + 1'b1;
        end
    initial
        $monitor ("ABC = %b F1 = %b F2 = %b", D, F1, F2);
endmodule

Simulation log: ABC = 000 F1 = 0 F2 = 0
ABC = 001 F1 = 1 F2 = 0 ABC = 010 F1 = 1 F2 = 0
ABC = 011 F1 = 0 F2 = 1 ABC = 100 F1 = 1 F2 = 0
ABC = 101 F1 = 0 F2 = 1 ABC = 110 F1 = 0 F2 = 1
ABC = 111 F1 = 1 F2 = 1
```

Chapter-5: Synchronous Sequential Logic

5.1 INTRODUCTION

- Logic circuit can be classified as combinational circuits and sequential circuits
- Logic circuits whose outputs at any instant of time depend on the input signals present at that time are known as combinational circuits as shown in fig.4.8.

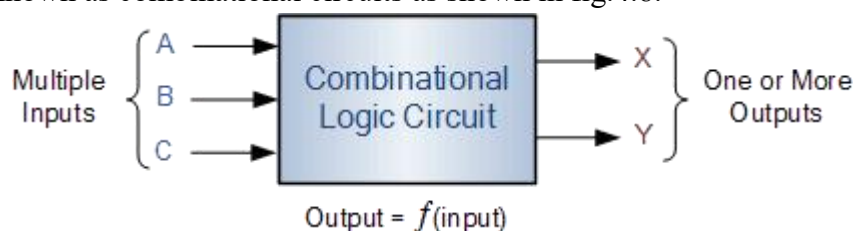


Fig. 5.1: Combinational circuit

- Logic circuits whose outputs at any instant of time depend not only on the present inputs but also on past outputs are known as sequential circuits
- In sequential circuits the outputs are fed back to input side thus the output depends on the present inputs and past outputs.

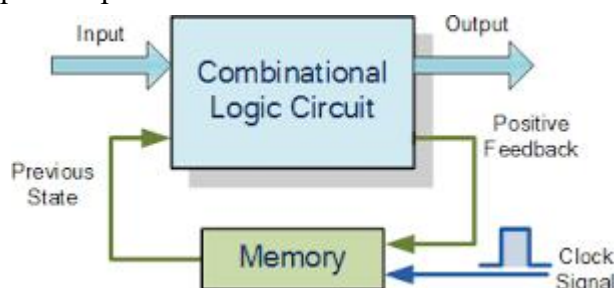


Fig. 5.2: Block diagram of sequential circuit

- A sequential circuit consists of a combinational circuit to which memory elements are added to form a feedback path as shown in fig.5.2.
 - Memory elements are the devices which are capable to store binary information (1 or 0).
 - Latches and flip-flops are commonly used memory devices in sequential circuits.
 - The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements.
 - The next state of the storage elements is also a function of external inputs and the present state.
- Thus, a sequential circuit is specified by a time sequence of inputs, outputs, and internal states.
In contrast, the outputs of combinational logic depend only on the present values of the inputs.

5.2 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. Synchronous Sequential Circuits

2. Asynchronous Sequential Circuits

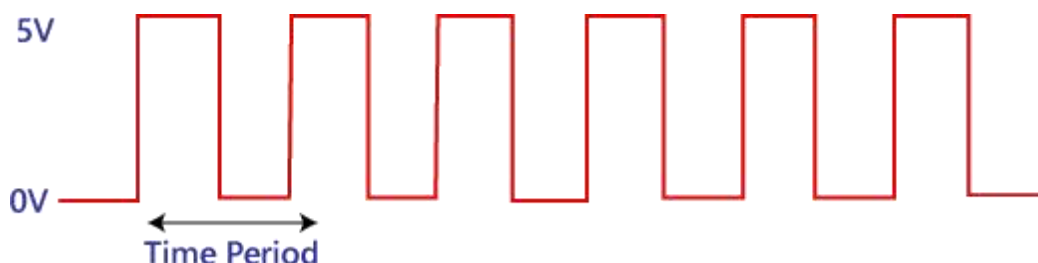
Parameters	Synchronous Sequential Circuits	Asynchronous Sequential Circuits
Definition	Synchronous sequential circuits are digital sequential circuits in which the feedback to the input for next output generation is governed by clock signals.	Asynchronous sequential circuits are digital sequential circuits in which the feedback to the input for next output generation is not governed by clock signals.

Memory Unit	In Synchronous sequential circuits, the memory unit which is being get used for governance is clocked <u>flip flop</u> .	Unlocked flip flop or time delay is used as memory element in case of Asynchronous sequential circuits.
State	The states of Synchronous sequential circuits are always predictable and thus reliable.	There are chances for the Asynchronous circuits to enter into a wrong state because of the time difference between the arrivals of inputs. This is called "race condition".
Complexity	It is easy to design Synchronous sequential circuits	The presence of feedback among logic gates causes instability issues making the design of Asynchronous sequential circuits difficult.
Performance	Due to the propagation delay of clock signal in reaching all elements of the circuit the Synchronous sequential circuits are slower in its operation speed	Since there is no clock signal delay, these are fast compared to the Synchronous Sequential Circuits
Example	Synchronous circuits are used in <u>counters</u> , <u>shift registers</u> , <u>memory units</u> .	Asynchronous circuits are used in low power and high speed operations such as simple <u>microprocessors</u> , <u>digital signal processing</u> units and in communication systems for email applications, internet access and <u>networking</u> .

Clock Signal and Triggering

Clock signal

A clock signal is a periodic signal in which ON time and OFF time need not be the same. When ON time and OFF time of the clock signal are the same, a square wave is used to represent the clock signal. Below is a diagram which represents the clock signal:



A clock signal is considered as the square wave. Sometimes, the signal stays at logic, either high 5V or low 0V, to an equal amount of time. It repeats with a certain time period, which will be equal to twice the 'ON time' or 'OFF time'.

Types of Triggering

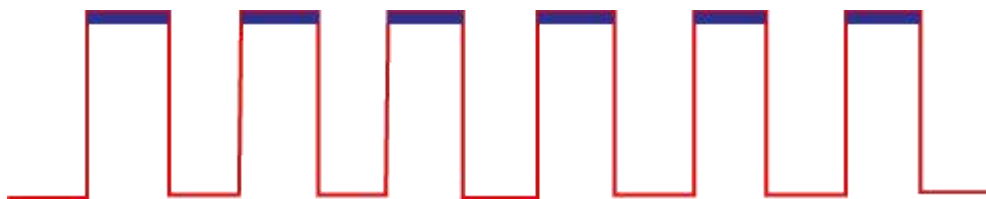
These are two types of triggering in sequential circuits:

Level triggering

The logic High and logic Low are the two levels in the clock signal. In level triggering, when the clock pulse is at a particular level, only then the circuit is activated. There are the following types of level triggering:

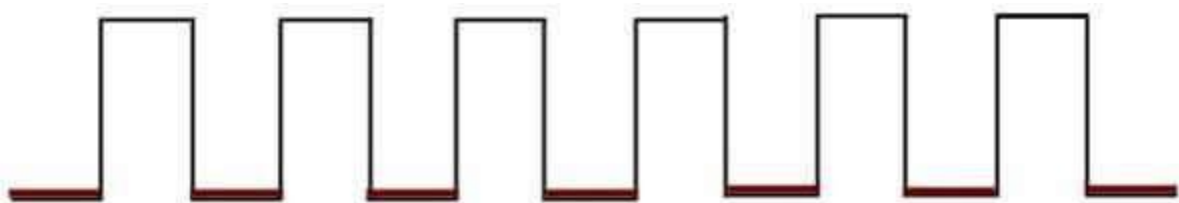
Positive level triggering

In a positive level triggering, the signal with Logic High occurs. So, in this triggering, the circuit is operated with such type of clock signal. Below is the diagram of positive level triggering:



Negative level triggering

In negative level triggering, the signal with Logic Low occurs. So, in this triggering, the circuit is operated with such type of clock signal. Below is the diagram of Negative level triggering:



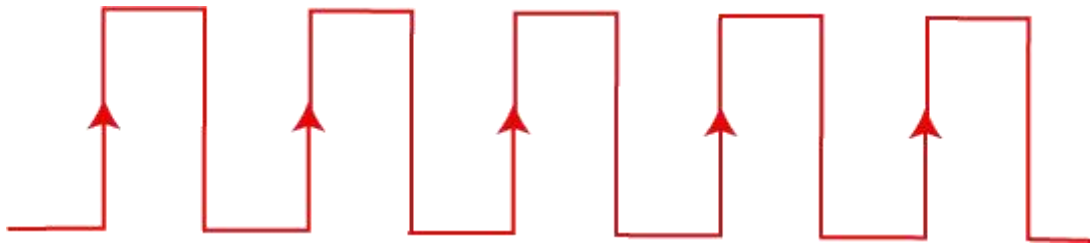
Edge triggering

In clock signal of edge triggering, two types of transitions occur, i.e., transition either from Logic Low to Logic High or Logic High to Logic Low.

Based on the transitions of the clock signal, there are the following types of edge triggering:

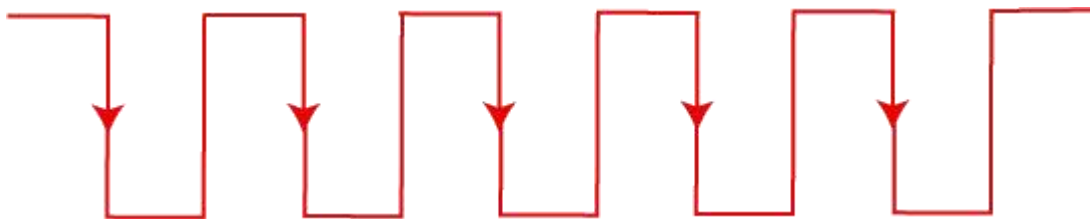
Positive edge triggering

The transition from Logic Low to Logic High occurs in the clock signal of positive edge triggering. So, in positive edge triggering, the circuit is operated with such type of clock signal. The diagram of positive edge triggering is given below.



Negative edge triggering

The transition from Logic High to Logic low occurs in the clock signal of negative edge triggering. So, in negative edge triggering, the circuit is operated with such type of clock signal. The diagram of negative edge triggering is given below.



5.3 STORAGE ELEMENTS: LATCHES

- A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states.
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.
- Storage elements that operate 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.
- 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.
- A **latch** is a device with exactly two stable states. These states are Q and $\bar{Q}$. These states Q and $\bar{Q}$ are always complement to each other.
- A latch has a feedback path, so information can be retained by the device. The latches can store one bit of data for as long as the device is powered.
- A **flip-flop** is a device very similar to a latch with exactly two stable states.
- The difference between a latch and a flip-flop is that a latch is asynchronous circuit, and flip-flop is a synchronous circuit.
- In latch the outputs can change as soon as the inputs are applied (level triggered). In flip-flop the outputs can change only when a control signal (clock) goes from high to low or low to high. i.e. latch does not have a clock signal, whereas a flip-flop always does.
- The switching circuits have feedback connections i.e. that the output of one of the gates is connected back into the input of another gate in the circuit so as to form a closed loop.



Fig.4.10: Feedback connections

- Consider the circuit in fig.4.10. If at some instant of time the inverter input is 0, this 0 will propagate through the inverter and cause the output to become 1 after the inverter delay.
- This 1 is fed back into the input, so after the propagation delay, the inverter output will become 0. When this 0 feeds back into the input, the output will again switch to 1, and so forth.
- The inverter output will continue to oscillate back and forth between 0 and 1, as shown in fig.4.10, and it will never reach a stable condition.

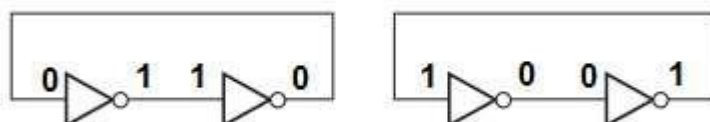


Fig.4.11: Feedback loop which has two inverters

- Consider a feedback loop which has two inverters in it, as shown in fig.4.11.
- The input to the first inverter is 0, its output will be 1. Then, the input to the second inverter will be 1, and its output will be 0. This 0 will feed back into the first inverter, but this input is already 0, no changes will occur. Therefore, the circuit is then in a stable state.

Difference between Flip-flop and Latch:

Parameter	Flip-Flop	Latch
Definition	Flip-flop is a bistable device i.e., it has two stable states that are represented as 0 and 1.	Latch is also a bistable device whose states are also represented as 0 and 1.
Basic Principle	Flip-flop utilizes an edge triggering approach.	Latch follows a level triggering approach.
Clock Signal	The clock signal is present.	The clock signal is absent.
Designed Using	You can design it using Latches along with a clock.	You can design it using Logic gates.
Sensitivity	Flip-flop is sensitive to the applied input and the clock signal.	Latches are sensitive to the applied input signal- only when enabled.
Operating Speed	It has a slow operating speed.	It has comparatively fast operating speed.
Classification	You can classify a flip-flop into a synchronous or asynchronous flip-flop.	A user cannot classify the Latch this way.
Basic Principle	Flip-flop utilizes an edge triggering approach.	Latch follows a level triggering approach.
Clock Signal	The clock signal is present.	The clock signal is absent.
Designed Using	You can design it using Latches along with a clock.	You can design it using Logic gates.
Sensitivity	Flip-flop is sensitive to the applied input and the clock signal.	Latches are sensitive to the applied input signal- only when enabled.
Operating Speed	It has a slow operating speed.	It has comparatively fast operating speed.
Classification	You can classify a flip-flop into a synchronous or asynchronous flip-flop.	A user cannot classify the Latch this way.

Basic Principle	Flip-flop utilizes an edge triggering approach.	Latch follows a level triggering approach.
Types	J-K, S-R, D, and T Flip-flops.	J-K, S-R, D, and T Latches.
Area Required	It requires more area.	It requires comparatively less area.
Uses	They constitute the building blocks of many sequential circuits such as counters.	Users can utilize these for designing sequential circuits. But they are still not generally preferred.
Input and Output	A flip-flop checks the inputs. It only changes the output at times defined by any control signal like the clock signal.	The latch responds to the changes in inputs continuously as soon as it checks the inputs.
Synchronicity	A flip-flop is synchronous. It works based on the clock signal.	A latch is asynchronous. It does not work based on the time signal.

Types of Latches

- Latch is a bi-stable device which responds to the change in input logic levels. It is said to be in SET state if output Q is high. It is said to be in RESET state if output Q is low.
- Latch can be designed with NOR gate or NAND gate.

1. Set-Reset (SR) Latch

- SR Latch can be designed using NOR gates (called as NOR latch) or NAND gates (called as NAND latch).

a) SR Latch using NOR Gate

- NOR gate latch is obtained by connecting two NOR gates back to back as shown in fig.4.12(a). Its symbol is shown in fig.4.12(b).
- When an input to a NOR gate is 1, the output is 0 irrespective of another input. When an input to a NOR gate is 0, the output depends on another input.

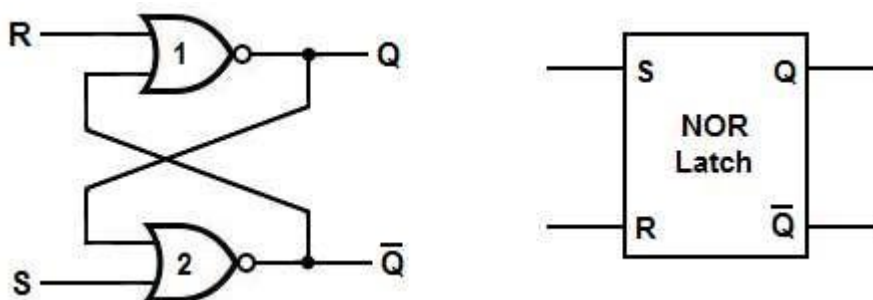


Fig.4.12: (a) NOR Gate latch (b) Symbol

Truth Table:

S	R	Q^+	$\bar{Q}^+$	State
0	0	Q	$\bar{Q}$	No change
0	1	0	1	Reset
1	0	1	0	Set
1	1	Does not exist		Forbidden

Working:

When S = 0 and R = 0

- Since both the inputs are S = 0 and R = 0, the output cannot be determined. Let us assume the output Q = 1 ($\bar{Q} = 0$), then the inputs to NOR gate 1 are R = 0 and $\bar{Q} = 0$, therefore Q = 1.

Now the inputs to NOR gate 2 are $S = 0$ and $Q = 1$, therefore $\bar{Q} = 0$. Thus there is no change in the output **$Q = Q$ and $\bar{Q} = \bar{Q}$**

When $S = 0$ and $R = 1$

- $R = 1$, will make $Q = 0$. Since $Q = 0$, it will result in $\bar{Q} = 1$. Thus **$Q = 0$ and $\bar{Q} = 1$**

When $S = 1$ and $R = 0$

- $S = 1$, will make $\bar{Q} = 0$. Since $\bar{Q} = 0$, it will result in $Q = 1$. Thus **$Q = 1$ and $\bar{Q} = 0$**

When $S = 1$ and $R = 1$

- $S = 1$ and $R = 1$, will make both Q and $\bar{Q}$ equal 0 ($\bar{Q} = Q = 0$) which is not possible. Therefore, the condition $S = 1$ and $R = 1$ does not exist for NOR latch.
- Fig.4.13 shows a timing diagram for the S-R latch.
- When S changes to 1 at time t_1 , Q changes to 1 after a short time (ϵ) (ϵ represents the response time or delay time of the latch.)
- At time t_2 , when S changes back to 0, Q does not change.
- At time t_3 , R changes to 1, and Q changes back to 0 a short time (ϵ) later.
- The duration of the S (or R) input pulse must be at least as great as ϵ in order for a change in the state of Q to occur. If $S = 1$ for a time less than ϵ , the gate output will not change and the latch will not change state.

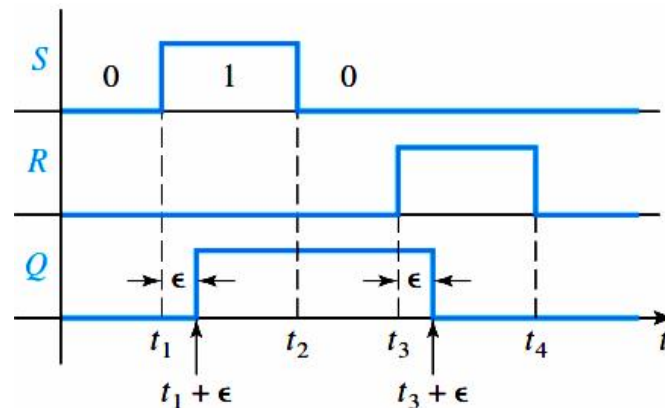


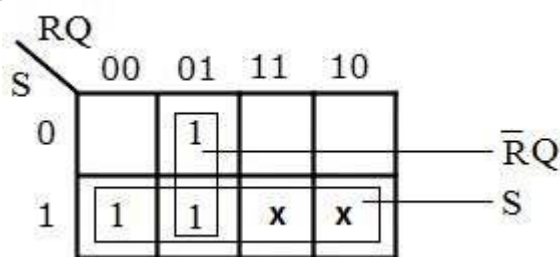
Fig.4.13: Timing diagram for the S-R latch

- For latches and flip-flops, the term **present state** to denote the state of the Q output of the latch or flip-flop at the time when any input signal changes, and the term **next state** to denote the state of the Q output after the latch or flip-flop has reacted to the input change and stabilized.
- If $Q(t)$ represent the present state and $Q(t + \epsilon)$ represent the next state. Normally Q is used to represent the present state of the latch and Q^+ is used to represent the next state.
- An equation that expresses the next state of a latch in terms of its present state and inputs is referred as a **next-state equation**, or **characteristic equation**.
- The characteristic equation for an S-R latch is obtained by constructing a truth table for the next state of Q .

Characteristics table of SR Latch:

Inputs		Outputs	
		Present State	Next State
S	R	Q	Q^+
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

**Characteristics Equation:**

$$Q^+ = S + RQ$$

b) SR Latch using NAND Gate

- SR latch can also be constructed by connecting two NAND gates back to back as shown in fig.4.15(a). Its symbol is shown in fig.4.15(b).
- When an input to a NAND gate is 0, the output is 1 irrespective of another input. When an input to a NAND gate is 1, the output depends on another input.

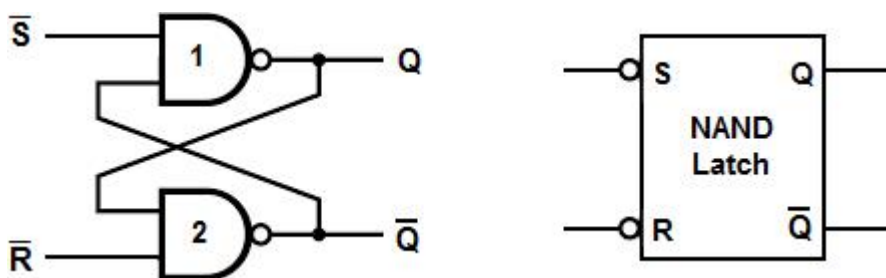


Fig.4.15: (a) NAND Gate latch (b) Symbol

Truth Table:

S	R	S	R	Q^+	$\bar{Q}^+$	State
1	1	0	0	Does not exist		Forbidden
1	0	0	1	1	0	Set
0	1	1	0	0	1	Reset
0	0	1	1	Q	$\bar{Q}$	No Change

When $S = 0$ and $R = 0$

- $\bar{S} = 0$ and $\bar{R} = 0$, will make both Q and $\bar{Q}$ equal 1 ($\bar{Q} = Q = 1$) which is not possible. Therefore the condition $\bar{S} = 0$ and $\bar{R} = 0$ does not exist for NAND latch.

When $S = 0$ and $R = 1$

- $\bar{S} = 0$, will make $Q = 1$. Since $Q = 1$, it will result in $\bar{Q} = 0$. Thus **$Q = 1$ and $\bar{Q} = 0$**

When $S = 1$ and $R = 0$

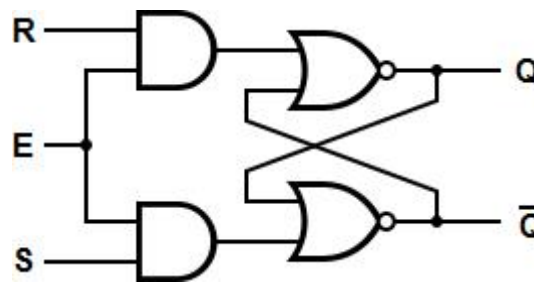
- $\bar{R} = 0$, will make $\bar{Q} = 1$. Since $\bar{Q} = 1$, it will result in $Q = 0$. Thus **$Q = 0$ and $\bar{Q} = 1$**

When $S = 1$ and $R = 1$

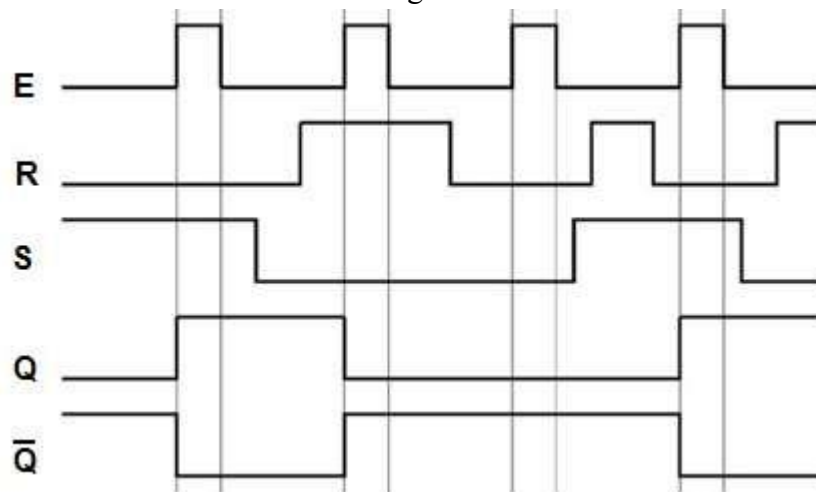
- Since both the inputs are 1, the output cannot be determined. Let us assume the output $Q = 1$ ($\bar{Q} = 0$), then the inputs to NAND gate 1 are $\bar{S} = 1$ and $\bar{Q} = 0$, therefore $Q = 1$. Now the inputs to NAND gate 2 are $\bar{R} = 0$ and $Q = 1$, therefore $\bar{Q} = 0$. Thus there is no change in the output $Q = Q$ and $\bar{Q} = \bar{Q}$

2. Gated Latches: SR latch with control input

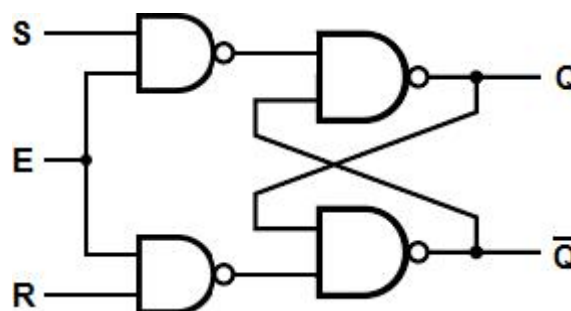
- Gated latches have an additional input called the gate or enable input.
- When the gate input is inactive (low value), the state of the latch cannot change. When the gate input is active, the latch is controlled by the other inputs and operates as a normal latch.
- The gated S-R latch using AND and NOR gates is shown in fig.4.16.

**Fig.4.16: Gated S-R latch using AND and NOR gates**

- Fig.4.17 shows a timing diagram for the S-R latch. When $E = 1$, Q changes and when $E = 0$ Q holds the previous value as shown in the fig.4.17.

**Fig.4.17: Timing diagram for gated S-R latch**

- The gated S-R latch using NAND gates is shown in fig.4.18.

**Fig.4.18: Gated S-R latch using NAND gates****Truth Table:**

E	S	R	Q^+	$Q^{\bar{F}}$	State
0	x	x	Q	$\bar{Q}$	No change

1	0	0	Q	$\bar{Q}$	No change
1	0	1	0	1	Reset
1	1	0	1	0	Set
1	1	1	Does not exist		Forbidden

3. Gated D latch (Transparent Latch):

- The gated D-latch can be constructed using a basic S-R latch, two AND gates and an inverter as shown in the fig.4.19.

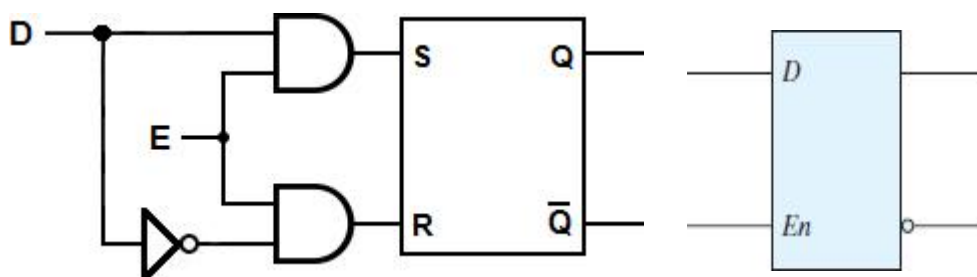


Fig.4.19: (a) Logic diagram of Gated D Latch

(b) Graphic symbol for latches

- When $E = 0$, the SR latch inputs are $S = R = 0$, so Q does not change.
- When $E = 1$ and $D = 1$, the SR latch inputs are $S = 1$ and $R = 0$, so Q is set to 1.
- When $E = 1$ and $D = 0$, the SR latch inputs are $S = 0$ and $R = 1$, so Q is reset to 0.
- In other words, when $E = 1$, the Q output follows the D input, and when $E = 0$, the Q output holds the last value of D (no state change). This type of latch is also referred to as a transparent latch because when $E = 1$, the Q output is the same as the D input.

Truth table

E	D	Q^+	$\bar{Q}^+$	State
0	X	Q	$\bar{Q}$	No change
1	0	0	1	Reset
1	1	1	0	Set

- Fig.4.18 shows a timing diagram for the D latch.

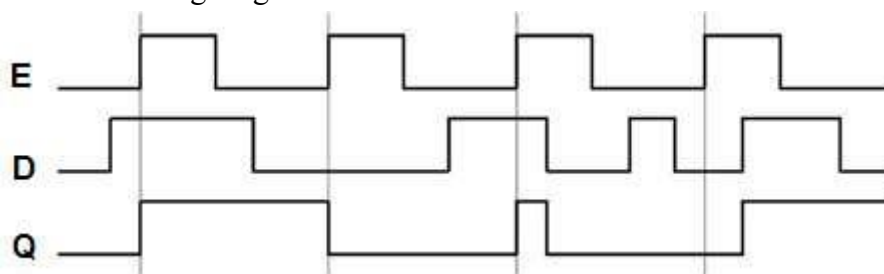


Fig.4.20: Timing diagram for the D latch

- The characteristic table and characteristic equation for the D latch is obtained by considering present state and next state outputs as shown below.

Characteristics table of D Latch:

Inputs	Outputs	
	Present State	Next State

D	Q	Q ⁺
0	0	0
0	1	0
1	0	1
1	1	1

Characteristics Equation:

$$Q^+ = D$$

5.4 STORAGE ELEMENTS: FLIP-FLOPS

Edge-Triggered Flip-Flops:

- Edge triggered flip-flop changes its output in response to the clock signal, not to a change in inputs. The small arrowhead on the flip-flop symbol identifies the clock input as shown in the fig.4.21.
- If the output changes in response to a 0 to 1 transition on the clock input, then the flip-flop is triggered on the rising edge (or positive edge) of the clock.
- If the output changes in response to a 1 to 0 transition on the clock input, then the flip-flop is triggered on the falling edge (or negative edge) of the clock.
- An inversion bubble on the clock input indicates a falling-edge trigger and no bubble indicates a rising-edge trigger as shown in the fig.4.21.
- The term active edge refers to the clock edge (rising or falling) that triggers the flip-flop state change.

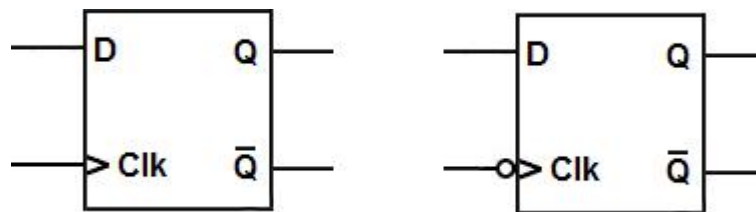


Fig.4.21: (a) Positive-edge trigger D Flip-Flops (b) Negative-edge trigger D Flip-Flops

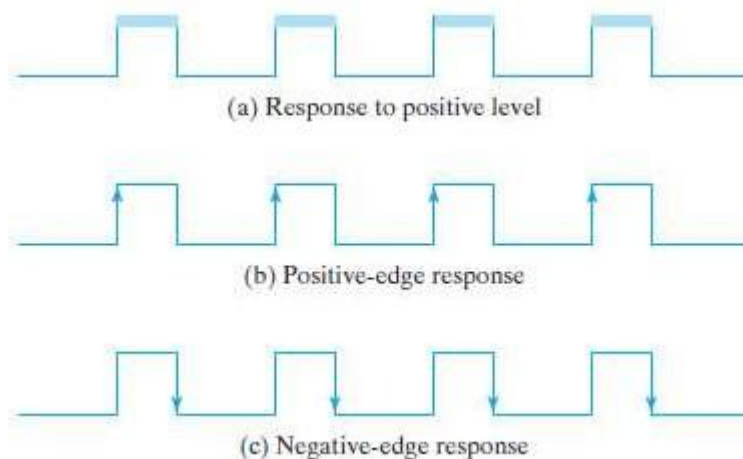


Fig. Clock response in latch and flip-flop

1. SR Flip-flop

- The SR flip flop shown in fig.4.26 has three inputs: Clock (Clk), Set (S), Reset (R) and two outputs Q and $\bar{Q}$.

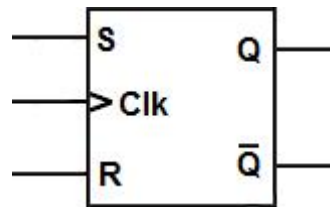


Fig.4.26: Positive Edge Trigger SR flip-flop

Truth Table

Clk	S	R	Q^+	$\bar{Q}^+$	State
0, 1, ↓	X	X	Q	$\bar{Q}$	No change
↑	0	0	Q	$\bar{Q}$	No change
↑	0	1	0	1	Reset
↑	1	0	1	0	Set
↑	1	1	Does not exists		Forbidden

- An S-R flip-flop is similar to an S-R latch, i.e. $S = 1$ sets the Q output to 1, and $R = 1$ resets the Q output to 0. The essential difference is that the flip-flop has a clock input, and the Q output can change only after an active clock edge.
- The timing diagram for SR flipflop is shown in fig.4.27. At the 1st positive edge, $S = 0$ and $R = 0$, so Q holds the previous value, even though there is a change in the inputs during the clock period.

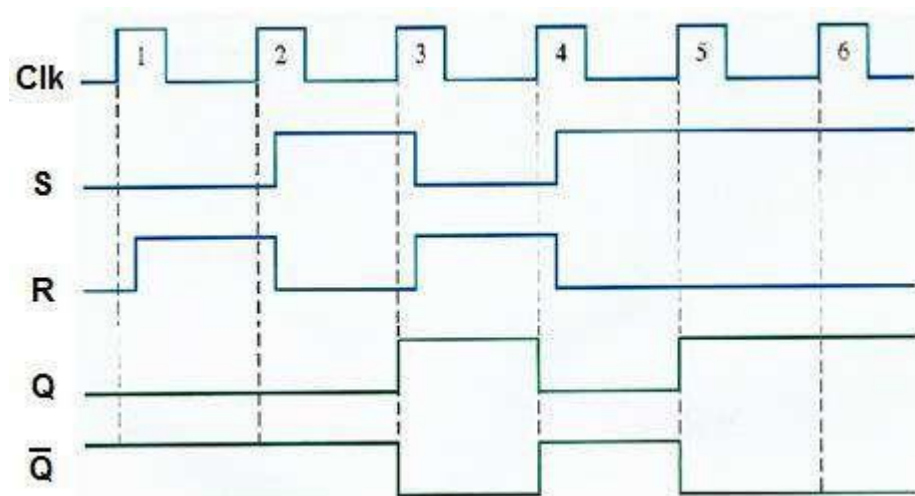


Fig.4.27: Timing diagram of SR Flip-Flop

- At the 2nd positive edge, $S = 0$ and $R = 1$, so Q goes low. At the 3rd positive edge, $S = 1$ and $R = 0$, so Q goes high. At the 4th positive edge, $S = 0$ and $R = 1$, so Q goes low.
- At the 5th and 6th positive edge, $S = 1$ and $R = 0$, so Q goes high.

2. Master-Slave SR flip-flop

- A positive edge triggered SR flip-flop is constructed from two SR latches and AND gates as shown in fig.4.28. The positive edge triggered S-R flip-flop constructed from two SR latches is often referred to as a **master-slave SR flip-flop**.
- The positive edge triggered flip-flop changes its state after the rising edge of the clock.

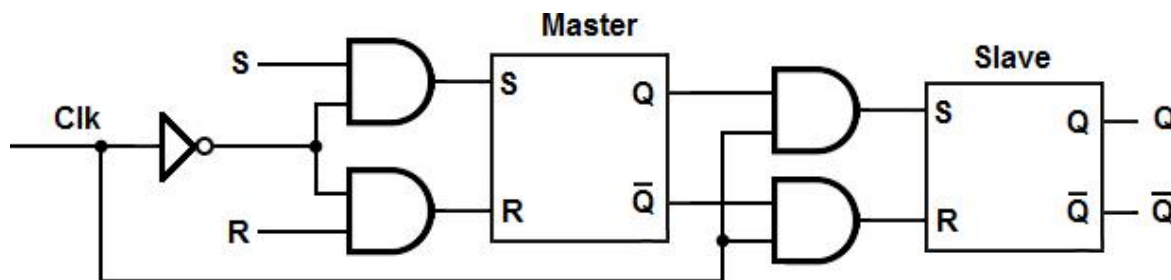


Fig.4.28: Construction of S-R Flip-Flop Implementation from two gated SR Latches

- When CLK = 0, the S and R inputs set the outputs of the master latch to the appropriate value while the slave latch holds the previous value of Q.
- When the clock changes from 0 to 1, the value of Q_M is held in the master latch is transferred to the slave latch.
- The master latch holds the value of Q_M while CLK = 1, and, hence, Q does not change.
- When the clock changes from 1 to 0, the Q value is latched in the slave, and the master can process new inputs.

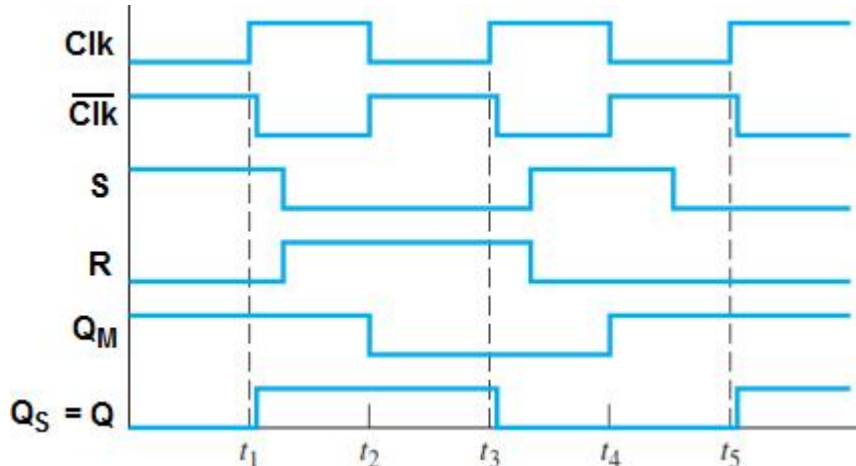


Fig.4.29: Timing diagram of Master – Slave SR Flip-Flop

- Fig.4.29 shows the timing diagram of master slave SR flipflop. Initially, S = 1 and Q changes to 1 at t_1 . Then R = 1 and Q changes to 0 at t_3 .
- At t_4 , S = 1 and R = 0, so Q_M changes to 1. Then S changes to 0 at t_5 , but Q_M does not change, so at t_5 , Q changes to 1 after the rising edge of CLK.
- However, at t_5 , S = R = 0, so the state of Q should not change. This problem can be solved if S and R inputs are allowed to change while the clock is high.

3. D Flip-Flop

- The D flip flop shown in fig.4.22 has two inputs: Clock (Clk), Data (D) and two outputs Q and $\bar{Q}$.

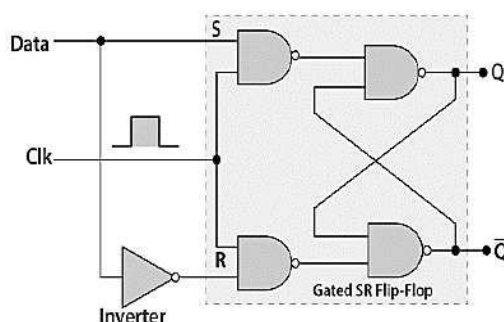
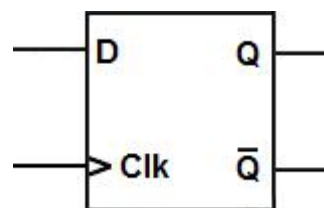


Fig.4.22: (a) Logic diagram



(b) Graphic symbol of Positive Edge Trigger D Flip flop

Truth Table

Clk	D	Q+	$\bar{Q}$	State
0, 1, ↓	X	Q	$\bar{Q}$	No change
↑	0	0	1	Reset
↑	1	1	0	Set

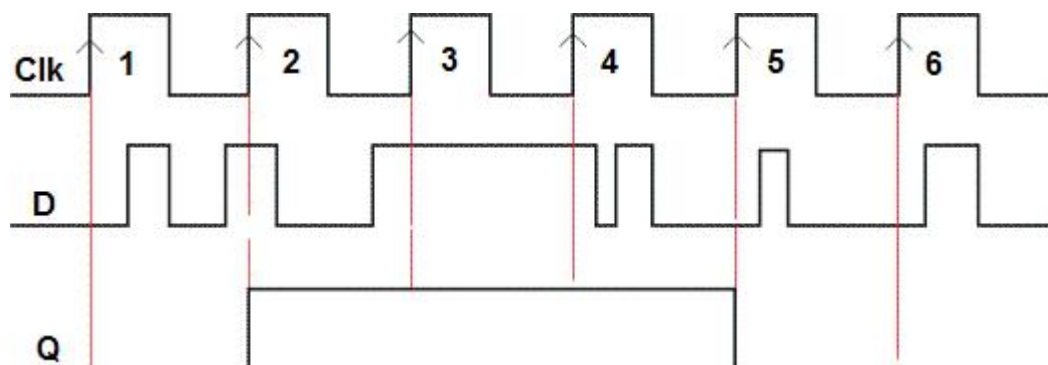


Fig.4.23: Timing diagram of D Flip-Flop

- The timing diagram for D flipflop is shown in fig.4.23.
- At the 1st positive edge, $D = 0$, so Q stays low, even though there is a positive pulse at D input during the clock period.
- At the 2nd positive edge, $D = 1$ so Q goes high. At the 3rd positive edge, D is still high, so Q stays high. At the 4th positive edge, D is still high and Q stays high.
- At the 5th positive edge, D has gone low so Q goes low. At the 6th positive edge, data is still low and Q stays low.

4. Master-slave D flip-flop

- A positive edge triggered D flip-flop can be constructed from two gated D latches and an inverter as shown in the fig.4.24.

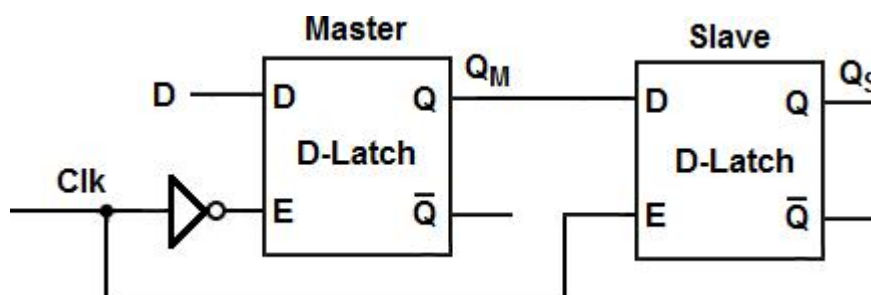


Fig.4.24: Construction of D Flip-Flop from two gated D Latches (Rising- Edge Trigger)

- The first latch is called as master latch and the second latch is called as slave latch.
- When CLK = 0, the master latch is active, hence the output of the master latch follows the D input. The slave latch is de-active; hence the output of the slave latch holds the current value Q_s .
- When CLK = 1, the master latch is de-active and the current value of D is stored in the master latch. The slave latch is active; hence the output of the slave latch follows the master latch output.
- When CLK changes back to 0, the slave latch holds the value and, then, the master latch starts following the D input again.

- The value of D at the time of the rising edge of the clock determines the value of Q, as shown in the fig.4.25 and any extra changes in D that occur between rising clock edges have no effect on Q.
- If the master latch starts following the D input before the slave latch takes on the previous value of master, then the flip-flop will not function properly. Therefore, the circuit designers must pay careful attention to timing issues when designing edge-triggered flip-flops.

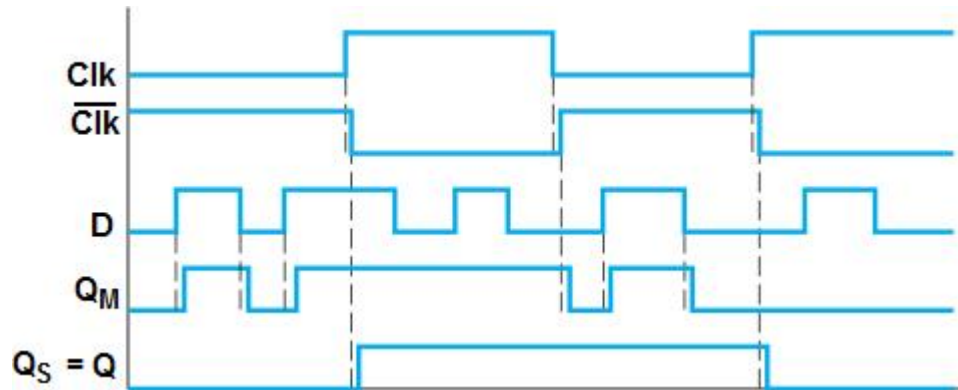


Fig.4.25: Timing diagram of Master – Slave D Flip-Flop

5. J-K Flip-Flop

- The JK flip flop shown in fig.4.30 has three inputs: Clock (Clk), J, K and two outputs Q and $\bar{Q}$.

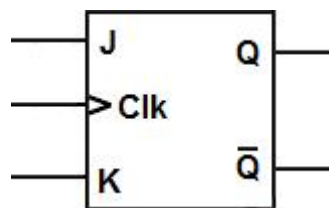


Fig.4.30: Positive Edge Trigger JK flip-flop

- The J input corresponds to S, and K corresponds to R input of SR flipflop. i.e, if J = 1 and K = 0, the flip-flop output is set to Q = 1 after the active clock edge; and if K = 1 and J = 0, the flip-flop output is reset to Q = 0 after the active edge.
- When J = K = 1, the active edge will cause Q to change from 0 to 1, or from 1 to 0.

Truth Table:

Clk	J	K	Q ⁺	$\bar{Q}^{\bar{F}}$
0, 1, ↓	X	X	Q	$\bar{Q}$
↑	0	0	Q	$\bar{Q}$
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	$\bar{Q}$	Q

- The timing diagram for JK flipflop is shown in fig.4.31. At the 1st positive edge, J = 1 and K = 0, so Q goes high, even though there is a change in the inputs during the clock period.
- At the 2nd and 3rd positive edge, J = 0 and K = 1, so Q goes low. At the 4th, 5th and 6th positive edge, J = 1 and K = 1, so Q toggles (complements) every positive edge of clock.

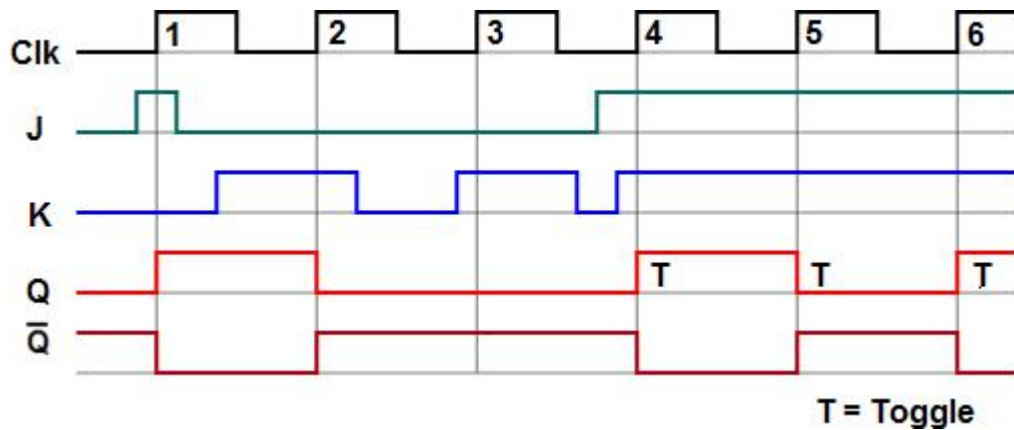
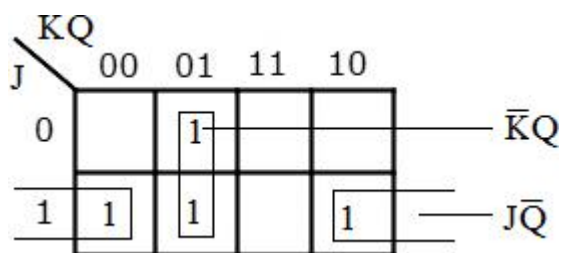


Fig.4.31: Timing diagram of JK Flip-Flop

- The characteristic table and characteristic equation for the J-K flip-flop is obtained by considering present state and next state outputs as shown below.

Characteristics table of JK Flip-flop:

Inputs		Outputs	
		Present State	Next State
J	K	Q	Q ⁺
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



Characteristics Equation:

$$Q^+ = JQ + KQ$$

6. Master-Slave J-K Flip-Flop

- A positive edge triggered JK flip-flop is constructed from two S-R latches connected in a master-slave arrangement, as shown in fig.4.32.
- This is the same circuit as for the S-R master-slave flip-flop, except S and R have been replaced with J and K, and the Q and $\bar{Q}$ outputs are feeding back into the input gates.
- Since $S = J * \bar{Q} * \bar{Clk}$ and $R = K * Q * \bar{Clk}$, only one of S and R inputs to the first latch can be 1 at any given time.

- If $Q = 0$ and $J = 1$, then $S = 1$ and $R = 0$, regardless of the value of K . If $Q = 1$ and $K = 1$, then $S = 0$ and $R = 1$, regardless of the value of J .

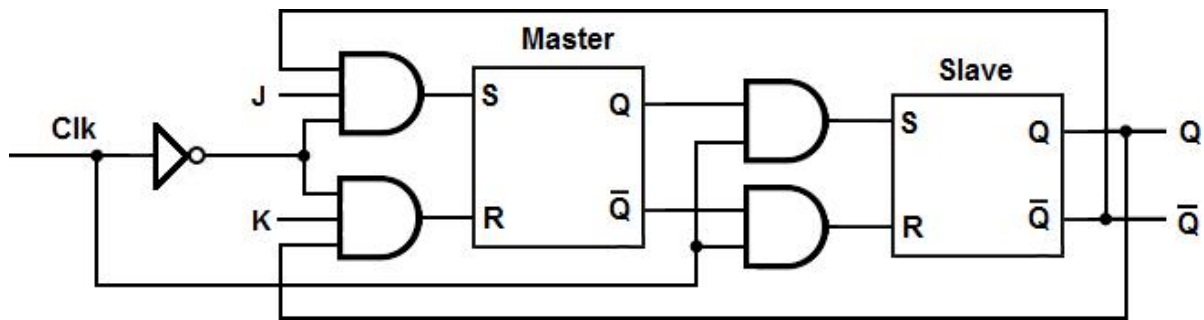
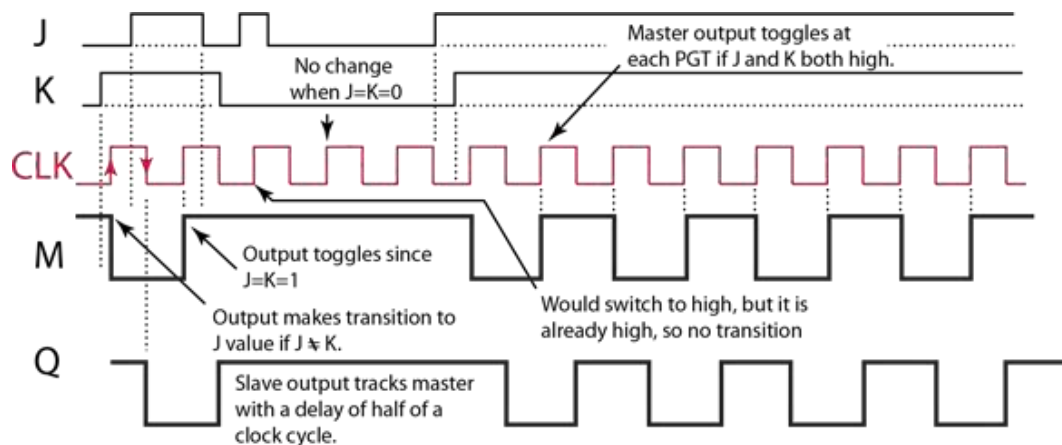
Fig.4.32: Master-Slave J-K Flip-Flop (Q Changes on Rising Edge)

Fig.4.29: Timing diagram of Master – Slave JK Flip-Flop

7. T Flip-Flop

- The T flip-flop, also called the toggle flip-flop shown in fig.4.33, has two inputs T and clock and two outputs Q and $\bar{Q}$.

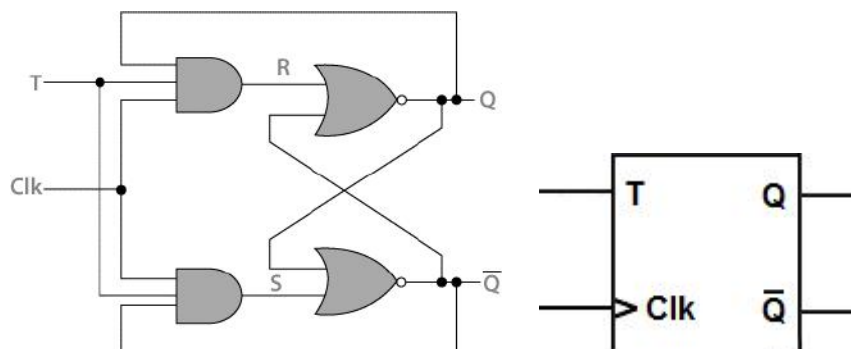


Fig.4.33: Positive Edge Trigger T flip-flop

- When $T = 1$ the flip-flop changes state after the active edge of the clock. When $T = 0$, no state change occurs.

Truth Table

Clk	T	Q^+	$\bar{Q}^+$	State
0, 1, ↓	X	Q	$\bar{Q}$	No Change
↑	0	Q	$\bar{Q}$	No Change
↑	1	$\bar{Q}$	Q	Toggle

- Fig.4.34 shows a timing diagram of T flip-flop.

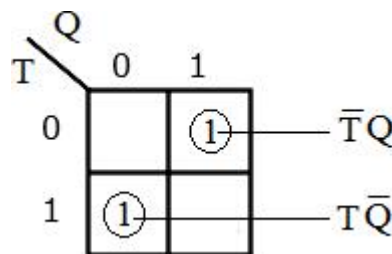
- At the 1st and 2nd rising edge of the clock signal, T input is 0 so the flip-flop output (Q) does not change the state (i.e. retains the previous value)
- At the 3rd rising edge of the clock signal, T input is 1 so the flip-flop output (Q) toggles (i.e. complements the previous value).
- Similarly, at the 4th and 5th rising edge of the clock signal, T input is 1 so the flip-flop output (Q) toggles (i.e. complements the previous value).



Fig.4.34: Timing diagram of T Flip-Flop

Characteristics table of T Flip-Flop:

Inputs	Outputs	
	Present State	Next State
T	Q	Q ⁺
0	0	0
0	1	1
1	0	1
1	1	0



Characteristics Equation:

$$Q^+ = TQ + T\bar{Q} = T \oplus Q$$

8. Other Flip-Flops

a) Implementation of T Flip-Flop from JK Flip-Flop

- One way to implement a T flip-flop is to connect the J and K inputs of a J-K flip-flop together, as shown in fig.4.35.

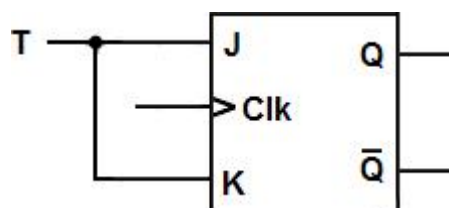


Fig.4.35: Implementation of T Flip-Flop from JK Flip-Flop

- Substituting $J = K = T$ in the J-K characteristic equation gives

$$Q^+ = JQ + K\bar{Q} = TQ + T\bar{Q} = T \oplus Q$$

which is the characteristic equation for the T flip-flop.

b) Implementation of T Flip-Flop from D Flip-Flop

- Another way to realize a T flip-flop is with a D flip-flop and an exclusive-OR gate as shown in fig.4.36.

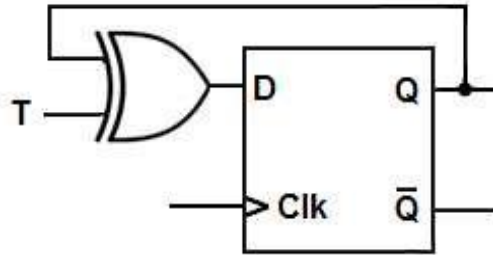


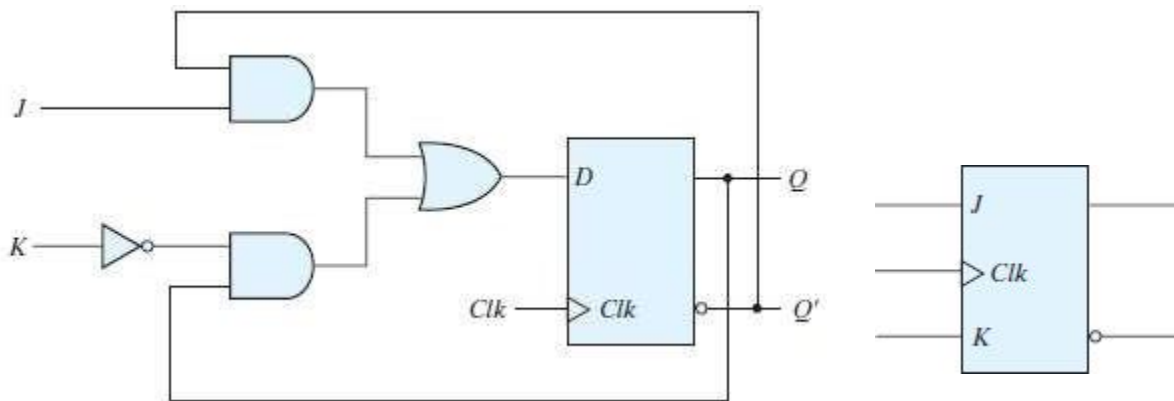
Fig.4.36: Implementation of T Flip-Flop from D Flip-Flop

- Substituting $D = Q \oplus T$, in the D characteristic equation gives

$$Q^+ = D = T \oplus Q = TQ + \bar{T}Q$$

which is the characteristic equation for the T flip-flop.

c) Implementation of JK flip-flop from D Flip-Flop



$$Q^+ = D = JQ' + K'Q$$

❖ Table of Characteristics Equation

Type of Flip Flop	Characteristics Equation
SR Flip flop	$Q^+ = S + RQ$
D Flip flop	$Q^+ = D$
JK Flip flop	$Q^+ = JQ' + KQ$
T Flip flop	$Q^+ = TQ + \bar{T}Q = T \oplus Q$