# Module-I

**One's Complement:** Complement all the bits .i.e. makes all 1s as 0s and all 0s as 1s

**Two's Complement:** One's complement+1

**SIGNED BINARY NUMBERS**

Positive integers (including zero) can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. There are three different format to represent a negative (signed) number. For example:

Three different ways to represent -9 with eight bits:

Signed magnitude representation: 10001001
signed-1's-complement representation: 11110110
signed-2's-complement representation: 11110111

*Signed Binary Numbers*

| Decimal | Signed-2's Complement | Signed-1's Complement | Signed Magnitude |
|---|---|---|---|
| +7 | 0111 | 0111 | 0111 |
| +6 | 0110 | 0110 | 0110 |
| +5 | 0101 | 0101 | 0101 |
| +4 | 0100 | 0100 | 0100 |
| +3 | 0011 | 0011 | 0011 |
| +2 | 0010 | 0010 | 0010 |
| +1 | 0001 | 0001 | 0001 |
| +0 | 0000 | 0000 | 0000 |
| −0 | — | 1111 | 1000 |
| −1 | 1111 | 1110 | 1001 |
| −2 | 1110 | 1101 | 1010 |
| −3 | 1101 | 1100 | 1011 |
| −4 | 1100 | 1011 | 1100 |
| −5 | 1011 | 1010 | 1101 |
| −6 | 1010 | 1001 | 1110 |
| −7 | 1001 | 1000 | 1111 |
| −8 | 1000 | — | — |

**Two's Complement as -ve Number**

Two's complement is -ve number because binary addition of a *n*-bit number with it's complement gives n bit result with all bits = 0s

Highest Two's Complement format +ve Number: A highest positive arithmetic number is when at msb there is 0 and all remaining bits are 1s
Lowest Two's Complement format -ve Number : A lowest negative arithmetic number is when at msb there is 1 and all remaining bits are 0s
Therefore for 8-bits

- Maximum 8-bit number = *0111 1111*( +127)
- Minimum 8-bit number = 1000 0000 ( −128)

## Subtraction using 1′ and 2′ complements

**Example: Calculate the following binary Subtraction: 11101.101 – 1011.11, then verify the result in decimal System.**

| Direct subtraction | 1's complement | 2's complement |
|---|---|---|
| 11101.101 | 11101.101 | 11101.101 |
| − 01011.110 | + 10100.001 | + 10100.010 |
| 10001.111 | [1]10001.110 | [X]10001.111 |
| | + 1 | |
| | 10001.111 | |

| Direct subtraction | 9's complement | 10's complement |
|---|---|---|
| 29.625 | 29.625 | 29.625 |
| − 11.750 | + 88.249 | + 88.250 |
| 17.875 | [1]17.874 | [X]17.875 |
| | + 1 | |
| | 17. 875 | |

**Important Note:**

- When using the complement methods in subtraction and having no additional 1 in the extreme left cell, then, this means a negative result.
- In this case, the solution is the negative of 1's complement of the result (if using 1's complement initially), or the negative of 2's complement of the result (if using 2's complement initially).

  It is shown in the following examples, where the results are –ve.

| Direct subtraction | 1's complement | 2's complement |
|---|---|---|
| 01101.101 | 01101.101 | 01101.101 |
| − 11011.110 | + 00100.001 | + 00100.010 |
| | [0]10001.110 | [0] 10001.111 |

# BINARY CODES

In the coding, when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code. The digital data is represented, stored and transmitted as group of binary bits. This group is also called as binary code. The binary code is represented by the number as well as alphanumeric letter.

**Binary-Coded Decimal Code**

Binary Coded Decimal (BCD) as the name implies is a way of representing Decimal numbers in a 4 bit binary code. BCD numbers are useful when sending data to display devices for example. The numbers 0 through 9 are the only valid BCD values. Notice in the table that the binary and BCD values are the same for the numbers 0 to 9. When we exceed the value of 9 in BCD each digit in the BCD number is now represented by a 4 bit binary value.

In this code each decimal dig it is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only first ten of these are used (0000 to 1001). The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.

| Number | Binary | BCD |
|---|---|---|
| 0 | 0000 | 0000 |
| 1 | 0001 | 0001 |
| 2 | 0010 | 0010 |
| 3 | 0011 | 0011 |
| 4 | 0100 | 0100 |
| 5 | 0101 | 0101 |
| 6 | 0110 | 0110 |
| 7 | 0111 | 0111 |
| 8 | 1000 | 1000 |
| 9 | 1001 | 1001 |
| 10 | 1010 – invalid BCD number | 0001 0000 |
| 11 | 1011 – invalid BCD number | 0001 0001 |

**BCD Addition**

Consider the addition of two decimal digits in BCD, together with a possible carry from a previous less significant pair of digits. Since each digit does not exceed 9, the sum cannot be greater than 9 + 9 + 1 = 19, with the 1 being a previous carry. Suppose we add the BCD digits as if they were binary numbers. Then the binary sum will produce a result in the range from 0 to 19. In binary, this range will be from 0000 to 10011, but in BCD, it is from 0000 to 1 1001, with the first (i.e., leftmost) 1 being a carry and the next four bits being the BCD sum. When the binary sum is equal to or less than 1001 (without a carry), the corresponding BCD digit is correct. However, when the binary sum is greater than or equal to 1010, the result is

an invalid BCD digit. The addition of 6 = (0110)2 to the binary sum converts it to the correct digit and also produces a carry as required. This is because a carry in the most significant bit position of the binary sum and a decimal carry differ by 16 - 10 = 6. Consider the following three BCD additions:

| | | | | | |
|---|---|---|---|---|---|
| 4 | 0100 | 4 | 0100 | 8 | 1000 |
| +5 | +0101 | +8 | +1000 | +9 | 1001 |
| 9 | 1001 | 12 | 1100 | 17 | 10001 |
| | | | +0110 | | +0110 |
| | | | 10010 | | 10111 |

In each case, the two BCD digits are added as if they were two binary numbers. If the binary sum is greater than or equal to 1010, we add 0110 to obtain the correct BCD sum and a carry. In the first example, the sum is equal to 9 and is the correct BCD sum. In the second example, the binary sum produces an invalid BCD digit. The addition of 0110 produces the correct BCD sum, 0010 (i.e., the number 2), and a carry. In the third example, the binary sum produces a carry. This condition occurs when the sum is greater than or equal to 16. Although the other four bits are less than 1001, the binary sum requires a correction because of the carry. Adding 0110, we obtain the required BCD sum 0111 (i.e., the number 7) and a BCD carry.

The addition of two *n*-digit unsigned BCD numbers follows the same procedure.

Consider the addition of 184 + 576 = 760 in BCD:

| BCD | 1 | 1 | | |
|---|---|---|---|---|
| | 0001 | 1000 | 0100 | 184 |
| | +0101 | 0111 | 0110 | +576 |
| Binary sum | 0111 | 10000 | 1010 | |
| Add 6 | | 0110 | 0110 | |
| BCD sum | 0111 | 0110 | 0000 | 760 |

The first, least significant pair of BCD digits produces a BCD digit sum of 0000 and a carry for the next pair of digits. The second pair of BCD digits plus a previous carry produces a digit sum of 0110 and a carry for the next pair of digits. The third pair of digits plus a carry produces a binary sum of 0111 and does not require a correction.

**Advantages of BCD Codes:** It is very similar to decimal system. We need to remember binary equivalent of decimal numbers 0 to 9 only.

**Disadvantages of BCD Codes:** The addition and subtraction of BCD have different rules. The BCD arithmetic is little more complicated. BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

## Gray Codes

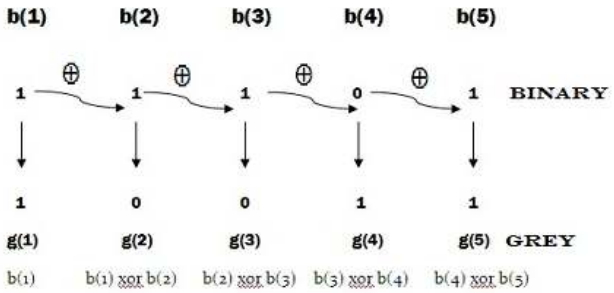
It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that has only one bit will chang e, eachtime the decimal number is incremented as shown in the table . As only one bit chang es at a time, the g ray code is called as a unit distance code. The g ray code is a cyclic code. Gray code cannot be used for arithmetic operation

| Decimal | BCD Code | Gray Code |
|---|---|---|
| 0 | 0000 | 0000 |
| 1 | 0001 | 0001 |
| 2 | 0010 | 0011 |
| 3 | 0011 | 0010 |
| 4 | 0100 | 0110 |
| 5 | 0101 | 0111 |
| 6 | 0110 | 0101 |
| 7 | 0111 | 0100 |
| 8 | 1000 | 1100 |
| 9 | 1001 | 1101 |

## Binary to Gray Conversion:

Follow the below Steps to convert Binary number to gray code.
Lets Consider the Binary number $B_1$ $B_2$ $B_3$ $B_4$ ... $B_n$ and the Gray code is $G_1$ $G_2$ $G_3$ $G_4$ ... $G_n$

1. Most significant bit ($B_1$) is same as the most significant bit in Gray Code ($B_1 = G_1$)
2. To find next bit perform Ex-OR (Exclusive OR) between the Current binary bit and previous bit.
   $G_n = B_n$ (Ex-OR) $B_{n-1}$
3. Look the below Image for Binary to Gray code Conversion



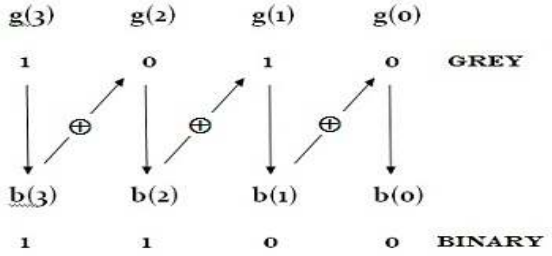

## Gary to Binary Conversion

Follow the below steps to convert Gray Code to Binary

1. Most significant bit ($G_1$) is same as the most significant bit in Binary Code ($G_1 = B_1$)
2. The next number can be obtain by taking Exclusive OR operation between the previous binary bit, and the current gray code bit and write down the value.

Repeat the Above Step until you find $B_n$

Look at the below example for Converting Binary to Gray Code.




i.e $b(3) = g(3)$

$b(2) = b(3) \oplus g(3)$

$b(1) = b(2) \oplus g(1)$

$b(0) = b(1) \oplus g(0)$

**Application of Gray code**: Gray code is popularly used in the shaft position encoders. A shaft position encoder produces a code word which represents the angular position of the shaft.

## Excess-3 code

The Excess-3 code is also called as XS-3 code. It is non-weighted code used to express decimal numbers. The Excess-3 code words are derived from the 8421 BCD code words adding (0011)2 or (3)10 to each code word in8421. The excess-3 codes are obtained as follows

Decimal Number ⟶ BCD Code ⟶ Exess-3 Code

| Decimal | BCD Code | Excess-3(BCD+0011) |
|---|---|---|
| 0 | 0000 | 0011 |
| 1 | 0001 | 0100 |
| 2 | 0010 | 0101 |
| 3 | 0011 | 0110 |
| 4 | 0100 | 0111 |
| 5 | 0101 | 1000 |
| 6 | 0110 | 1001 |
| 7 | 0111 | 1010 |
| 8 | 1000 | 1011 |
| 9 | 1001 | 1100 |

**ASCII Code**

(American Standard Code for Information Interchange)

ASCII code is a 7-bit code whereas. ASCII code is more commonly used worldwide. This standard binary code for the alphanumeric characters is the American Standard Code for Information Interchange (ASCII), which uses seven bits to code 128 characters, as shown in Table given below. The seven bits of the code are designated by $b1$ through $b7$, with $b7$ the most significant bit. The letter *A*, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code also contains 94 graphic characters that can be printed and 34 nonprinting characters used for various control functions.

The graphic characters consist of the 26 uppercase letters (A through Z), the 26 lowercase letters (a through z), the 10 numerals (0 through 9), and 32 special printable characters, such as %, *, and $.

*American Standard Code for Information Interchange (ASCII)*

| | $b_7b_6b_5$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $b_4b_3b_2b_1$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0000 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | BEL | ETB | ' | 7 | G | W | g | w |
| 1000 | BS | CAN | ( | 8 | H | X | h | x |
| 1001 | HT | EM | ) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [ | k | { |
| 1100 | FF | FS | , | < | L | \ | l | \| |
| 1101 | CR | GS | − | = | M | ] | m | } |
| 1110 | SO | RS | . | > | N | ^ | n | ~ |
| 1111 | SI | US | / | ? | O | − | o | DEL |

The 34 control characters are designated in the following ASCII table with abbreviated names

**Control Characters**

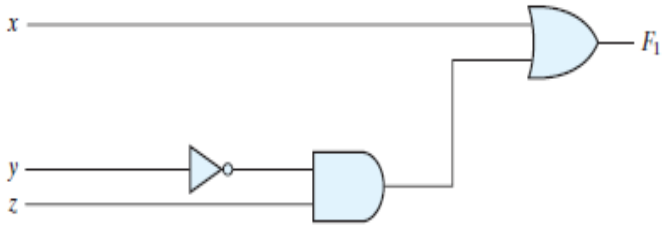
| | | | |
|---|---|---|---|
| NUL | Null | DLE | Data-link escape |
| SOH | Start of heading | DC1 | Device control 1 |
| STX | Start of text | DC2 | Device control 2 |
| ETX | End of text | DC3 | Device control 3 |
| EOT | End of transmission | DC4 | Device control 4 |
| ENQ | Enquiry | NAK | Negative acknowledge |
| ACK | Acknowledge | SYN | Synchronous idle |
| BEL | Bell | ETB | End-of-transmission block |
| BS | Backspace | CAN | Cancel |
| HT | Horizontal tab | EM | End of medium |
| LF | Line feed | SUB | Substitute |
| VT | Vertical tab | ESC | Escape |
| FF | Form feed | FS | File separator |
| CR | Carriage return | GS | Group separator |
| SO | Shift out | RS | Record separator |
| SI | Shift in | US | Unit separator |
| SP | Space | DEL | Delete |

# BOOLEAN FUNCTIONS

Boolean algebra is an algebra that deals with binary variables and logic operations. A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols. For a given value of the binary variables, the function can be equal to either 1 or 0. As an example, consider the Boolean function

$$F1 = x + y'z$$

A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure. The logic-circuit diagram (also called a schematic) for $F1$ is shown in Figure using different logic gates.




Truth Tables for $F_1$

| x | y | z | $F_1$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

## CANONICAL AND STANDARD FORMS

### Minterms and Maxterms

A binary variable may appear either in its normal form ($x$) or in its complement form ($x'$)
Now consider two binary variables $x$ and $y$ combined with an AND operation. Since each variable may appear in either form, there are four possible combinations: $x'y'$, $x'y$, $xy'$ and $xy$. Each of these four AND terms is called a *minterm*, or a *standard product*. In a similar manner, $n$ variables can be combined to form $2n$ minterms. The $2n$ different minterms may be determined by a method similar to the one shown in Table given below for three variables. Each minterm is obtained from an AND term of the $n$ variables, with each variable being primed if the corresponding bit of the binary number is a 0 and unprimed if a 1. A symbol for each minterm is also shown in the table and is of the form $mj$, where the subscript $j$ denotes the decimal equivalent of the binary number of the minterm designated.
In a similar fashion, $n$ variables forming an OR term, with each variable being primed or unprimed, provide $2n$ possible combinations, called *maxterms*, or *standard sums*. The eight maxterms for three variables, together with their symbolic designations, are listed in Table 2.3 . Any $2n$ maxterms for $n$ variables may be determined similarly. It is important to note that (1) each maxterm is obtained from an OR term of the $n$ variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1, and (2) each maxterm is the complement of its corresponding minterm and vice versa.
**A Boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a 1 in the function and then taking the OR of all those terms.** For example, the function $f1$ in Table 2.4 is

determined by expressing the combinations 001, 100, and 111 as $x\_y\_z$, $xy\_z\_$, and $xyz$, respectively. Since each one of these minterms results in $f1 = 1$, we have

$$f1 = x'y'z + xy'z' + xyz = m1 + m4 + m7 = \sum(1,4,7)$$

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

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

**Standard Forms**

Another way to express Boolean functions is in *standard* form.

-SOP (*sum of products*)

-POS (*product of sums*)

The *sum of products* is a Boolean expression containing AND terms, called *product terms,* with one or more literals each. The *sum* denotes the ORing of these terms. An example of a function expressed as a sum of products is

$$F1 = y' + xy + x'yz'$$

The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation. The logic diagram of a sum-of-products expression consists of a group of AND gates followed by a single OR gate. This configuration pattern is shown in Fig. 2.3 (a). Each product term requires an AND gate, except for a term with a single literal. The logic sum is formed with an OR gate whose inputs are the outputs of the AND gates and the single literal. It is assumed that the input variables are directly available in their complements, so inverters are not included in the diagram. This circuit configuration is referred to as a *two-level implementation*.

A *product of sums* is a Boolean expression containing OR terms, called *sum terms*. Each term may have any number of literals. The *product* denotes the ANDing of these terms. An example of a function expressed as a product of sums is

$$F2 = x\,(y' + z)\,(x' + y + z')$$

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

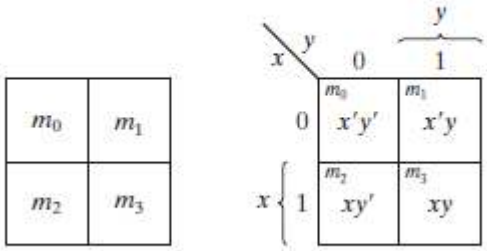
# Gate-Level Minimization

*Gate-level minimization* is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit. The map method presented here provides a simple, straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method is also known as the *Karnaugh map* or *K-map*. A K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized.

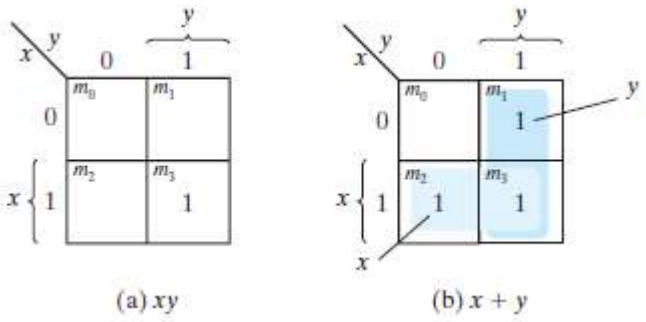
**Two-Variable K-Map**

There are four minterms for two variables; hence, the map consists of four squares, one for each minterm.. The 0 and 1 marked in each row and column designate the values of variables. Variable $x$ appears primed in row 0 and unprimed in row 1. Similarly, $y$ appears primed in column 0 and unprimed in column 1. If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables. As an example, the function $xy$ is shown in Figure (a). Since $xy$ is equal to $m3$, a 1 is placed inside the square that belongs to $m3$. Similarly, the function $x + y$ is represented in the map of Figure (b) by three squares marked with 1's. These squares are found from the minterms of the function:

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

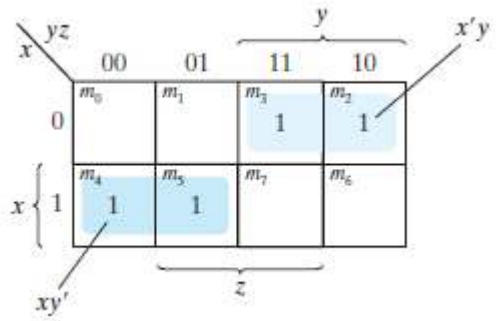



The three squares could also have been determined from the intersection of variable $x$ in the second row and variable $y$ in the second column, which encloses the area belonging to $x$ or $y$ . In each example, the minterms at which the function is asserted are marked with a 1.



**Three-Variable K-Map**

**Example**: Simplify the Boolean function
$F(x,y,z) = \sum(2,3,4,5)$




The simplified function is: $F(x,y,z)$ = x'y+yx'

**Example**
For the Boolean function
$F = A'C + A'B + AB'C + BC$

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

Note that $F$ is a *sum of products*. Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term, $A'C$, are found in Fig-a from the coincidence of $A'$ (first row) and $C$ (two middle columns) to give squares 001 and 011. Note that, in marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term.
This happens with the second term, $A'B$, which has 1's in squares 011 and 010. Square 011 is common with the first term, $A'C$, though, so only one 1 is marked in it. Continuing in this fashion, we determine that the term $AB'C$ belongs in square 101, corresponding to minterm 5, and the term $BC$ has two 1's in squares 011 and 111. The function has a total of five minterms, as indicated by the five 1's in the map of Figure . The minterms are read directly from the map to be 1, 2, 3, 5, and 7. The function can be expressed in sum-of-minterms form as $F(x,y,z) = \sum(1,2,3,5,7)$
The sum-of-products expression, as originally given, has too many terms. It can be simplified, as shown in the map, to an expression with only two terms:

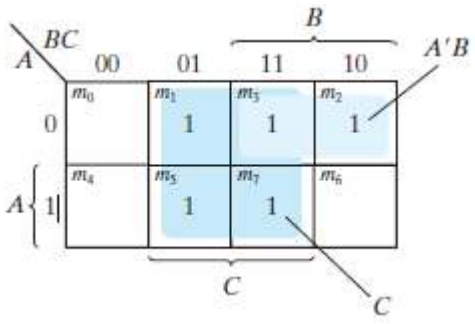
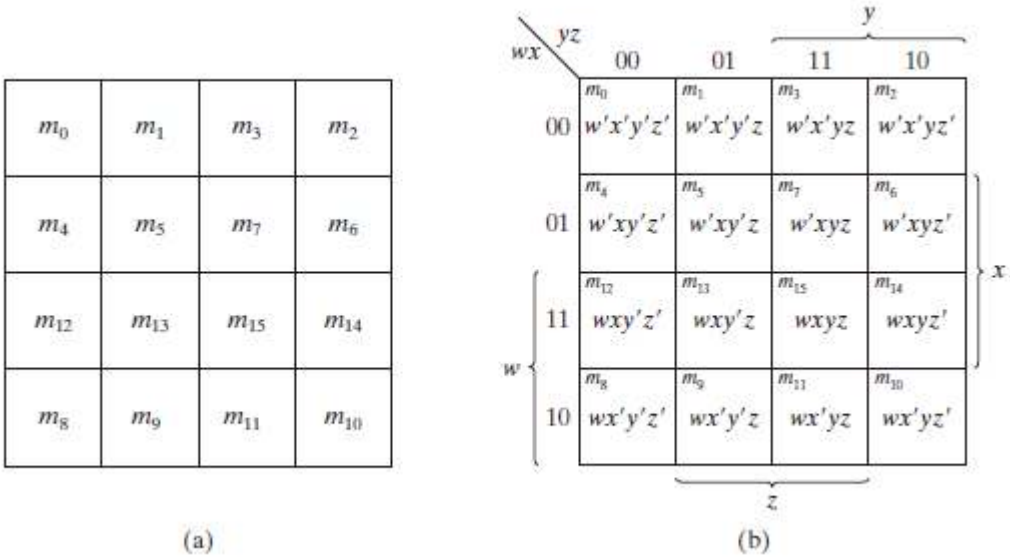



**Fig-a**

The simplified function is: $F = C + A'B$

**FOUR-VARIABLE K-MAP**

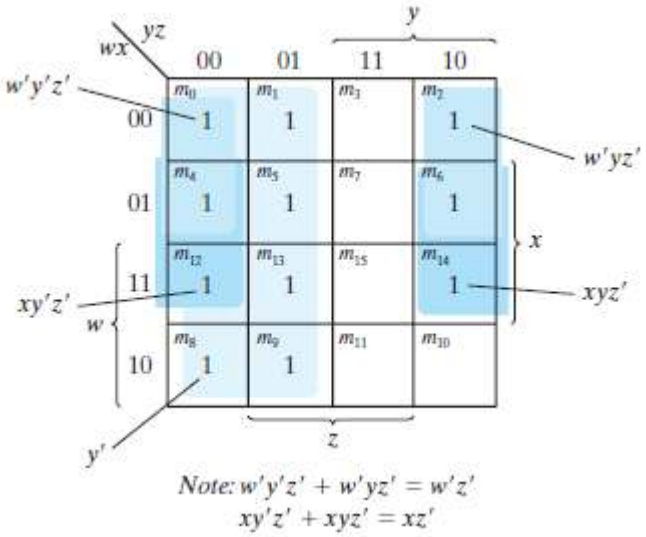
Here for four variables we have 16 minterms. So a map of 16 squares is required. Lets say 4 variables are w,x,y,z




***Example***

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

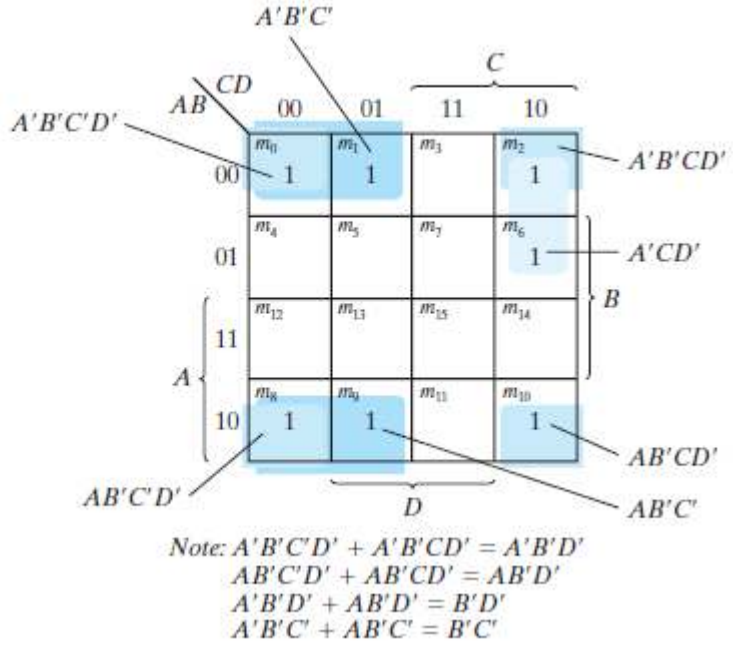
Eight adjacent squares marked with 1's can be combined to form the one literal term $z'$. The remaining three 1's on the right cannot be combined to give a simplified term; they must be combined as two or four adjacent squares. The larger the number of squares combined, the smaller is the number of literals in the term. In this example, the top two 1's on the right are combined with the top two 1's on the left to give the term $w'z'$ . Note that it is permissible to use the same square more than once. We are now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term with four literals), we combine it with squares already used to form an area of four adjacent squares. These squares make up the two middle rows and the two end columns, giving the term $xz'$.




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

**Example**

Simplify the Boolean function: $F$ = A'B'C'+ B'CD' + A'BCD' + AB'C'




The simplified function is: $F$ = B'D' + A'CD' + B'C'

**Five-Variable Map**

A five-variable map needs 32 squares and a six-variable map needs 64 squares. It can be explained in the class.

**PRODUCT-OF-SUMS SIMPLIFICATION**

By using K-map the simplified function is in SOP format.

So if we want to get Final answer in POS format, we need to simplified for the F' and at the end take complement of F' to get F in POS form. It can be easily explained in the following example

**Example**

Simplify the following Boolean function into (a) sum-of-products form and (b) product-of-sums form:

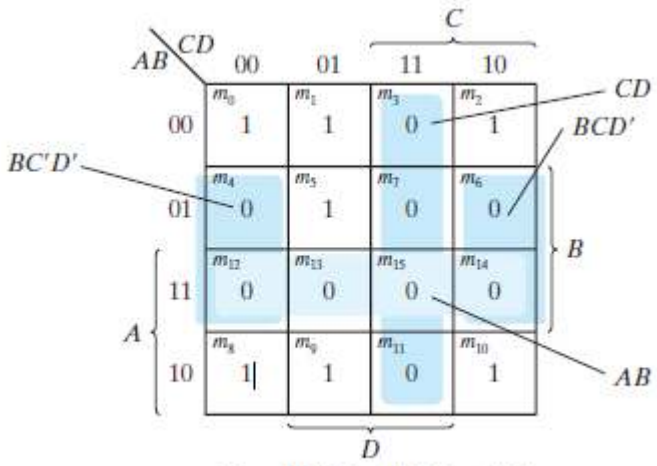
$F$ (A,B,C,D)= $\sum(0,1,2,5,8,9,10)$

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

(a) $F = B'D' + B'C' + A'C'D$
If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:
$F\_ = AB + CD + BD'$
Applying DeMorgan's theorem, we obtain the simplified function in productof-sums form:

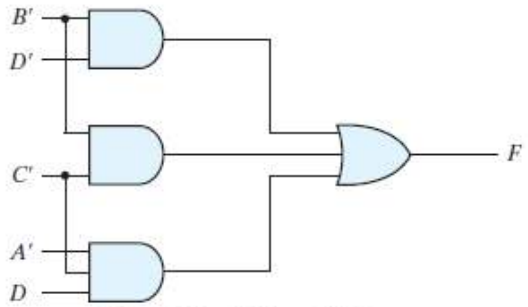
(b) $F = (A' + B')\,(C' + D')\,(B' + D)$

The gate-level implementation of the simplified expressions obtained in Example 3.7 is shown in Fig. 3.13 . The sum-of-products expression is implemented in (a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in (b) in its product-of-sums

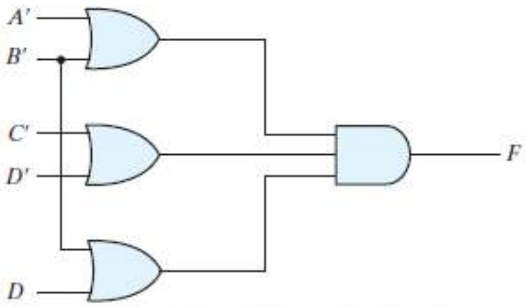


Note: $BC'D' + BCD' = BD'$

$\boldsymbol{F\,(A, B, C, D)} = \sum(\mathbf{0, 1, 2, 5, 8, 9, 10}) = B'D' + B'C' + A'C'\boldsymbol{D} =$
$\mathbf{(}A' + B'\mathbf{)(}\,C' + D'\mathbf{)(}\,B' + \boldsymbol{D}\mathbf{)}$



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



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

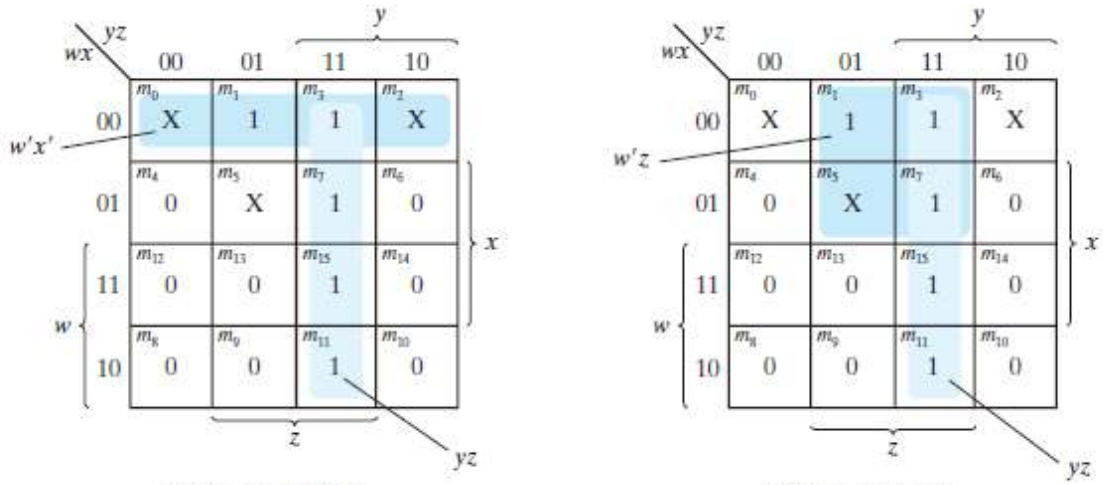
## DON'T-CARE CONDITIONS

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. In practice, in some applications the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified. Functions that have unspecified outputs for some input combinations are called *incompletely*

*specified functions* . these unspecified minterms are don't-care conditions and can be used on a map to provide further simplification of the Boolean expression.
A don't-care minterm is a combination of variables whose logical value is not specified. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to *F* for the particular minterm.
In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

**Example**
Simplify the Boolean function: $F(w,x,y,z)=\sum(1,3,7,11,15)$
which has the don't-care conditions: $d(w,x,y,z)=\sum(0,2,5)$

The minterms of *F* are the variable combinations that make the function equal to 1. The minterms of *d* are the don't-care minterms that may be assigned either 0 or 1(marked by X's) and the remaining squares are filled with 0's. To get the simplified expression in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified. The term *yz* covers the four minterms in the third column. The remaining minterm, *m*1, can be combined




with minterm *m*3 to give the three-literal term *w'x'z*. However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In Figure (a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

$$F = yz + w'x'$$

In Figure (b), don't-care minterm 5 is included with the 1's, and the simplified function is now $F = yz + w'z$
Either one of the preceding two expressions satisfies the conditions stated for this example.

**More examples will be done in the Class.**

## NAND AND NOR IMPLEMENTATION

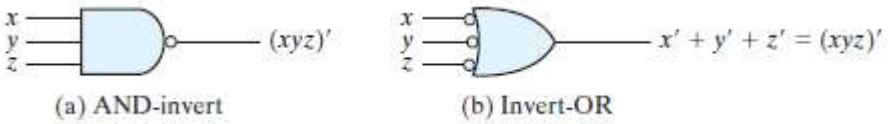
Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. Rules and procedures have been developed for the conversion from Boolean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

### NAND Circuits

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

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

Two equivalent graphic symbols for the NAND gate are shown in Figure . The AND-invert symbol has been defined previously and consists




(a) AND-invert (b) Invert-OR

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

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

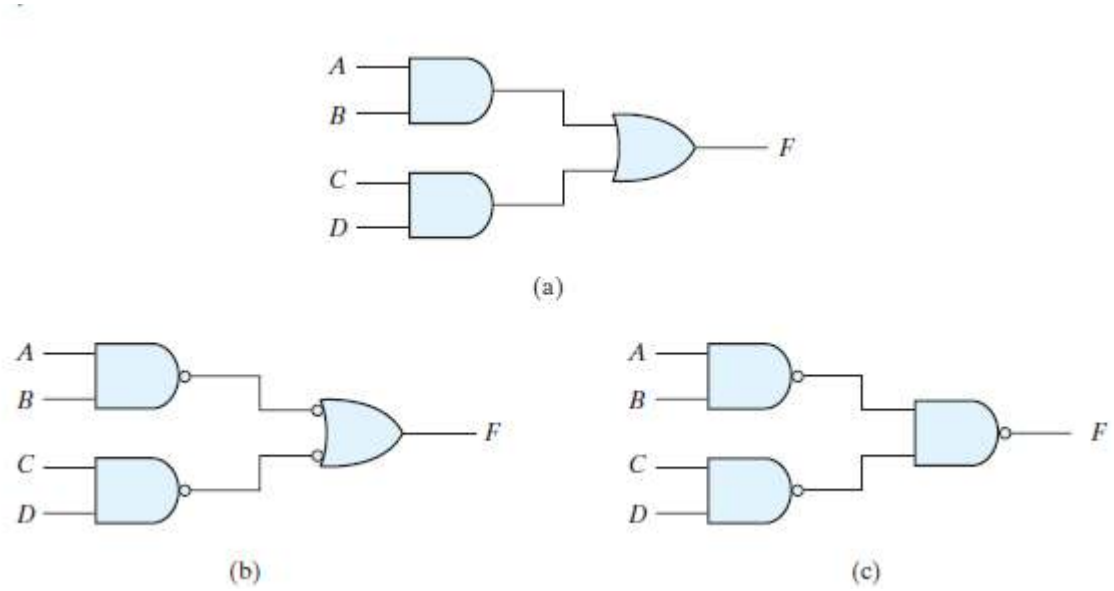
### Two-Level Implementation

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

$$F = AB + CD$$

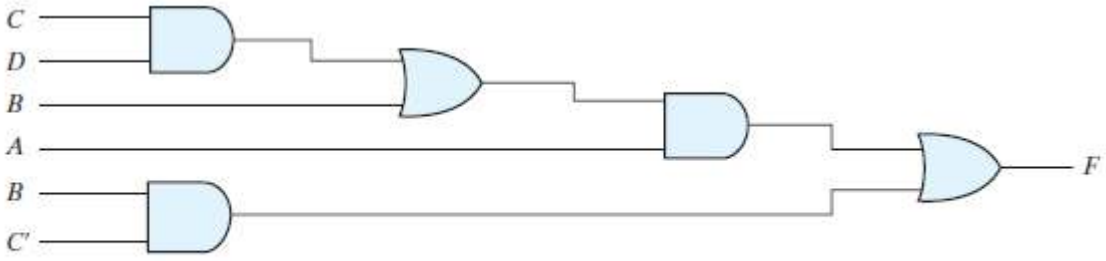
The function is implemented in Figure(a) with AND and OR gates.

In Figure(b), the AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an OR-invert graphic symbol. Remember that a bubble denotes complementation and two bubbles along the same line represent double complementation, so both can be removed. Removing the bubbles on the gates of (b) produces the circuit of (a).

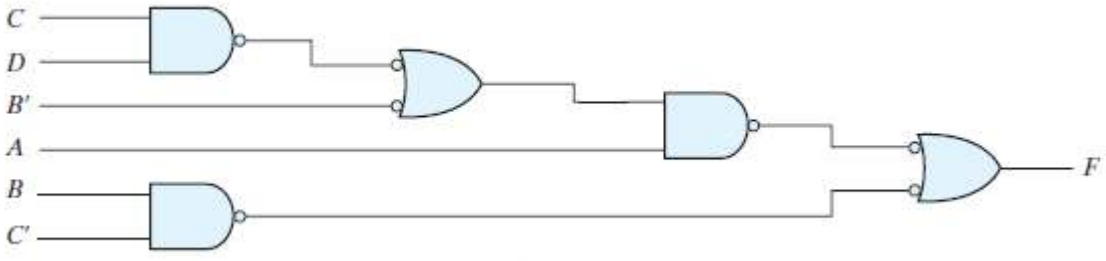



**Multilevel NAND Circuits**

*Implement :* $F = A\,(CD + B) + BC'$



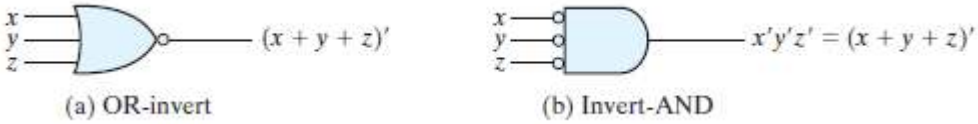

(a) AND–OR gates




(b) NAND gates

## NOR Implementation

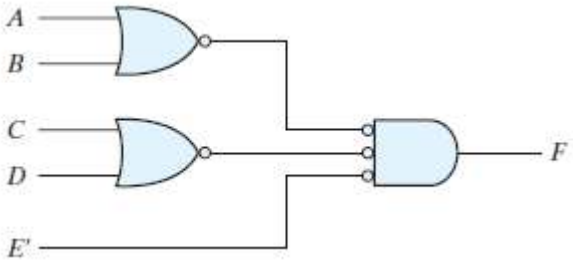
Two equivalent graphic symbols for the NOR gate are shown in Figure .



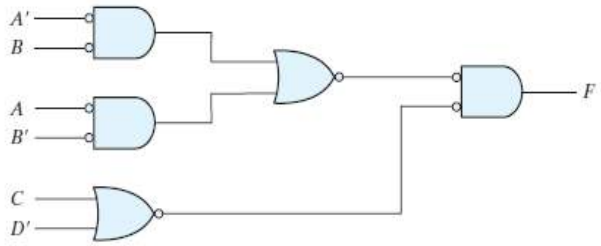

(a) OR-invert (b) Invert-AND

NOR implementation of a function expressed as a product of sums. Then the OR-AND pattern can be easily converted to NOR gates.For example:

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




Ex.The Boolean function for this circuit is: $F = (AB\_ + A\_B)(C + D\_)$




## THER TWO-LEVEL IMPLEMENTATIONS

The eight *nondegenerate* forms are as follows:

| | |
|---|---|
| AND–OR | OR–AND |
| NAND–NAND | NOR–NOR |
| NOR–OR | NAND–AND |
| OR–NAND | AND–NOR |

## AND–OR–INVERT Implementation

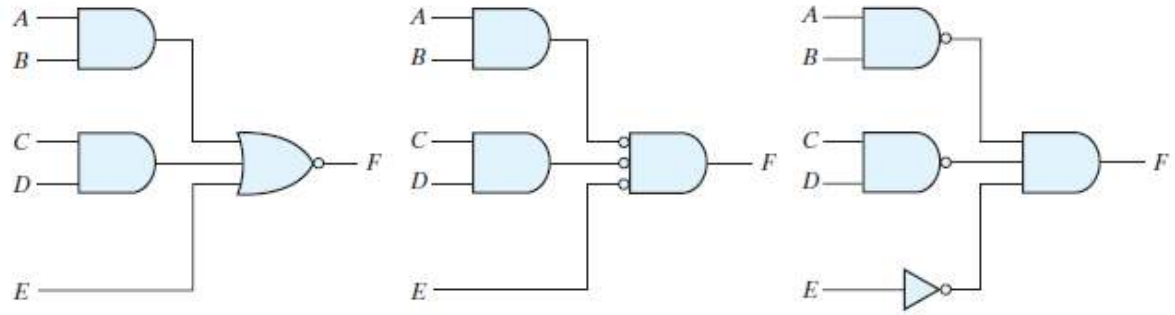
The two forms, NAND–AND and AND–NOR, are equivalent and can be treated together. Both perform the AND–OR–INVERT function, as shown in Figure given below . The AND–NOR form resembles the AND–OR form, but with an inversion done by the bubble in the output of the NOR gate. It implements the function

$$F = (AB + CD + E)\_$$

By using the alternative graphic symbol for the NOR gate, we obtain the diagram of Figure (b). Note that the single variable *E* is *not* complemented, because the only

change made is in the graphic symbol of the NOR gate. Now we move the bubble from the input terminal of the second-level gate to the output terminals of the first-level gates. An inverter is needed for the single variable in order to compensate for the bubble. Alternatively, the inverter can be removed, provided that input $E$ is complemented. The circuit of Fig. 3.27 (c) is a NAND–AND form and was shown in Fig. 3.26 to implement the AND–OR–INVERT function.

An AND–OR implementation requires an expression in sum-of-products form. The AND–OR–INVERT implementation is similar, except for the inversion. Therefore, if the *complement* of the function is simplified into sum-of-products form (by combining the 0's in the map), it will be possible to implement $F\_$ with the AND–OR part of the function. When $F\_$ passes through the always present output inversion (the INVERT part), it will




(a) AND–NOR (b) AND–NOR (c) NAND–AND

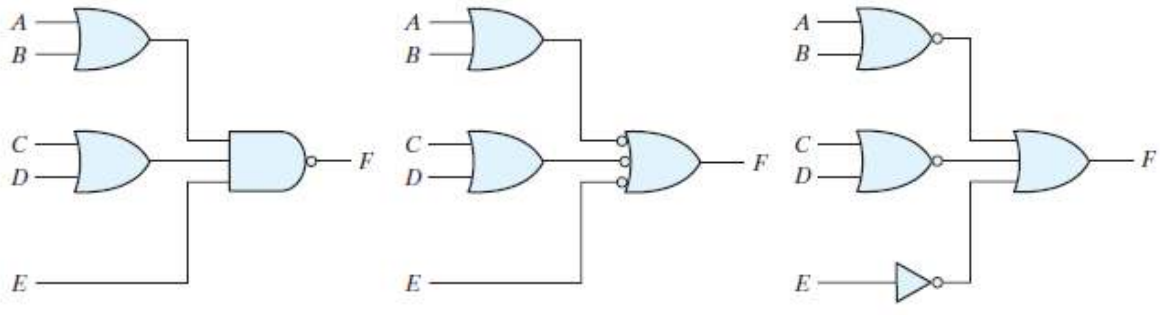
## OR–AND–INVERT Implementation

The OR–NAND and NOR–OR forms perform the OR–AND–INVERT function, as shown in Fig. 3.28 . The OR–NAND form resembles the OR–AND form, except for the inversion done by the bubble in the NAND gate. It implements the function

$$F = 3(A + B)(C + D)E'$$

By using the alternative graphic symbol for the NAND gate, we obtain the diagram of Figure (b). The circuit in Figure(c) is obtained by moving the small circles from the inputs of the second-level gate to the outputs of the first-level gates. The circuit of Fig. (c) is a NOR–OR form and was shown in Fig. 3.26 to implement the OR–AND–INVERT function.

The OR–AND–INVERT implementation requires an expression in product-of-sums form. If the complement of the function is simplified into that form, we can implement $F'$ with the OR–AND part of the function. When $F\_$ passes through the INVERT part, we obtain the complement of $F'$, or $F$ , in the output.




(a) OR–NAND (b) OR–NAND (c) NOR–OR

*Implementation with Other Two-Level Forms*

| Equivalent Nondegenerate Form (a) | Equivalent Nondegenerate Form (b)* | Implements the Function | Simplify $F'$ into | To Get an Output of |
|---|---|---|---|---|
| AND–NOR | NAND–AND | AND–OR–INVERT | Sum-of-products form by combining 0's in the map. | $F$ |
| OR–NAND | NOR–OR | OR–AND–INVERT | Product-of-sums form by combining 1's in the map and then complementing. | $F$ |

*Form (b) requires an inverter for a single literal term.

## EXCLUSIVE-OR FUNCTION

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

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

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

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

The exclusive-NOR is equal to 1 if both $x$ and $y$ are equal to 1 or if both are equal to 0.
The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation:

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

The following identities apply to the exclusive-OR operation:

$x \oplus 0 = x$

$x \oplus 1 = x'$

$x \oplus x = 0$

$x \oplus x' = 1$

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

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

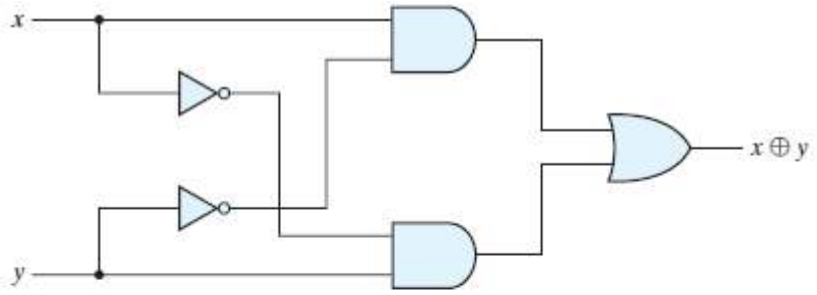
$$A \oplus B = B \oplus A$$

and

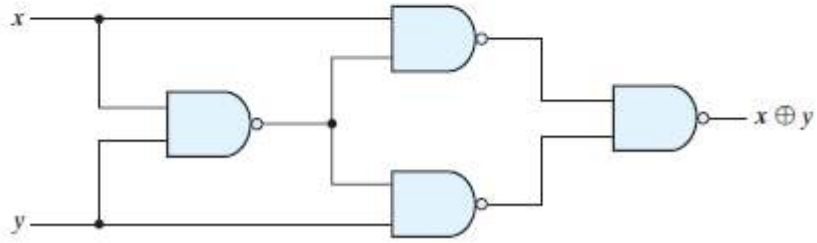
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation. It also means that we can evaluate a three-variable exclusive-OR operation in any order, and for this reason, three or more variables can be expressed without parentheses. This would imply the possibility of using exclusive-OR gates with three or more inputs. However, multiple-input exclusive-OR gates are difficult to fabricate with hardware. In fact, even a two-input function is usually constructed with other types of gates. A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate, as shown in Fig (a). Figure (b) shows the implementation of the exclusive-OR with four NAND gates. The first NAND gate performs the operation $(xy)' = (x' + y')$. The other two-level NAND circuit

produces the sum of products of its inputs:
$(x' + y') + (x' + y') = xy' + x'y = x \oplus y$




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




(b) Exclusive-OR with NAND gates

**Odd Function**

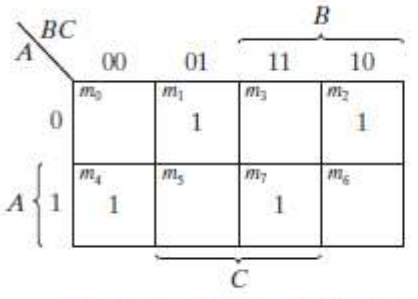
The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the ⊕ symbol with its equivalent Boolean expression. In particular, the three-variable case can be converted to a Boolean expression as follows:

$A \oplus B \oplus C = (AB' + A'B)C' + (AB + A'B')C$

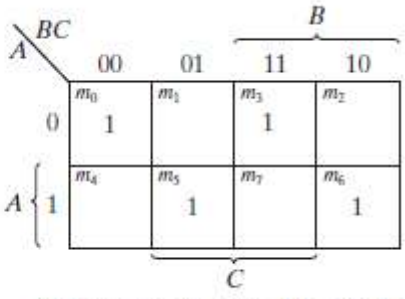
$= AB'C' + A'BC' + ABC + A'B'C$

$= \sum(1, 2, 4, 7)$

The Boolean expression clearly indicates that the three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1. Contrary to the two-variable case, in which only one variable must be equal to 1, in the case of three or more variables the requirement is that an odd number of variables be equal to 1. As a consequence, the multiple-variable exclusive-OR operation is defined as an *odd function.*




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




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

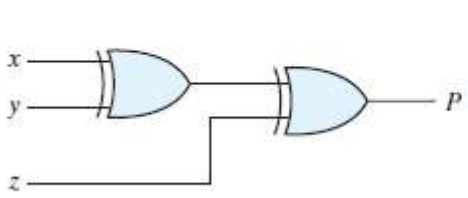
**Parity Generation and Checking**

Exclusive-OR functions are very useful in systems requiring error detection and correction codes. A parity bit is used for the purpose of detecting errors during the transmission of inary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a *parity generator.* The circuit that checks the parity in the receiver is called a *parity checker.*

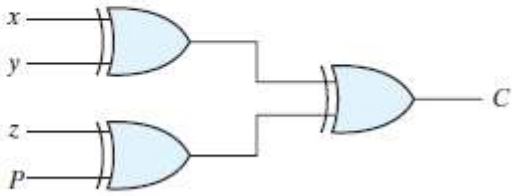
As an example, consider a three-bit message to be transmitted together with an even-parity bit. Table shows the truth table for the parity generator. The three bits $x$, $y$, and $z$ constitute the message and are the inputs to the circuit. The parity bit $P$ is the output. For even parity, the bit $P$ must be generated to make the total number of 1's (including $P$ ) even. From the truth table, we see that $P$ constitutes an

*Even-Parity-Generator Truth Table*

| Three-Bit Message | | | Parity Bit |
|---|---|---|---|
| x | y | z | P |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |




(a) 3-bit even parity generator




(b) 4-bit even parity checker

odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's. Therefore, $P$ can be expressed as a three-variable exclusive-OR function:

$$P = x \oplus y \oplus z$$

The logic diagram for the parity generator can be drawn using XOR gates. The three bits in the message, together with the parity bit, are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission. Since the information was transmitted with even parity, the four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission. The output of the parity checker, denoted by $C$ , will be equal to 1 if an error occurs—that is, if the four bits received have an odd number of 1's. The truth table for the even-parity checker is given

below. From it, we see that the function $C$ consists of the eight minterms with binary numerical values having an odd number of 1's. The table corresponds to the map of Fig.(a), which

*Even-Parity-Checker Truth Table*

| Four Bits Received | | | | Parity Error Check |
|---|---|---|---|---|
| *x* | *y* | *z* | *P* | *C* |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

represents an odd function. The parity checker can be implemented with exclusive-OR gates:

$$C = x \oplus y \oplus z \oplus P$$

The logic diagram of the parity checker can be drawn using XOR gates. It is obvious from the foregoing example that parity generation and checking circuits always have an output function that includes half of the minterms whose numerical values have either an odd or even number of 1's. As a consequence, they can be implemented with exclusive-OR gates. A function with an even number of 1's is the complement of an odd function. It is implemented with exclusive-OR gates, except that the gate associated with the output must be an exclusive-NOR to provide the required complementation.

# MODULE 2

# Combinational Logic:

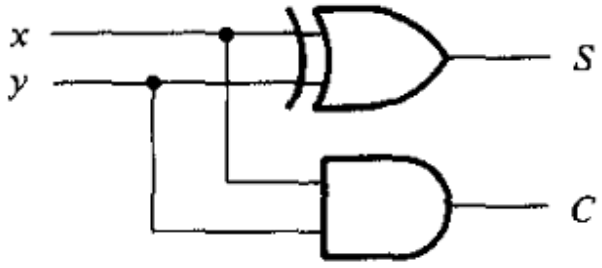
## Combinational Circuits

Circuits in which all outputs at any given time depend only on the inputs at that time are called combinationallogic circuits.

A combinational circuit performs a specific information-processing operation fully specified logically by a set of Boolean functions. Sequential circuits employ memory elements (binary cells) in addition to logic gates. Their outputs are a function of the inputs and the state of the memory elements. The state of memory elements, in turn, is a function of previous inputs. As a consequence, the outputs of a sequential circuit depend not only on present inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states.

**1 BINARY ADDERS**

One of the most important tasks performed by a digital computer is the operation of adding two binary numbers.




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

Half adder circuit

| x | y | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Truth table of half adder

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

S=x'y+xy'

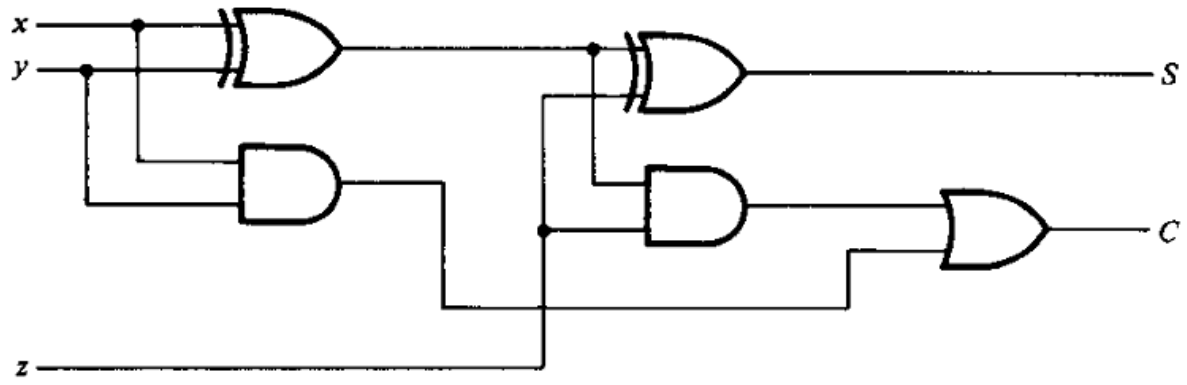
C = xy

## Full-Adder

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs.

Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position.The two outputs are designated by the symbols S for sum and C for carry. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full-adder is

| x | y | z | C | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |




Implementation of full adder

This implementation uses the following Boolean expressions:

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

$$C = xy + xz + yz$$

## Ripple-Carry Adder

The problem of adding two multidigit binary numbers has the following form.

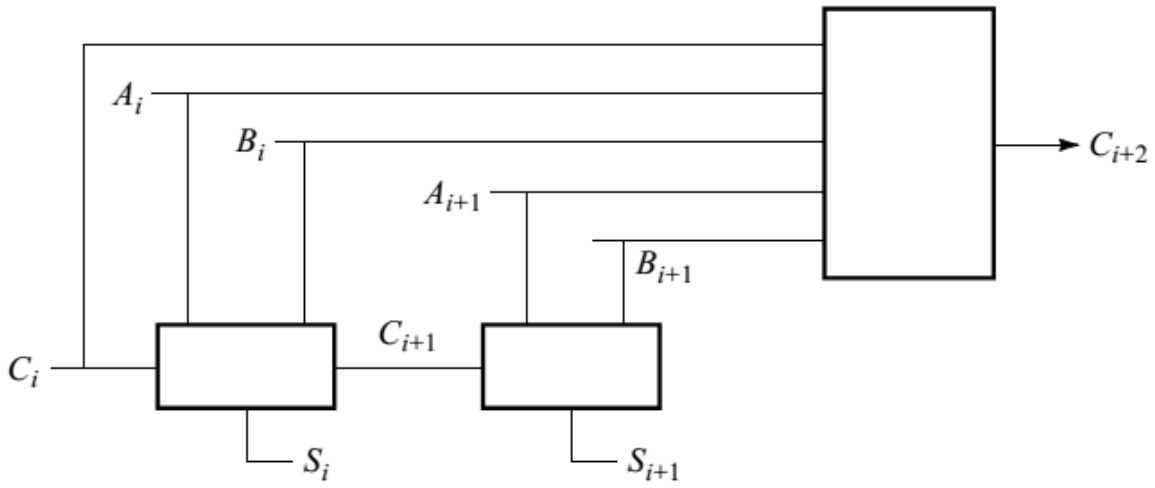
Two n-bit binary numbers are available, with all digits being presented in parallel. The addition is performed by using a full adder to add each corresponding pair of digits, one from each number. The full adders are connected in tandem so that the carry out from one stage becomes the carry into the next stage, as illustrated for the case of four-digit numbers in Figure 4. Thus, the carry ripples through each stage. For binary addition, the carry into the first (least significant) stage is 0. The last carry out (the overflowcarry) becomes the most significant bit of the (n+ 1)-bit sum.

Since the carry of each full adder has a propagation delay of $2t_p$, the totaldelay in carrying out the sum of two n-bit numbers is $2nt_p$. Not every pair of twon-bit numbers will experience this much delay. Take the following two numbers

as an example:

101010

010101




Carry-lookahead circuit schematic

Assuming that the carry into the first stage is zero, no carries are generated at any stage in taking the sum. Hence, there will be no carry ripple, and so no propagation delay along the carry chain.

However, to handle the general case, provision must be made for the worst case; no new numbers should be presented for addition before the total delayrepresented by the worst case. The maximum addition speed, thus, is limited by the worst case of carry propagation delay.

# Carry-Lookahead Adder

In contemplating the addition of two n-digit binary numbers, we were appalled by the thought of a single combinational circuit with all those inputs. So we considered the repeated use of a simpler circuit, a full adder, with the least possible number of inputs. But what is gained in circuit simplicity with this approach is lost in speed. Since the speed is limited by the delay in the carry function, some of the lost speed might be regained if we could design a circuit—just for the carry—with more inputs than 2 but not as many as 2n. Suppose that several full-adder stages are treated as a unit. The inputs to the unit are the carry into the unit as well as the input digits to all the full adders in that unit. Then perhaps the carry out could be obtained faster than the ripple carry through the same number of full adders.

These concepts are illustrated in above figure with a unit consisting of just twofull adders and a carry-lookahead circuit. The four digits to be added, as well as the input carry $C_i$, are present simultaneously. It is possible to get an expressionfor the carry out,$C_{i+2}$, from the unit by using the expression for the carry of thefull adder

For reasons which will become clear shortly, let’s attach names to the twoterms in the carry expression, changing the names of the variables to A and Bfrom xand y in accordance with above figure.

Define the generated carry $G_i$and the propagated carry $P_i$for the ith full adder as follows:

$$Gi= AiBi$$

$$Pi= Ai \oplus Bi$$

Inserting these into the expression for the carryout gives

$$C_{i+1}= A_iB_i+ C_i(A_i \oplus B_i) = G_i+ P_iC_i$$

# Subtractor

Subtractor circuits take two binary numbers as input and subtract one binary number input from the other binary number input. Similar to adders, it gives out two outputs, difference and borrow (carry-in the case of Adder). There are two types of subtractors.

- Half Subtractor
- Full Subtractor

### Half Subtractor

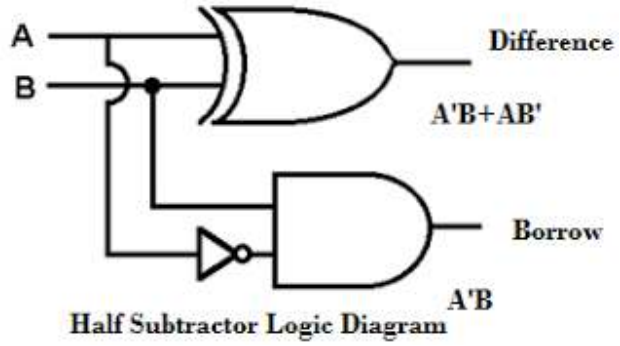
The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow). The logic symbol and truth table are shown below.

#### Truth Table

| X | Y | D | B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

Difference=A'B+AB'=A $\oplus$ B
Borrow=A'B

The logic Diagram of Half Subtractor is shown below.



Half Subtractor Logic Diagram

# Full Subtractor

A full subtractor is a combinational circuit that performs subtraction involving three bits, namely minuend, subtrahend, and borrow-in. so it allows *cascading* which results in the possibility of **multi-bit subtraction**. The truth table for a full subtractor is given below.

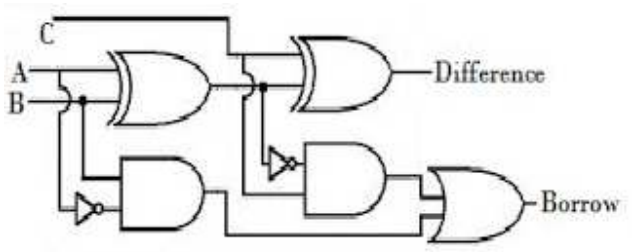
**Truth Table**

| X | Y | Bin | D | Bout |
|---|---|---|---|---|
| 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 |
| | | | | |

Difference=A'B'C+A'BB'+AB'C'+ABC
Reduce it like adder
Then We got
Difference=A ⊕ B ⊕ C
Borrow=A'B'C+A'BC'+A'BC+ABC
=A'B'C+A'BC'+A'BC+A'BC+A'BC+ABC ----------> A'BC=A'BC+A'BC+A'BC

=A'C(B'+B)+A'B(C'+C)+BC(A'+A)
Borrow=A'C+A'B+BC

The logic diagram of Full Subtractor is shown below




Full Subtractor-Logic Diagram

# Multiplexers:-

Multiplexing means transmitting a large number of information units over a smaller number of channels or lines. A digital multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line.

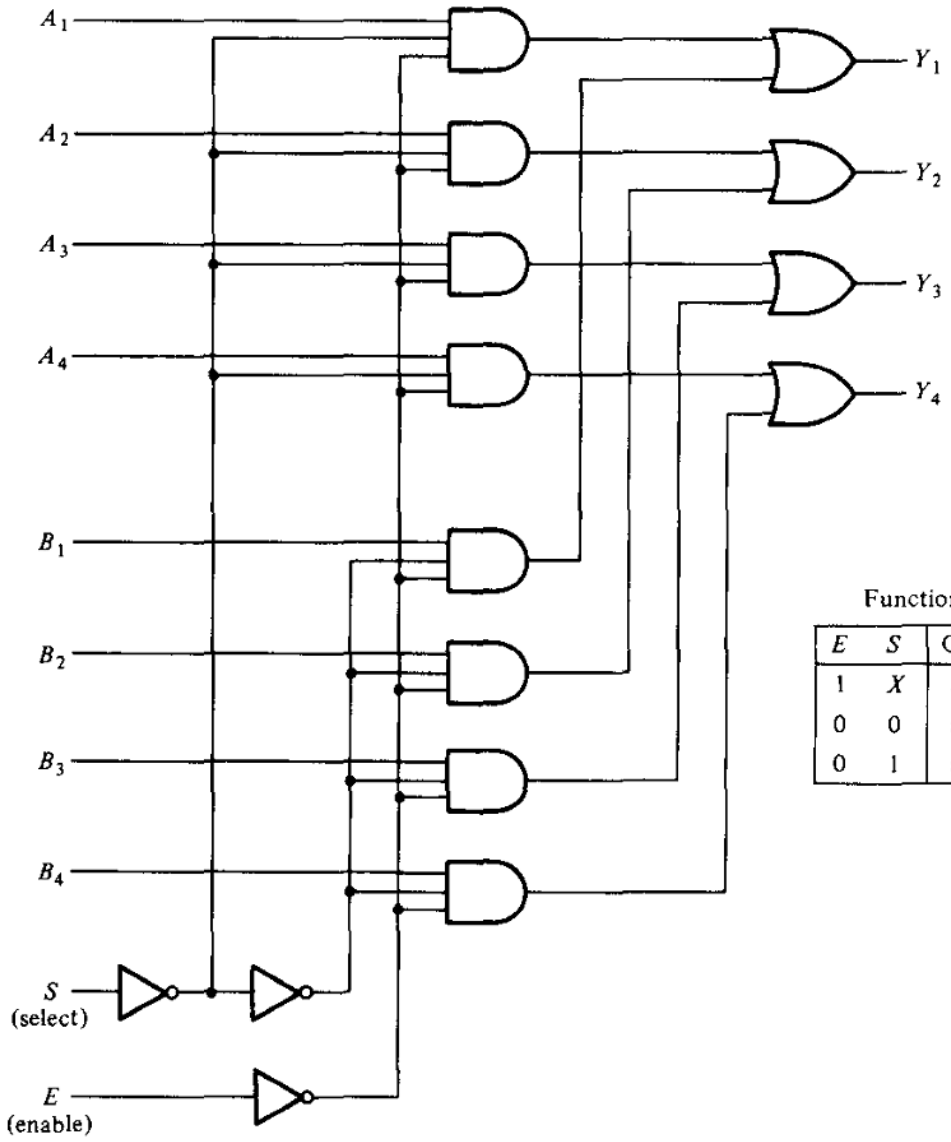
The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2" input lines and n selection lines whose bit combinations determine which input is selected.

A 4-to-l-line multiplexer is shown in given Fig. Each of the four input lines, $I_0$ to $I_3$is applied to one input of an AND gate. Selection lines $S_I$ and $S_o$ are decoded to select a particular AND gate. The function table,shown in given , lists the input-to-output path for each possible bit combination of the selection lines. When this MSI function is used in the design of a digital system, it is represented in block diagram form, as shown in given Fig. To demonstrate the circuit operaiion, consider the case when $S_IS_O$ = 10. The AND gate associated with input J, has two of its inputs equal to 1 and the third input connected to J,. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of $I_2$ thusproviding a path from the selected input to the output. A multiplexer is also called a dutu selector, since it selects one of many inputs and steers the binary information to the output line.

The AND gates and inverters in the multiplexer resemble a decoder circuit and, indeed, they decode the input-selection lines. In general, a $2^n$-to-1-line multiplexer is constructed from an n-to-$2^n$ decoder by adding to it $2^n$ input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate to provide the 1-line output. The size of a multiplexer is

specified by the number $2^n$ of its input lines and the single output line. It is then implied that it also contains n selection lines. A multiplexer is often abbreviated as MUX.

As in decoders, multiplexer Ies may have an enable input to control the operation of the unit. When the enable input is in a given binary state, the outputs are disabled, and when it is in the other state (the enable state), the circuit functions as a normal multiplexer. The enable input (sometimes called strobe) can be used to expand two or more multiplexerIes to a digital multiplexer with a larger number of inputs.




Function table

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

Fig 2 to 1 line multiplexer

As shown in the function table, the unit is selected when E = O. Then, if S = 0, the four A inputs have a path to the outputs. On the other hand, if S = 1, the four B inputs are selected. The outputs have all D's when E = 1, regardless of the value of S.

**Applications of Multiplexer:**

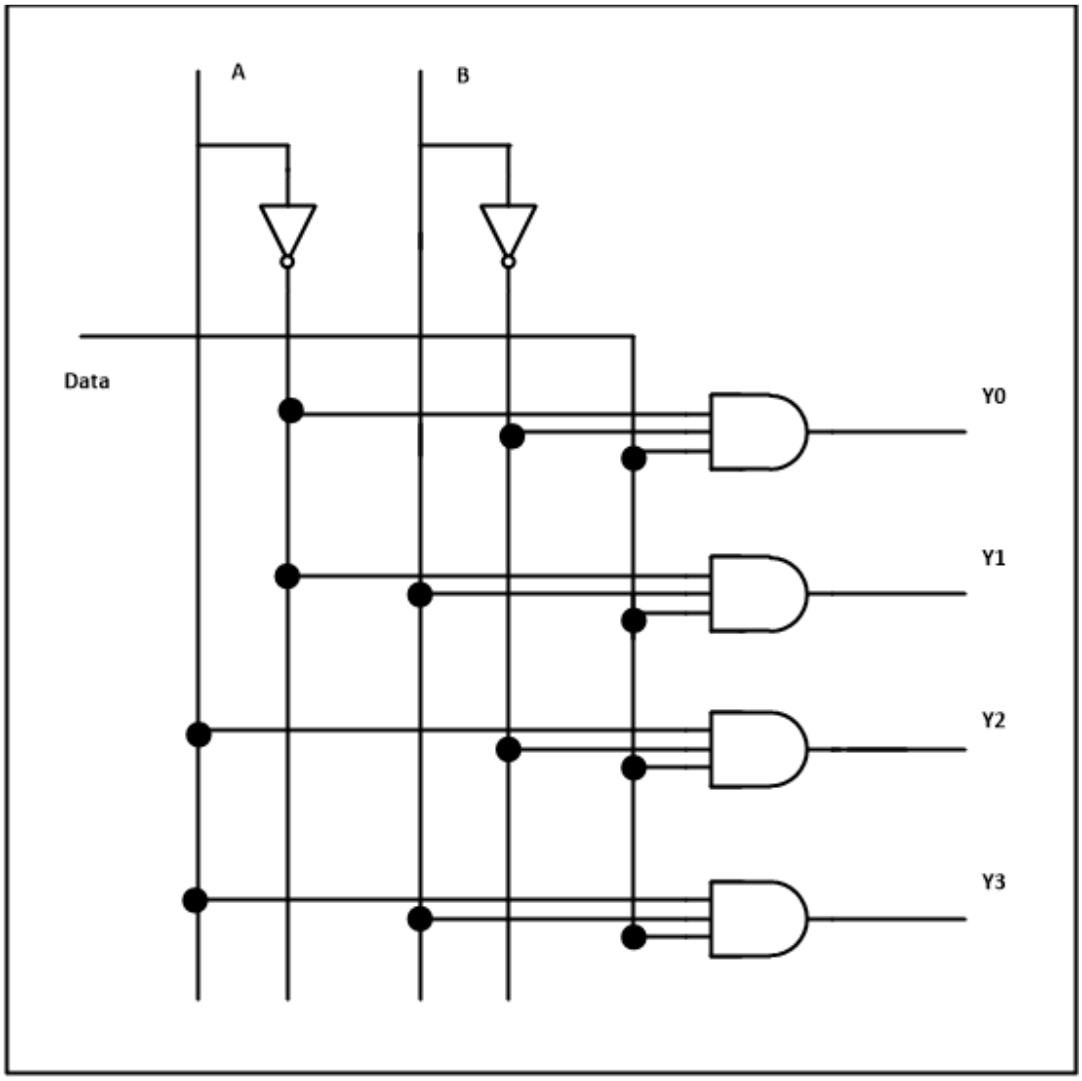
Multiplexer are used in various fields where multiple data need to be transmitted using a single line. Following are some of the applications of multiplexers -

1. **Communication system** – Communication system is a set of system that enable communication like transmission system, relay and tributary station, and communication network. The efficiency of communication system can be increased considerably using multiplexer. Multiplexer allow the process of transmitting different type of data such as audio, video at the same time using a single transmission line.
2. **Telephone network –** In telephone network, multiple audio signals are integrated on a single line for transmission with the help of multiplexers. In this way, multiple audio signals can be isolated and eventually, the desire audio signals reach the intended recipients.
3. **Computermemory** - Multiplexers are used to implement huge amount of memory into the computer, at the same time reduces the number of copper lines required to connect the memory to other parts of the computer circuit.
4. **Transmission from the computer system of a satellite** – Multiplexer can be used for the transmission of data signals from the computer system of a satellite or spacecraft to the ground system using the GPS (Global Positioning System) satellites.

# Demultiplexer:

Demultiplexer means one to many. A demultiplexer is a circuit with one input and many output. By applying control signal, we can steer any input to the output. Few types of demultiplexer are 1-to 2, 1-to-4, 1-to-8 and 1-to 16 demultiplexer.

Following figure illustrate the general idea of a demultiplexer with 1 input signal, m control signals, and n output signals.




**1 to 4 Dempultiplexer Circuit Diagram**

The input bit is labelled as Data D. This data bit is transmitted to the data bit of the output lines. This depends on the value of AB, the control input.

When AB = 01, the upper second AND gate is enabled while other AND gates are disabled. Therefore, only data bit D is transmitted to the output, giving Y1 = Data.

If D is low, Y1 is low. IF D is high,Y1 is high. The value of Y1 depends upon the value of D. All other outputs are in low state.

If the control input is changed to AB = 10, all the gates are disabled except the third AND gate from the top. Then, D is transmitted only to the Y2 output, and Y2 = Data.

Example of 1-to-16 demultiplexer is IC 74154 it has 1 input bit, 4 control bits and 16 output bit.

**Applications of Demultiplexer:**

1. Demultiplexer is used to connect a single source to multiple destinations. The main application area of demultiplexer is communication system where multiplexer are used. Most of the communication system are bidirectional i.e. they function in both ways (transmitting and receiving signals). Hence, for most of the applications, the multiplexer and demultiplexer work in sync. Demultiplexer are also used for reconstruction of parallel data and ALU circuits.
2. **Communication System** - Communication system use multiplexer to carry multiple data like audio, video and other form of data using a single line for transmission. This process make the transmission easier. The demultiplexer receive the output signals of the multiplexer and converts them back to the original form of the data at the receiving end. The multiplexer and demultiplexer work together to carry out the process of transmission and reception of data in communication system.
3. **ALU (Arithmetic Logic Unit)** – In an ALU circuit, the output of ALU can be stored in multiple registers or storage units with the help of demultiplexer. The output of ALU is fed as the data input to the demultiplexer. Each output of demultiplexer is connected to multiple register which can be stored in the registers.
4. **Serial to parallelconverter** - A serial to parallel converter is used for reconstructing parallel data from incoming serial data stream. In this technique, serial data from the incoming serial data stream is given as data input to the demultiplexer at the regular intervals. A counter is attach to the control input of the demultiplexer. This counter directs the data signal to the output of the demultiplexer where these data signals are stored. When all data signals have been stored, the output of the demultiplexer can be retrieved and read out in parallel.

## The Digital Comparator

Another common and very useful combinational logic circuit is that of the **Digital Comparator** circuit. Digital or Binary Comparators are made up from standard AND, NOR and NOT gates that compare the digital signals present at their input terminals and produce an output depending upon the condition of those inputs.

For example, along with being able to add and subtract binary numbers we need to be able to compare them and determine whether the value of input A is greater than, smaller than or equal to the value at input B etc. The digital comparator accomplishes this using several logic gates that operate on the principles of Boolean Algebra. There are two main types of **Digital Comparator** available and these are.

- 1. Identity Comparator – an *Identity Comparator* is a digital comparator that has only one output terminal for when A = B either “HIGH” A = B = 1 or “LOW” A = B = 0
- 2. Magnitude Comparator – a *Magnitude Comparator* is a type of digital comparator that has three output terminals, one each for equality, A = B greater than, A > B and less than A < B

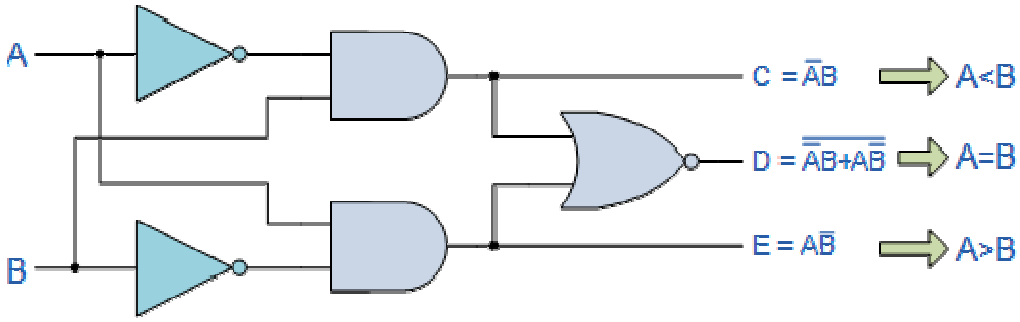
The purpose of a **Digital Comparator** is to compare a set of variables or unknown numbers, for example A (A1, A2, A3, …. An, etc) against that of a constant or unknown value such as B (B1, B2, B3, ….Bn, etc) and produce an output condition or flag depending upon the result of the comparison. For example, a magnitude comparator of two 1-bits, (A and B) inputs would produce the following three output conditions when compared to each other.

$$A > B, \quad A = B, \quad A < B$$

Which means: A is greater than B, A is equal to B, and A is less than B

This is useful if we want to compare two variables and want to produce an output when any of the above three conditions are achieved. For example, produce an output from a counter when a certain count number is reached. Consider the simple 1-bit comparator below.

### 1-bit Digital Comparator




Then the operation of a 1-bit digital comparator is given in the following Truth Table.

### Digital Comparator Truth Table

| Inputs | | Outputs | | |
|---|---|---|---|---|
| B | A | A > B | A = B | A < B |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

You may notice two distinct features about the comparator from the above truth table. Firstly, the circuit does not distinguish between either two "0" or two "1"'s as an output A = B is produced when they are both equal, either A = B = "0" or A = B = "1". Secondly, the output condition for A = B resembles that of a commonly available logic gate, the Exclusive-NOR or Ex-NOR function (equivalence) on each of the n-bits giving: $Q = A \oplus B$

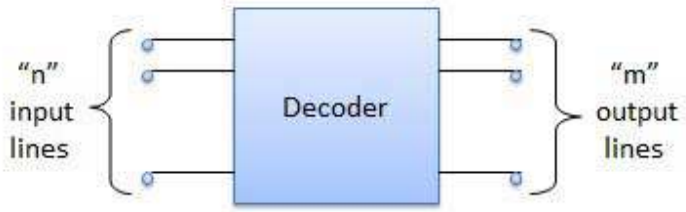
Digital comparators actually use Exclusive-NOR gates within their design for comparing their respective pairs of bits. When we are comparing two binary or BCD values or variables against each other, we are comparing the "magnitude" of these values, a logic "0" against a logic "1" which is where the term **Magnitude Comparator** comes from.

As well as comparing individual bits, we can design larger bit comparators by cascading together n of these and produce an-bit comparator just as we did for the n-bit adder in the previous tutorial. Multi-bit comparators can be constructed to compare whole binary or BCD words to produce an output if one word is larger, equal to or less than the other.

## Decoder

A decoder is a combinational circuit. It has n input and to a maximum m = 2n outputs. Decoder is identical to a demultiplexer without any data input. It performs operations which are exactly opposite to those of an encoder.

### Block diagram



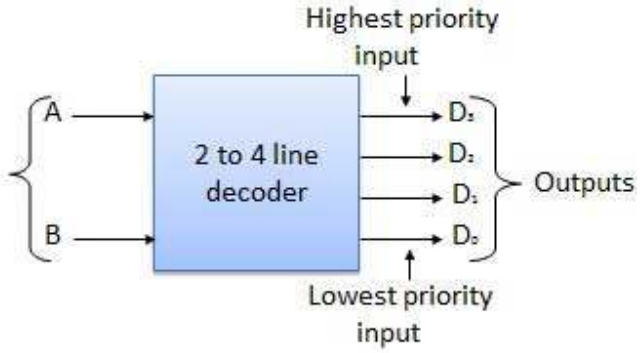

Examples of Decoders are following.

- Code converters
- BCD to seven segment decoders
- Nixie tube decoders
- Relay actuator

## 2 to 4 Line Decoder

The block diagram of 2 to 4 line decoder is shown in the fig. A and B are the two inputs where D through D are the four outputs. Truth table explains the operations of a decoder. It shows that each output is 1 for only a specific combination of inputs.

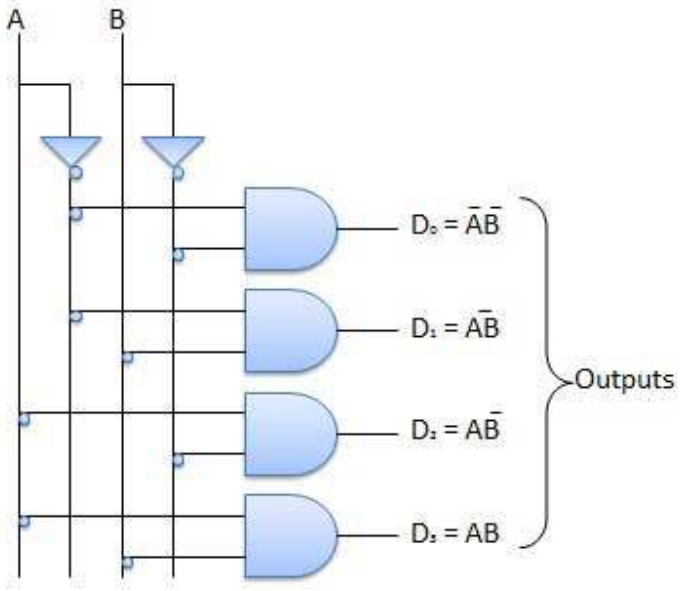
## Block diagram




## Truth Table

| Inputs | | Output | | | |
|---|---|---|---|---|---|
| A | B | $D_0$ | $D_1$ | $D_2$ | $D_3$ |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

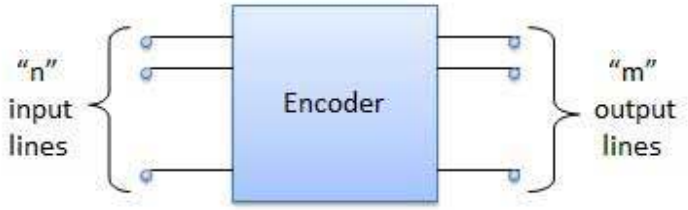
## Logic Circuit



## Encoder

Encoder is a combinational circuit which is designed to perform the inverse operation of the decoder. An encoder has n number of input lines and m number of output lines. An encoder produces an m bit binary code corresponding to the digital input number. The encoder accepts an n input digital word and converts it into an m bit another digital word.

### Block diagram



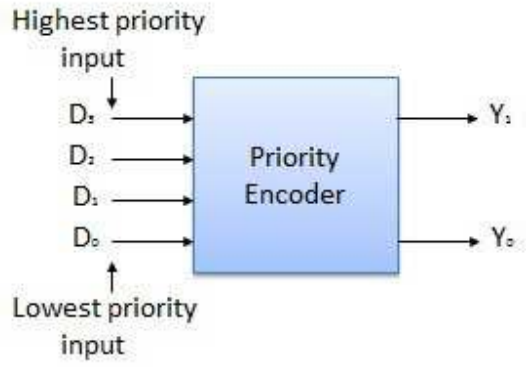

Examples of Encoders are following.

- Priority encoders
- Decimal to BCD encoder
- Octal to binary encoder
- Hexadecimal to binary encoder

## Priority Encoder

This is a special type of encoder. Priority is given to the input lines. If two or more input line are 1 at the same time, then the input line with highest priority will be considered. There are four input $D_0$, $D_1$, $D_2$, $D_3$ and two output $Y_0$, $Y_1$. Out of the four input $D_3$ has the highest priority and $D_0$ has the lowest priority. That means if $D_3 = 1$ then $Y_1Y_1 = 11$ irrespective of the other inputs. Similarly if $D_3 = 0$ and $D_2 = 1$ then $Y_1$ $Y_0 = 10$ irrespective of the other inputs.

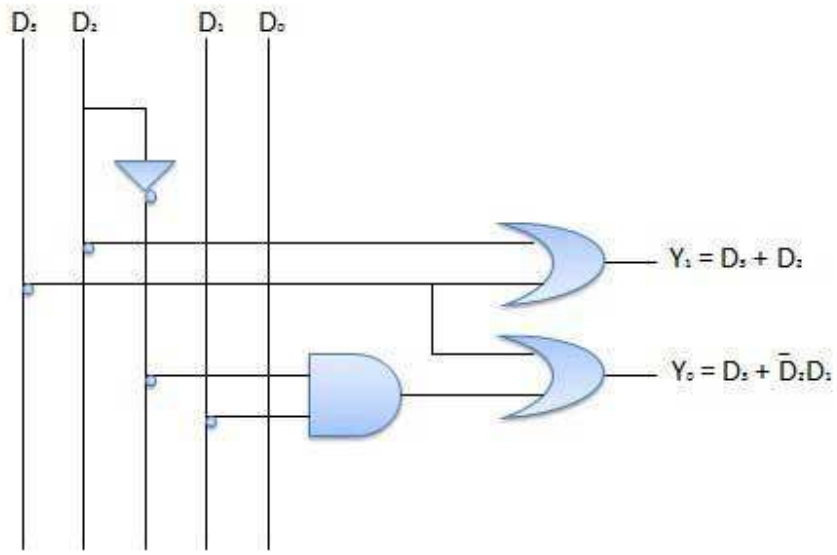
### Block diagram



### Truth Table

| Highest | Inputs | | Lowest | Outputs | |
|---|---|---|---|---|---|
| $D_3$ | $D_2$ | $D_1$ | $D_0$ | $Y_0$ | $Y_1$ |
| 0 | 0 | 0 | 0 | x | x |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | x | 0 | 1 |
| 0 | 1 | x | x | 1 | 0 |
| 1 | x | x | x | 1 | 1 |

### Logic Circuit




## PARITY GENERATOR AND CHECKER

•Parity is a very useful tool in information processing in digital computers to indicate any presence of error in bit information.

•External noise and loss of signal strength cause loss of data bitinformation while transporting data from one device to other device,located inside the computer or externally.

•To indicate any occurrence of error, an extra bit is included withthe message according to the total number of 1s in a set of data,which is called parity.

•If the extra bit is considered 0 if the total number of 1s is even and 1for odd quantities of 1s in a set of data, then it is called even parity.

•On the other hand, if the extra bit is 1 for even quantities of 1s and0 for an odd number of 1s, then it is called odd parity

## Parity Generator:-

A parity generator is a combination logic system to generatethe parity bit at the transmitting side

| *Four bit Message* $D_3D_2D_1D_0$ | *Even Parity* ( $P_e$ ) | *Odd Parity* ( $P_o$ ) |
|---|---|---|
| 0000 | 0 | 1 |
| 0001 | 1 | 0 |
| 0010 | 1 | 0 |
| 0011 | 0 | 1 |
| 0100 | 1 | 0 |
| 0101 | 0 | 1 |
| 0110 | 0 | 1 |
| 0111 | 1 | 0 |
| 1000 | 1 | 0 |
| 1001 | 0 | 1 |
| 1010 | 0 | 1 |
| 1011 | 1 | 0 |
| 1100 | 0 | 1 |
| 1101 | 1 | 0 |
| 1110 | 1 | 0 |
| 1111 | 0 | 1 |

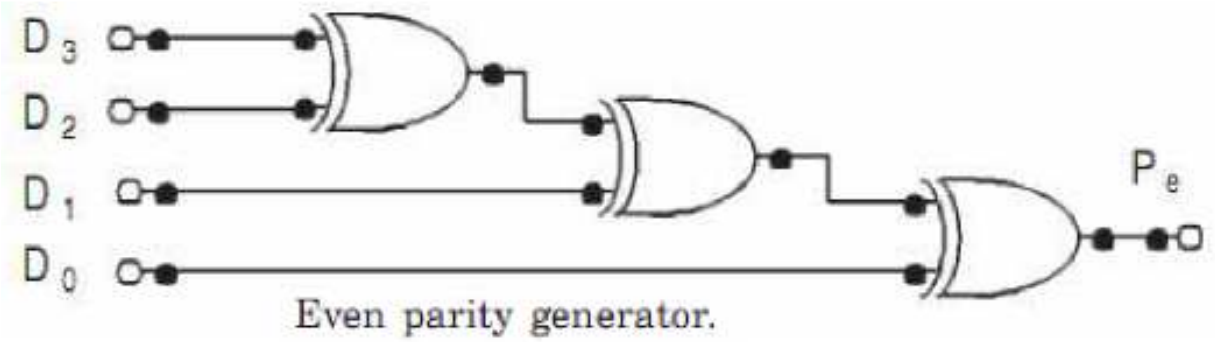
If the message bit combination is designated as $D_3D_2D_1D_0$,and$P_e$, $P_o$are the even and odd parity respectively, then it isobvious from the table that the Boolean expressions of even

parity and odd parity are

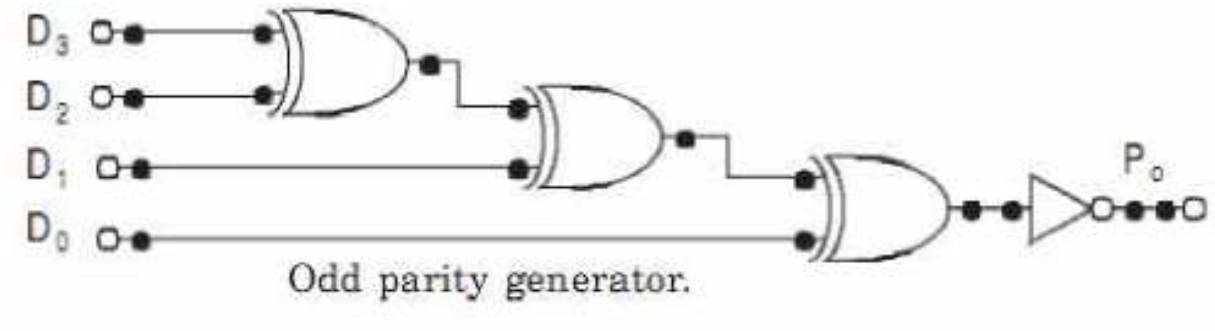
Pe= D3 ⊕ D2 ⊕ D1 ⊕D0

and

Po= (D3 ⊕ D2 ⊕ D1 ⊕ D0)'



Even parity generator.



Odd parity generator.

The above illustration is given for a message with four bits of information. However, the logic diagrams can be expanded with more XOR gates for any number of bits.

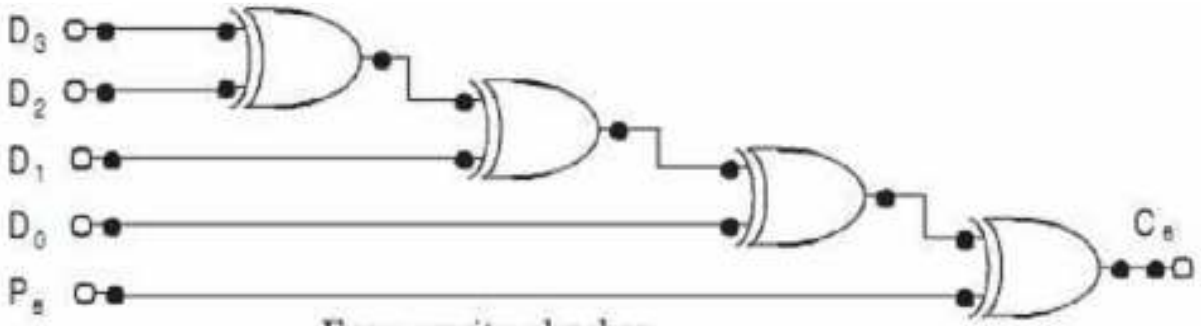
## Parity Checker

•The message bits with the parity bit are transmitted to their destination, where they are applied to a parity checker circuit.

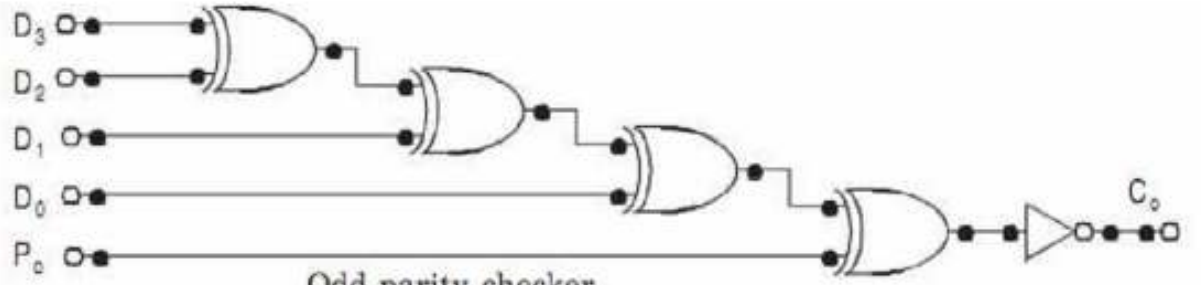
•The circuit that checks the parity at the receiver side is called the parity checker. The parity checker circuit produces a check bit and is very similar to the parity generator circuit.

•If the check bit is 1, then it is assumed that the received dataisincorrect. The check bit will be 0 if the received data is correct.

Note that the check bit is 0 for all the bit combinations of correct data. For Incorrect data the parity check bit will be another logic value




Even parity checker.




Odd parity checker.

| *4-bit message* $D_3D_2D_1D_0$ | *Even Parity* ($P_e$) | *Even Parity Checker* ($C_e$) |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 0 |
| 0010 | 1 | 0 |
| 0011 | 0 | 0 |
| 0100 | 1 | 0 |
| 0101 | 0 | 0 |
| 0110 | 0 | 0 |
| 0111 | 1 | 0 |
| 1000 | 1 | 0 |
| 1001 | 0 | 0 |
| 1010 | 0 | 0 |
| 1011 | 1 | 0 |
| 1100 | 0 | 0 |
| 1101 | 1 | 0 |
| 1110 | 1 | 0 |
| 1111 | 0 | 0 |

Even parity checker.

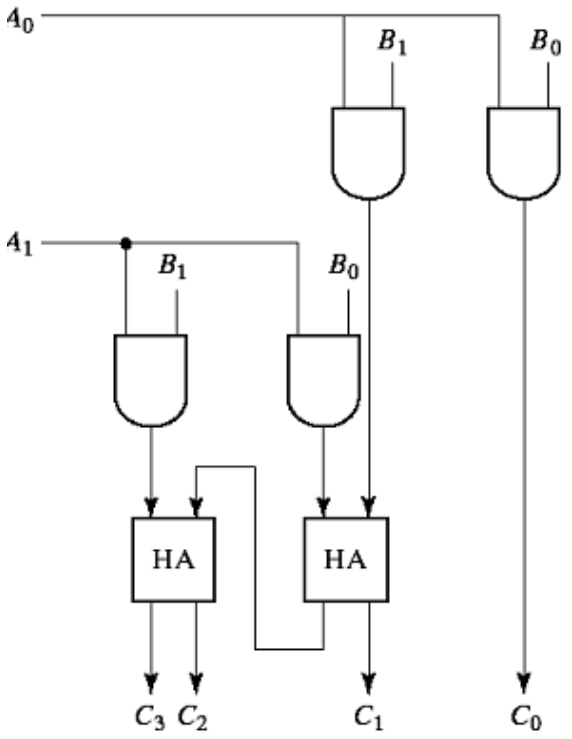
| *4-bit message* $D_3D_2D_1D_0$ | *Odd Parity* ($P_o$) | *Odd Parity Checker* ($C_o$) |
|---|---|---|
| 0000 | 1 | 0 |
| 0001 | 0 | 0 |
| 0010 | 0 | 0 |
| 0011 | 1 | 0 |
| 0100 | 0 | 0 |
| 0101 | 1 | 0 |
| 0110 | 1 | 0 |
| 0111 | 0 | 0 |
| 1000 | 0 | 0 |
| 1001 | 1 | 0 |
| 1010 | 1 | 0 |
| 1011 | 0 | 0 |
| 1100 | 1 | 0 |
| 1101 | 0 | 0 |
| 1110 | 0 | 0 |
| 1111 | 1 | 0 |

Odd parity checker.

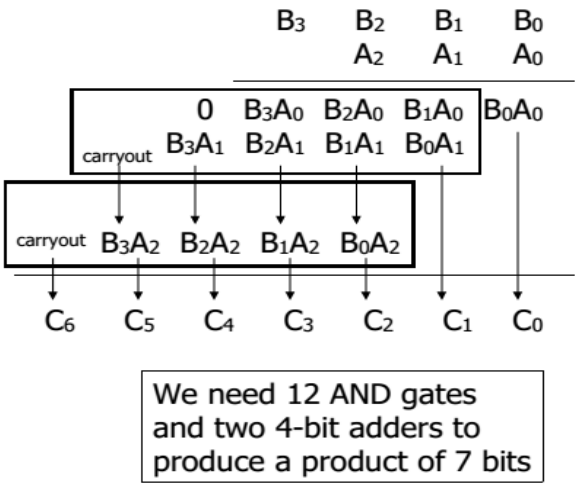
# Binary Multiplier

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

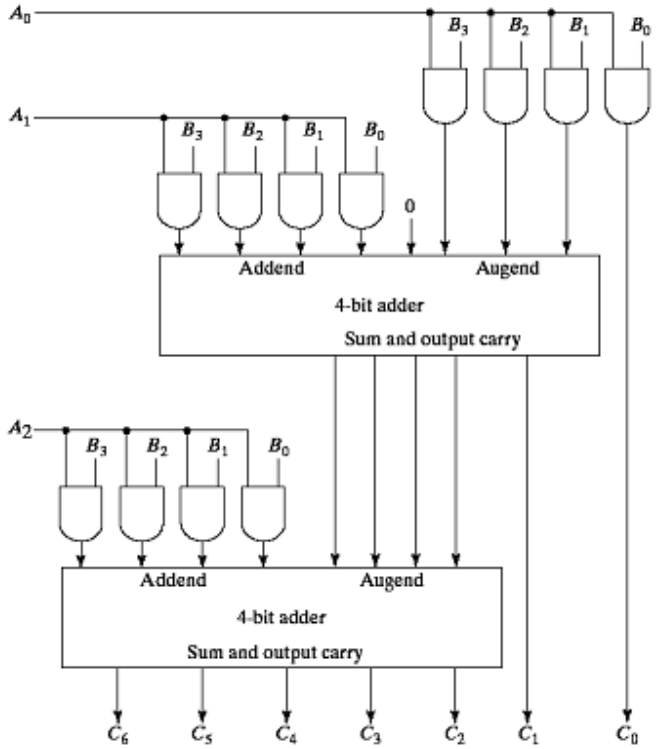
Circuit diagram



# 4-Bit By 3-Bit Binary Multiplier




Circuit diagram

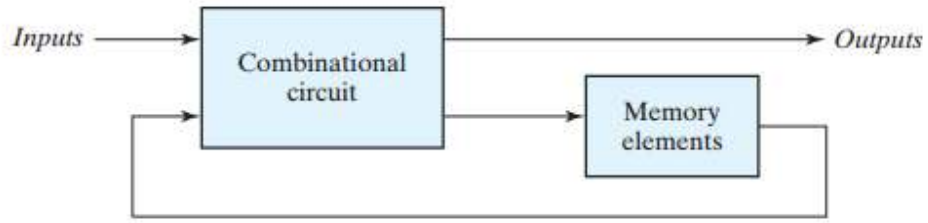


# Module-3

## SEQUENTIAL LOGIC CIRCUITS

Till now we studied the logic circuits whose outputs at any instant of time depend only on the input signals present at that time are known as combinational circuits. Moreover, in a combinational circuit, the output appears immediately for a change in input, except for the propagation delay through circuit gates.

On the other hand, the logic circuits whose outputs at any instant of time depend on the present inputs as well as on the past outputs are called sequential circuits. In sequential circuits, the output signals are fed back to the input side. A block diagram of a sequential circuit is shown in Figure below:-




It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the ***state*** of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state in the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, **a sequential circuit is specified by a time sequence of inputs, outputs, and internal states.**

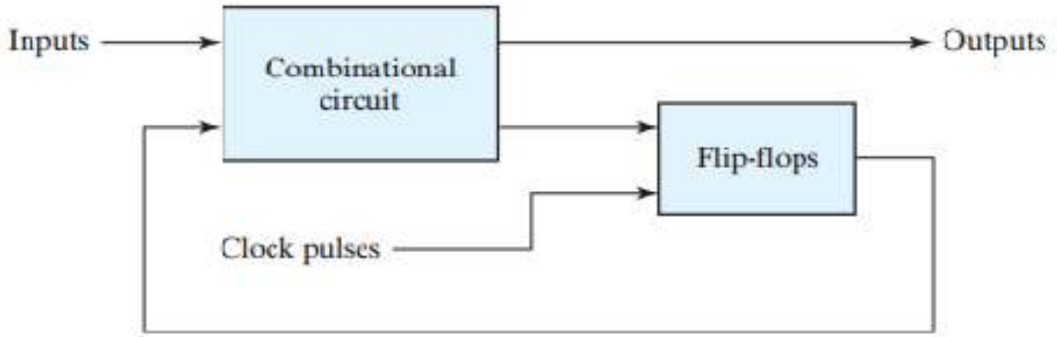
There are two types of sequential circuits, and their classification is a function of the timing of their signals.

### *Asynchronous sequential circuit*:

A sequential circuit whose behavior depends upon the sequence in which the input signals change is referred to as an ***asynchronous sequential circuit***. The output will be affected whenever the input changes. The commonly used memory elements in these circuits are time-delay devices. There is no need to wait for a clock pulse. Therefore, in general, asynchronous circuits are faster than synchronous sequential circuits. However, in an asynchronous circuit, events are allowed to occur without any synchronization. And in such a case, the system becomes unstable. Since the designs of asynchronous circuits are more tedious and difficult, their uses are rather limited. The memory elements used in sequential circuits are flip-flops which are capable of storing binary information.

### *Synchronous sequential circuit:*

A sequential circuit whose behavior can be defined from the knowledge of its signal at discrete instants of time is referred to as a ***synchronous sequential circuit***. In these systems, the memory elements are affected only at discrete instants of time. The synchronization is achieved by a timing device known as a system clock, which generates a periodic train of lock pulses. The outputs are affected only with the application of a clock pulse.




(a) Block diagram

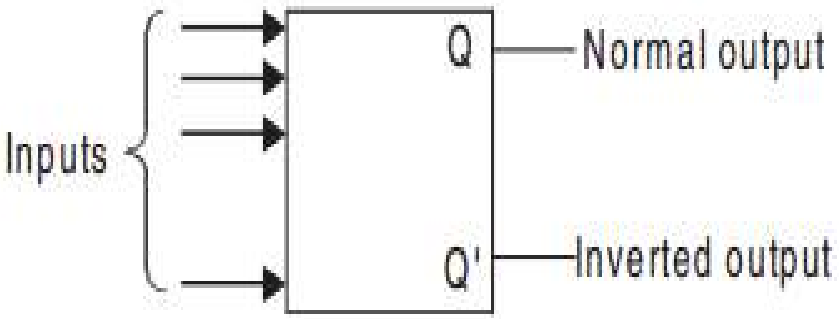
(b) Timing diagram of clock pulses

Synchronous clocked sequential circuit

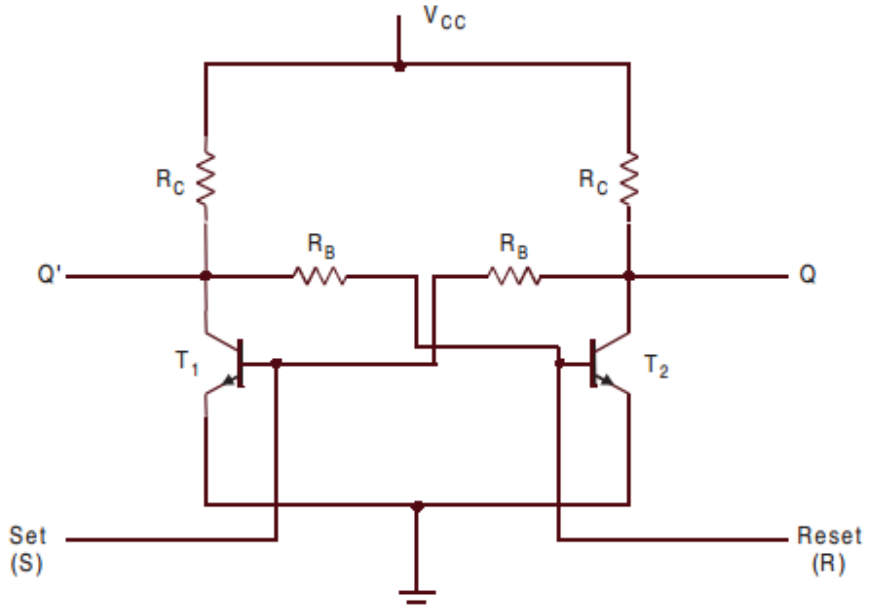
The storage elements (memory) used in clocked sequential circuits are called ***flipflops***

## FLIPFLOPS

The basic 1-bit digital memory circuit is known as a flip-flop. It can have only two states, either the 1 state or the 0 state. A flip-flop is also known as a bistable multivibrator. Flip-flops can be obtained by using NAND or NOR gates. The general block diagram representation of a flip-flop is shown in Figure below. It has one or more inputs and two outputs. The two outputs are complementary to each other. If Q is 1 *i.e.*, Set, then Q' is 0; if Q is 0 *i.e.*, Reset, then Q' is 1. That means Q and Q' cannot be at the same state simultaneously. If it happens by any chance, it violates the definition of a flip-flop and hence is called an *undefined* condition. Normally, the state of Q is called the *state* of the flip-flop, whereas the state of Q' is called the *complementary state* of the flip-flop. When the output Q is either 1 or 0, it remains in that state unless one or more inputs are excited to effect a change in the output. Since the output of the flip-flop remains in the same state until the trigger pulse is applied to change the state, it can be regarded as a memory device to store one binary bit. The block diagram of a flip-flop is given below:-




The Bistable multivibrator circuit of a flip-flop is given below:-



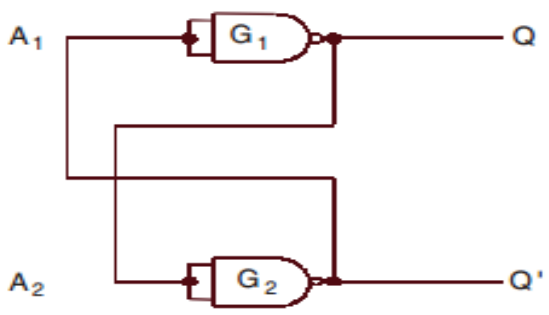

.

From the circuit shown in above, the multivibrator is basically two cross-coupled inverting amplifiers, consist of two transistors and four resistors. Obviously, if transistor $T_1$ is initially turned ON (saturated) by applying a positive signal through the Set input at its base, its collector will be at $V_{CE(sat)}$ (0.2 to 0.4 V). The collector of $T_1$ is connected to the base of $T_2$, which cannot turn $T_2$ On. Hence, $T_2$ remains OFF (cut off). Therefore, the voltage at the collector of $T_2$ tries to reach $V_{CC}$. This action only enhances the initial positive signal applied to the base of $T_1$. Now if the initial signal at the Set input is removed, the circuit will maintain $T_1$ in the ON state and $T_2$ in the OFF state indefinitely, *i.e.*, Q = 1 & Q' = 0. In this condition the bistable multivibrator is said to be in the **Set state**. A positive signal applied to the Reset input at the base of $T_2$ turns it ON. As we have discussed earlier, in the same sequence $T_2$ turns ON & $T_1$ turns OFF, resulting in a second stable state *i.e.* Q = 0 & Q' = 1. In this condition the bistable multivibrator is said to be in the ***Reset* state.**

## LATCHES

The basic difference between a latch & flip-flop is, 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.




We consider the fundamental circuit shown in Fig.(last page). It consists of two inverters $G_1$ and $G_2$ (NAND gates are used as inverters). The output of $G_1$ is connected to the input of $G_2$ ($A_2$) and the output of $G_2$ is connected to the input of $G_1$ ($A_1$).

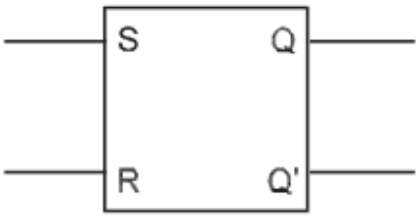
Let us assume the output of $G_1$ to be Q = 0, which is also the input of $G_2$ ($A_2$ = 0). So, the output of $G_2$ will be Q'= 1, which makes $A_1$ = 1 and consequently Q = 0 which is according to our assumption. Similarly, we can demonstrate that if Q = 1, then Q' = 0 and this is also consistent with the circuit connections. Hence we see that Q and Q' are always complementary. And if the circuit is in 1 state, it continues to remain in this state and vice versa is also true. Since this information is locked or latched in this circuit, therefore, this circuit is also referred to as a ***latch***. In this circuit there is no way to enter the desired digital information to be stored in it. To make that possible we have to modify the circuit by replacing the inverters by NAND gates and then it becomes a flip-flop.

# TYPES OF FLIP-FLOPS

There are different types of flip-flops depending on how their inputs and clock pulses cause transition between two states. We will discuss four different types of flip-flops in this chapter, *viz*., S-R, D, J-K, and T. Basically D, J-K, and T are three different modifications of the S-R flip-flop.

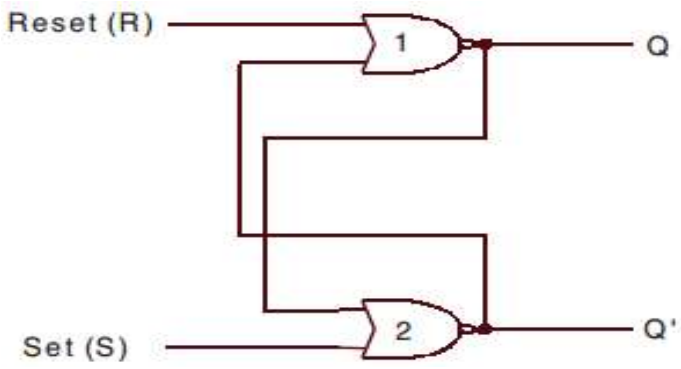
## S-R (Set-Reset) Flip-flop

An S-R flip-flop has two inputs named Set (S) and Reset (R), and two outputs Q and Q'. The outputs are complement of each other, *i.e.*, if one of the outputs is 0 then the other should be 1. This can be implemented using NAND or NOR gates. The block diagram of an S-R flip-flop is shown in Figure below:-




## S-R Flip-flop Based on NOR Gates

An S-R flip-flop can be constructed with NOR gates at ease by connecting the NOR gates back to back as shown in Figure below. The cross-coupled connections from the output of gate 1 to the input of gate 2 constitute a feedback path. This circuit is not clocked and is classified as an asynchronous sequential circuit. The truth table for the S-R flip-flop based on a NOR gate is shown in the table below



| Inputs | | Outputs | | Action |
|---|---|---|---|---|
| $S$ | $R$ | $Q_{n+1}$ | $Q'_{n+1}$ | |
| 0 | 0 | $Q_n$ | $Q'_n$ | No change |
| 0 | 1 | 0 | 1 | Reset |
| 1 | 0 | 1 | 0 | Set |
| 1 | 1 | 0 | 0 | Forbidden (Undefined) |
| 0 | 0 | – | – | Indeterminate |

To analyze the circuit of S-R Flip-flop Based on NOR Gates, we have to consider the fact that the output of a NOR gate is 0 if any of the inputs are 1, irrespective of the other input. The output is 1 only if all of the inputs are 0. The outputs for all the possible conditions as shown in the above table are described as follows.

**Case 1.** For S = 0 and R = 0, the flip-flop remains in its present state ($Q_n$). It means that the next state of the flip-flop does not change, *i.e.*, $Q_{n+1} = 0$ if $Q_n = 0$ and vice versa. First let us assume that $Qn = 1$ and $Q'_n = 0$.Thus the inputs of NOR gate 2 are 1 and 0, and therefore its output $Q'n+1 = 0$. This output $Q'_{n+1} = 0$ is fed back as the input of NOR gate1, thereby producing a 1 at the output, as both of the inputs of NOR gate 1 are 0 and 0; so $Q_{n+1} = 1$ as originally assumed. Now let us assume the opposite case, *i.e.,* $Q_n = 0$ and $Q'_n = 1$. Thus the inputs of NOR gate 1 are 1 and 0, and therefore its output $Q'_{n+1} = 0$. This output $Q_{n+1} = 0 = 0$ is fed back as the input of NOR gate 2, thereby producing a 1 at the output, as both of the inputs of NOR gate 2 are 0 and 0; so $Q'_{n+1} = 1$ as originally assumed. Thus we find that the condition S = 0 and R = 0 do not affect the outputs of the flip-flop, which means this is the memory condition of the S-R flip-flop.

**Case 2.** The second input condition is S = 0 and R = 1. The 1 at R input forces the output of NOR gate 1 to be 0 (*i.e.*, $Q_{n+1} = 0$). Hence both the inputs of NOR gate 2 are 0 and 0 and so its output $Q'_{n+1} = 1$. Thus the condition S = 0 and R = 1 will always reset the flip-flop to 0. Now if the R returns to 0 with S = 0, the flip-flop will remain in the same state.

**Case 3.** The third input condition is S = 1 and R = 0. The 1 at S input forces the output of NOR gate 2 to be 0 (*i.e.*, $Q'_{n+1} = 0$). Hence both the inputs of NOR gate 1 are 0 and 0 and so its output $Q_{n+1} = 1$. Thus the condition S = 1 and R = 0 will always set the flip-flop to 1. Now if the S returns to 0 with R = 0, the flip-flop will remain in the same state.

**Case 4.** The fourth input condition is S = 1 and R = 1. The 1 at R input and 1 at S input forces the output of both NOR gate 1 and NOR gate 2 to be 0. Hence both the outputs of NOR gate 1 and NOR gate 2 are 0 and 0; *i.e.*, $Q_{n+1} = 0$ and $Q'_{n+1} = 0$. Hence this condition S = 1 and R = 1 violates the fact that the outputs of a flip-flop will always be the complement of each other. Since the condition violates the basic definition of flip-flop, it is called the ***undefined*** condition. Generally this condition must be avoided by making sure that 1s are not applied simultaneously to both of the inputs.

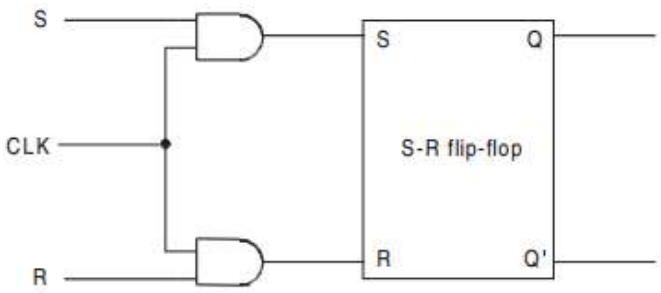
**Case 5.** If case 4 arises at all, then S and R both return to 0 and 0 simultaneously, and then any one of the NOR gates acts faster than the other and assumes the state. For example, if NOR gate 1 is faster than NOR gate 2,

then $Q_{n+1}$ will become 1 and this will make $Q'_{n+1} = 0$. Similarly, if NOR gate 2 is faster than NOR gate 1, then $Q'_{n+1}$ will become 1 and this will make $Q_{n+1} = 0$. Hence, this condition is determined by the flip-flop itself. Since this condition cannot be controlled and predicted it is called the ***indeterminate*** condition.

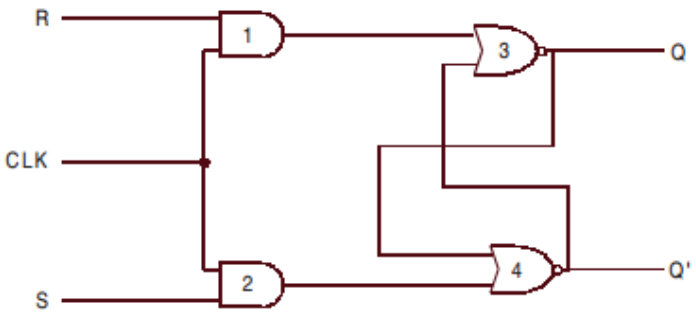
Similarly we can analyze the case of S'-R' Flip-flop Based on NAND Gates (assignment for the students).

## CLOCKED S-R FLIP-FLOP

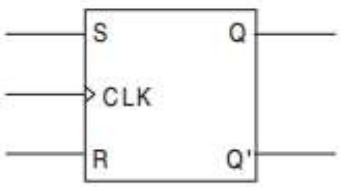
Generally, synchronous circuits change their states only when clock pulses are present. The operation of the basic flip-flop can be modified by including an additional input to control the behavior of the circuit. Such a circuit is shown below:-




The circuit shown above consists of two AND gates. The clock input is connected to both of the AND gates, resulting in LOW outputs when the clock input is LOW. In this situation the changes in S and R inputs will not affect the state (Q) of the flip-flop. On the other hand, if the clock input is HIGH, the changes in S and R will be passed over by the AND gates and they will cause changes in the output (Q) of the flip-flop. This way, any information, either 1 or 0, can be stored in the flip-flop by applying a HIGH clock input and be retained for any desired period of time by applying a LOW at the clock input. This type of flip-flop is called a ***clocked S-R flip-flop***. Such a clocked S-R flip-flop made up of two AND gates and two NOR gates is shown in Figure below:-




The logic symbol of the S-R flip-flop is shown below. It has three inputs: S, R, and CLK. The CLK input is marked with a small triangle. The triangle is a symbol that denotes the fact that the circuit responds to an edge or transition at CLK input.



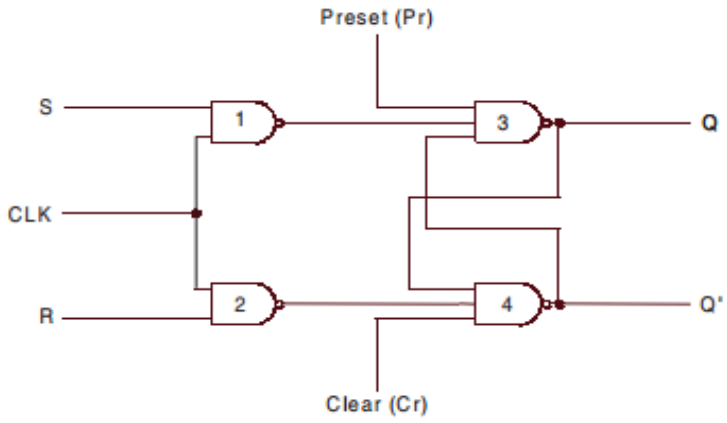
Assuming that the inputs do not change during the presence of the clock pulse, we can express the working of the S-R flip-flop in the form of the truth table shown here. Here, $S_n$ and $R_n$ denote the inputs and $Q_n$ denotes the output during the bit time $n$. $Q_{n+1}$ denotes the output after the pulse passes *i.e.* in the bit time $n + 1$.

| *Inputs* | | *Output* |
|---|---|---|
| $S_n$ | $R_n$ | $Q_{n+1}$ |
| 0 | 0 | $Q_n$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | – |

**Case 1.** If $S_n = R_n = 0$, and the clock pulse is not applied, the output of the fl ip-fl op remains in the present state. Even if $S_n = R_n = 0$, and the clock pulse is applied, the output at the end of the clock pulse is the same as the output before the clock pulse, *i.e.*, $Q_{n+1} = Q_n$. The first row of the table indicates that situation.
**Case 2.** For $S_n = 0$ and $R_n = 1$, if the clock pulse is applied (*i.e.* CLK = 1), the output of NAND gate 1 becomes 1; whereas the output of NAND gate 2 will be 0. Now a 0 at the input of NAND gate 4 forces the output to be 1 *i.e.* Q' = 1. This 1 goes to the input of NAND gate 3 to make both the inputs of NAND gate 3 as 1, which forces the output of NAND gate 3 to be 0, *i.e.*, Q = 0.
**Case 3.** For $S_n = 1$ and $R_n = 0$, if the clock pulse is applied (*i.e.*, CLK = 1), the output of NAND gate 2 becomes 1; whereas the output of NAND gate 1 will be 0. Now a 0 at the input of NAND gate 3 forces the output to be 1, *i.e.*, Q = 1. This 1 goes to the input of NAND gate 4 to make both the inputs of NAND gate 4 as 1, which forces the output of NAND gate 4 to be 0, *i.e.*, Q' = 0.
**Case 4.** For $S_n = 1$ and $R_n = 1$, if the clock pulse is applied (*i.e.* CLK = 1), the outputs of both NAND gate 2 and NAND gate 1 becomes 0. Now a 0 at the input of both NAND gate 3 and NAND gate 4 forces the outputs of both the gates to be 1, *i.e.*, Q = 1 and Q' = 1. When the CLK input goes back to 0 (while S and R remain at 1), it is not possible to determine the next state, as it depends on whether the output of gate 1 or gate 2 goes to 1 first.

## Preset and Clear

Till now the flip-flops we discussed there when the power is switched on, the state of the circuit is uncertain. It may come to reset (Q = 0) or set (Q = 1) state. But in many applications it is required to initially set or reset the flip-flop., *i.e.*, the initial state of the flip-flop is to be assigned. This is done by using the direct or asynchronous inputs. These inputs are referred to as ***preset*** **(Pr)** and ***clear*** **(Cr)** inputs. These inputs may be applied at any time between clock pulses and is not in synchronism with the clock. Such an S-R flip-flop containing preset and clear inputs is shown in Figure below.



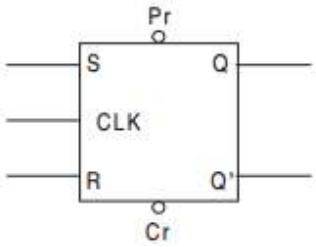

From the above Figure, we see that if Pr = Cr = 1, the circuit operates according to the table of clocked S-R flip-flop as we discussed just before.

If Pr = 1 and Cr = 0, the output of NAND gate 4 is forced to be 1, *i.e.*, Q' = 1 and the flip-flop is reset, overwriting the previous state of the flip-flop.

If Pr = 0 and Cr = 1, the output of NAND gate 3 is forced to be 1, *i.e.*, Q = 1 and the flip-flop is set, overwriting the previous state of the flip-flop. Once the state of the flip-flop is established asynchronously, the inputs Pr and Cr must be connected to logic 1 before the next clock is applied.

The condition Pr = Cr = 0 must not be applied, since this leads to an uncertain state.

The logic symbol of an S-R flip-flop with Pr and Cr inputs is shown in the side. Here, bubbles are used for Pr and Cr inputs, which indicate these are active low inputs, which means that the intended function is performed if the signal applied to Pr and Cr is LOW. The operation of the clocked S-R flip-flop is shown in the table in below. The circuit can be designed such that the asynchronous inputs override the clock, *i.e.*, the circuit can be set or reset even in the presence of the clock pulse.



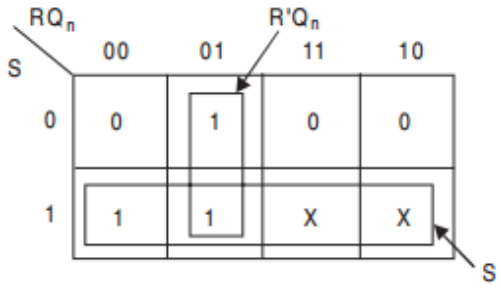

| *Inputs* | | | *Output* | *Operation* |
|---|---|---|---|---|
| *CLK* | *Cr* | *Pr* | *Q* | *performed* |
| 1 | 1 | 1 | $Q_{n+1}$ (Figure 7.3) | Normal flip-flop |
| 0 | 1 | 0 | 1 | Preset |
| 0 | 0 | 1 | 0 | Clear |
| 0 | 0 | 0 | – | Uncertain |

## Characteristic Table of an S-R Flip-flop

From the name itself it is very clear that the *characteristic table* of a flip-flop actually gives us an idea about the character, *i.e.*, the working of the flip-flop. Now, from all our above discussions, we know that the next state flip-flop output ($Q_{n+1}$) depends on the present inputs as well as the present output ($Q_n$). So in order to know the next state output of a flip-flop, we have to consider the present state output also. The characteristic table of an S-R fl ip-fl op is given in the table below. From the characteristic table we have to find out the characteristic equation of the S-R flip-flop.

| *Flip-flop inputs* | | *Present output* | *Next output* |
|---|---|---|---|
| *S* | *R* | $Q_n$ | $Q_{n+1}$ |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | X |
| 1 | 1 | 1 | X |

Now we will find out the characteristic equation of the S-R flip-flop from the characteristic table with the help of the Karnaugh map:-




From the Karnaugh map above we find the expression for $Q_{n+1}$ as

$$Q_{n+1} = S + R'Q_n$$

Along with the above equation we have to consider the fact that S and R cannot be simultaneously 0. In order to take that fact into account we have to incorporate another equation for the S-R flip-flop. The equation is given below.

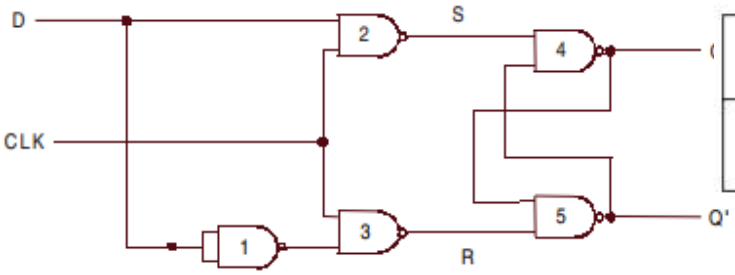
$$SR = 0$$

Hence the characteristic equations of an S-R flip-flop are

$$Q_{n+1} = S + R' Q_n$$
$$SR = 0$$

## CLOCKED D FLIP-FLOP

One way to eliminate the undesirable condition of the indeterminate state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time. This is done in the *D* latch. The D flip-flop has only one input referred to as the D (**data**) input & two outputs as usual Q and Q'. It transfers the data at the input after the delay of one clock pulse at the output Q. So in some cases the input is referred to as a delay input and the flip-flop gets the name ***delay*** (D) flip-flop. It can be easily constructed from an S-R flip-flop by simply incorporating an inverter between S and R such that the input of the inverter is at the S end & the output of the inverter is at the R end. We can get rid of the undefined condition, *i.e.*, S = R = 1 condition, of the S-R flip-flop in the D flip flop. The D flip-flop is either used as a delay device or as a latch to store one bit of binary information. The truth table of D flip-flop is given in the table below. The structure of the D flip-flop is shown in Figure below, which is being constructed using NAND gates. The same structure can be constructed using only NOR gates.



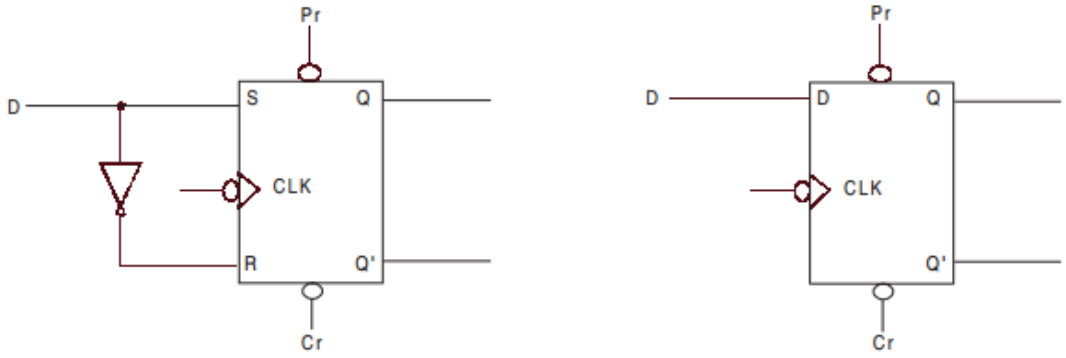

| *Input* $D_n$ | *Output* $Q_{n+1}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

**Case 1.** If the CLK input is low, the value of the D input has no effect, since the S and R inputs of the basic NAND flip-flop are kept as 1.

**Case 2.** If the CLK = 1 and D = 1, the NAND gate 1 produces 0, which forces the output of NAND gate 3 as 1. On the other hand, both the inputs of NAND gate 2 are 1, which gives the output of gate 2 as 0. Hence, output

of NAND gate 4 is forced to be 1, *i.e.*, Q = 1, whereas both the inputs of gate 5 are 1 and the output is 0, *i.e.*, Q' = 0. Hence, we find that when D = 1, after one clock pulse passes Q = 1, which means the output follows D.
**Case 3.** If the CLK = 1, and D = 0, the NAND gate 1 produces 1. Hence both the inputs of NAND gate 3 are 1, which gives the output of gate 3 as 0. On the other hand, D = 0 forces the output of NAND gate 2 to be 1. Hence the output of NAND gate 5 is forced to be 1, *i.e.*, Q' = 1, whereas both the inputs of gate 4 are 1 and the output is 0, *i.e.,* Q = 0. Hence, we find that when D = 0, after one clock pulse passes Q = 0, which means the output again follows D.

A simple way to construct a D flip-flop using an S-R flip-flop is shown in Figure below. The logic symbol of a D flip-flop is shown in Figure below. A D flip-flop is most often used in the construction of sequential circuits like registers.

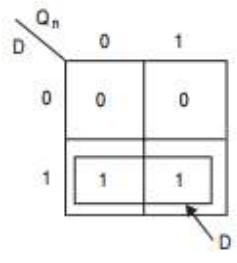



## Characteristic Table of a D Flip-flop

As we have already discussed the characteristic equation of an S-R flip-flop, we can similarly find out the characteristic equation of a D flip-flop. The characteristic table of a D flip-flop is given in the table below. From the characteristic table we have to find out the characteristic equation of the D flip-flop.

| *Flip-flop inputs* | *Present output* | *Next output* |
|---|---|---|
| *D* | $Q_n$ | $Q_{n+1}$ |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Now we will find out the characteristic equation of the D flip-flop from the characteristic table with the help of the Karnaugh map:-




Hence, the characteristic equation of a D flip-flop is

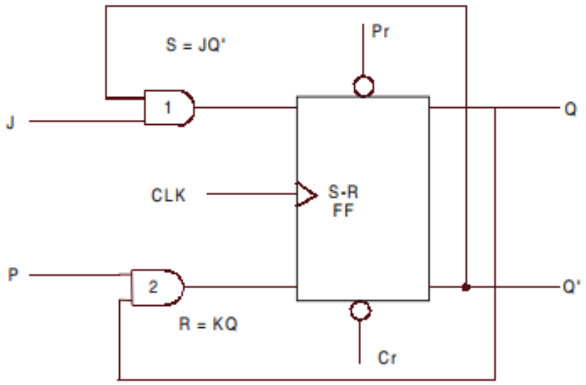
**$Q_{n+1} = D$**

# J-K FLIP-FLOP

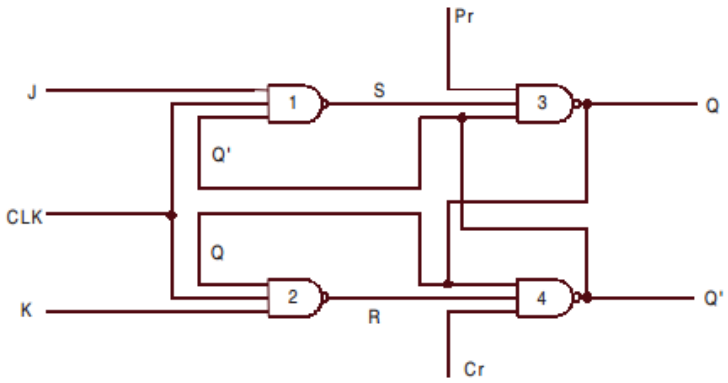
A J-K flip-flop has very similar characteristics to an S-R flip-flop. The only difference is that the undefined condition for an S-R flip-flop, *i.e.*, $S_n = R_n = 1$ condition, is also included in this case. Inputs J and K behave like inputs S and R to set and reset the flip-flop respectively. When J = K = 1, the flip-flop is said to be in a ***toggle state***, which means the output switches to its complementary state every time a clock passes.

The data inputs are J and K, which are ANDed with Q' and Q respectively to obtain the inputs for S and R respectively. A J-K flip-flop thus obtained is shown in Figure below.

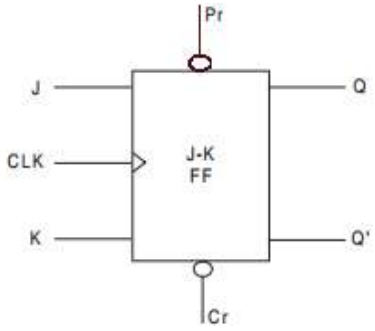
An S-R flip-flop converted into a J-K flip-flop:-




A J-K flip-flop using NAND gates:-

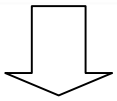



Logic symbol of a J-K flip-flop:-



The TRUTH table for JK flip-flop is:-

| *Data inputs* | | *Outputs* | | *Inputs to S-R FF* | | *Output* |
|---|---|---|---|---|---|---|
| $J_n$ | $K_n$ | $Q_n$ | $Q'_n$ | $S_n$ | $R_n$ | $Q_{n+1}$ |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 |



| *Inputs* | | *Output* |
|---|---|---|
| $J_n$ | $K_n$ | $Q_{n+1}$ |
| 0 | 0 | $Q_n$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $Q'_n$ |

**Case 1.** When the clock is applied and J = 0, whatever the value of $Q'_n$ (0 or 1), the output of NAND gate 1 is 1. Similarly, when K = 0, whatever the value of $Q_n$ (0 or 1), the output of gate 2 is also 1. Therefore, when J = 0 and K = 0, the inputs to the basic flip-flop are S = 1 and R = 1. This condition forces the flip-flop to remain in the same state.

**Case 2.** When the clock is applied and J = 0 and K = 1 & the previous state of the flip-flop is reset (*i.e.*, $Q_n = 0$ and $Q'_n = 1$), then S = 1 and R = 1. Since S = 1 and R = 1, the basic flip-flop does not alter the state and remains in the reset state. But if the flip-flop is in set condition (*i.e.*, $Q_n = 1$ & $Q'_n = 0$), then S = 1 and R = 0. Since S = 1 and R = 0, the basic flip-flop changes its state and resets.

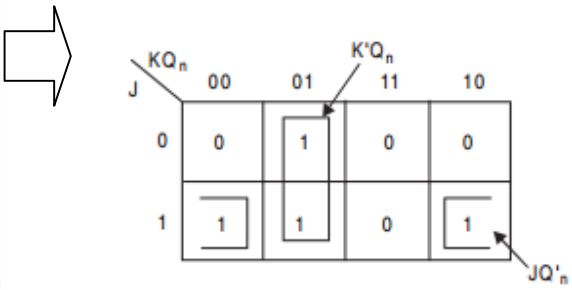
**Case 3.** When the clock is applied and J = 1 and K = 0 and the previous state of the flip-flop is reset (*i.e.*, $Q_n = 0$ and $Q'_n = 1$), then S = 0 and R = 1. Since S = 0 and R = 1, the basic flip-flop changes its state and goes to the set state. But if the flip-flop is already in set condition (*i.e.*, $Q_n = 1$ and $Q'_n = 0$), then S = 1 and R = 1. Since S = 1 and R = 1, the basic flip-flop does not alter its state and remains in the set state.

**Case 4.** When the clock is applied and J = 1 and K = 1 and the previous state of the flip-flop is reset (*i.e.*, $Q_n = 0$ and $Q'_n = 1$), then S = 0 and R = 1. Since S = 0 and R = 1, the basic flip-flop changes its state and goes to the set state. But if the flip-flop is already in set condition (*i.e.*, $Q_n = 1$ and $Q'_n = 0$), then S = 1 and R = 0. Since S = 1 and R = 0, the basic flip-flop changes its state and goes to the reset state. So we find that for J = 1 and K = 1, the flip-flop toggles its state from *set* to *reset* and vice versa. Toggle means to switch to the opposite state.

## Characteristic Table of a J-K Flip-flop

As we have already discussed the characteristic equation of an S-R flip-flop, we can similarly find out the characteristic equation of a J-K flip-flop. The characteristic table of a J-K flip-flop is given in the table below. From the characteristic table we have to find out the characteristic equation of the J-K flip-flop.

| *Flip-flop inputs* | | *Present output* | *Next output* |
|---|---|---|---|
| *J* | *K* | $Q_n$ | $Q_{n+1}$ |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

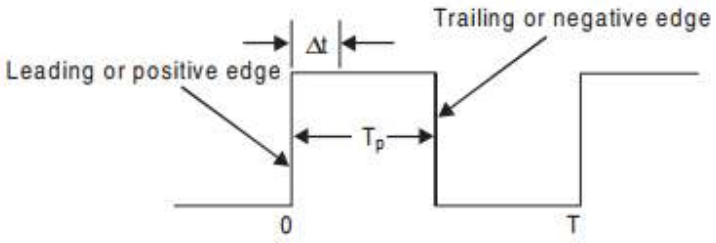



From the Karnaugh map, we obtain $\mathbf{Q_{n+1} = JQ'_n + K'Q_n}$.
Hence, the characteristic equation of a J-K flip-flop is

$$\mathbf{Q_{n+1} = JQ'_n + K'Q_n}$$

## Race-around Condition of a J-K Flip-flop

The inherent difficulty of an S-R flip-flop (*i.e.*, S = R = 1) is eliminated by using the feedback connections from the outputs to the inputs of gate 1 and gate 2 as discussed in JK flip-flop. Truth tables JK flip-flop were formed with the assumption that the inputs do not change during the clock pulse (CLK = 1). But the consideration is not true because of the feedback connections. Consider, for example, that the inputs are J = K = 1 and Q = 1, and a pulse as shown in Figure below is applied at the clock input.



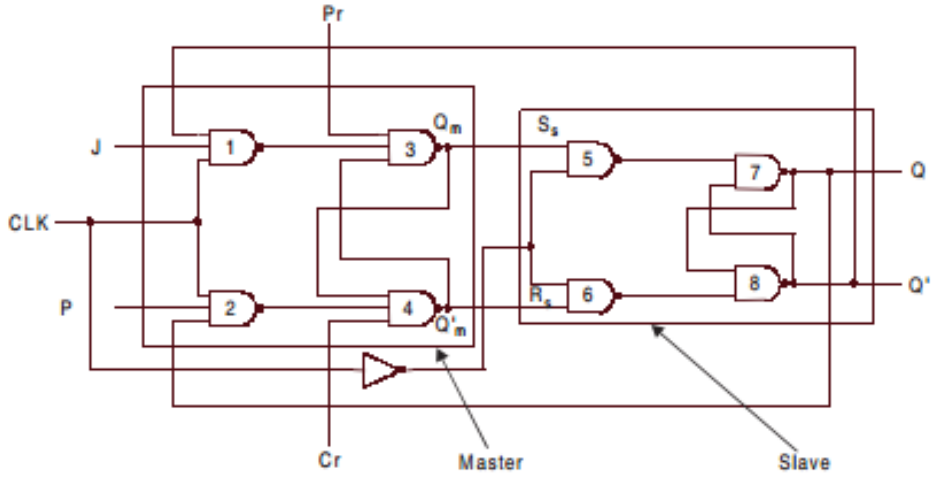

Consider, for example, that the inputs are J = K = 1 and Q = 1, and a pulse as shown above is applied at the clock input. After a time interval Δt equal to the propagation delay through two NAND gates in series, the outputs will change to Q = 0. So now we have J = K = 1 and Q = 0. After another time interval of Δt the output will change back to Q = 1. Hence, we conclude that for the time duration of $t_p$ of the clock pulse, the output will oscillate between 0 and 1. Hence, at the end of the clock pulse, the value of the output is not certain. This situation is referred to as a ***race-around condition***.

Generally, the propagation delay of TTL gates is of the order of nanoseconds. So if the clock pulse is of the order of microseconds, then the output will change thousands of times within the clock pulse. This race-around condition can be avoided if $t_p < \Delta t < T$. Due to the small propagation delay of the ICs it may be difficult to satisfy the above condition. A more practical way to avoid the problem is to use the master-slave (M-S) configuration as discussed below.

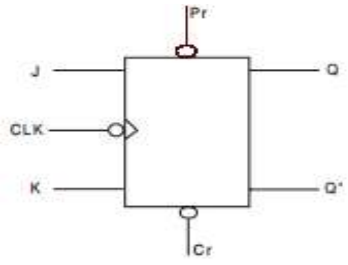
## Master-Slave J-K Flip-flop

A master-slave (M-S) flip-flop is shown in Figure below. Basically, a master-slave flip-flop is a system of two flip-flops—one being designated as *master* and the other is the *slave*. From the figure below we see that a clock pulse is applied to the master and the inverted form of the same clock pulse is applied to the slave.

When CLK = 1, the first flip-flop (*i.e.*, the master) is enabled and the outputs $Q_m$ and $Q'_m$ respond to the inputs J and K according to the table shown in Figure 7.13. At this time the second flip-flop (*i.e.*, the slave) is disabled because the CLK is LOW to the second flip-flop. Similarly, when CLK becomes LOW, the master becomes disabled and the slave becomes active, since now the CLK to it is HIGH. Therefore, the outputs Q and Q' follow the outputs $Q_m$ and $Q'_m$ respectively. Since the second flip-flop just follows the first one, it is referred to as a slave and the first one is called the master. Hence, the configuration is referred to as a master-slave (M-S) flip-flop.




In this type of circuit configuration the inputs to the gates 5 and 6 do not change at the time of application of the clock pulse. Hence the race-around condition does not exist. The state of the master-slave flip-flop, shown in above Figure, changes at the negative transition (trailing edge) of the clock pulse. Hence, it becomes negative triggering a master-slave flip-flop. This can be changed to a positive edge triggering flip-flop by adding two inverters to the system—one before the clock pulse is applied to the master and an additional one in between the master and the slave. The logic symbol of a negative edge master-slave is shown in Figure below.




The system of master-slave flip-flops is not restricted to J-K master-slave only. There may be an S-R master-slave or a D master-slave, etc., in all of them the slave is an S-R flip-flop, whereas the master changes to J-K or S-R or D flip-flops.

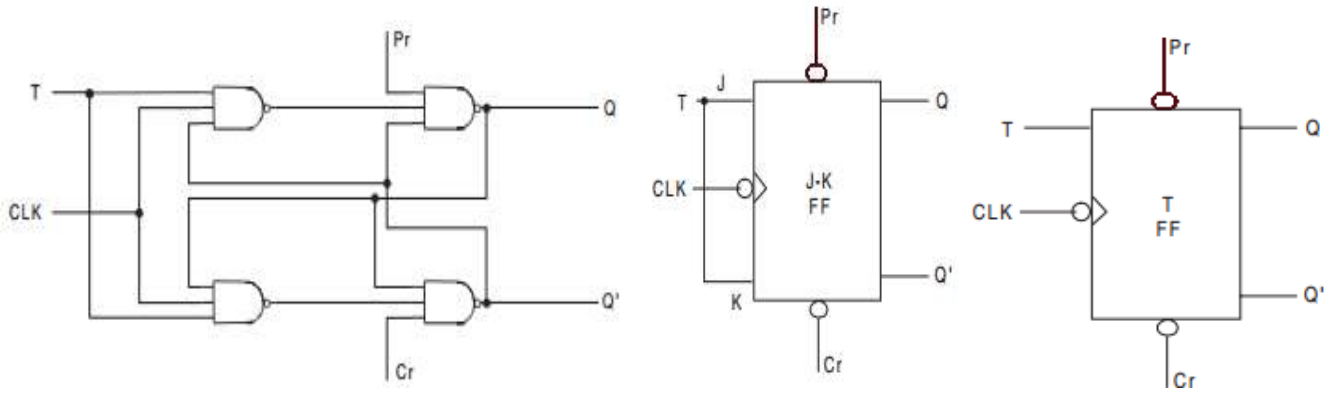
## T Flip-flop

With a slight modification of a J-K flip-flop, we can construct a new flip-flop called a T flip-flop. If the two inputs J and K of a J-K flip-flop are tied together it is referred to as a T flip-flop. Hence, a T flip-flop has only one input T and two outputs Q and Q'. The name T flip-flop actually indicates the fact that the flip-flop has the ability to toggle. It has actually only two states—***toggle state*** **and** ***memory state***. Since there are only two states, a T flip-flop is a very good option to use in counter design and in sequential circuits design where switching an operation is required. The truth table of a T flip-flop is given below:-

| $T$ | $Q_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

If the T input is in 0 state (*i.e.*, J = K = 0) prior to a clock pulse, the Q output will not change with the clock pulse. On the other hand, if the T input is in 1 state (*i.e.*, J = K = 1) prior to a clock pulse, the Q output will change to Q' with the clock pulse. In other words, we may say that, if T = 1 and the device is clocked, then the output toggles its state.

The truth table shows that when T = 0, then $Q_{n+1} = Q_n$, *i.e.*, the next state is the same as the present state and no change occurs. When T = 1, then $Q_{n+1} = Q'_n$, *i.e.*, the state of the flip-flop is complemented. The circuit diagram of a T flip-flop and the block diagram of the T flip-flop is shown below:-

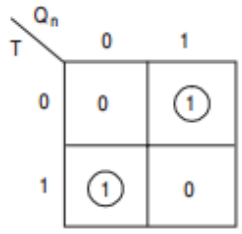



## Characteristic Table of a T Flip-flop

As we have already discussed the characteristic equation of a J-K flip-flop, we can similarly find out the characteristic equation of a T flip-flop. The characteristic table of a T flip-flop is given below. From the characteristic table we have to find out the characteristic equation of the T flip-flop.

| *Flip-flop inputs* | *Present output* | *Next output* |
|---|---|---|
| $T$ | $Q_n$ | $Q_{n+1}$ |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Now we will find out the characteristic equation of the T flip-flop from the characteristic table with the help of the Karnaugh map below:-




From the Karnaugh map, the Boolean expression of $Q_{n+1}$ is derived as $Q_{n+1} = TQ'_n + T'Q_n$. Hence, the characteristic equation of a T flip-flop is

$$Q_{n+1} = TQ'_n + T'Q_n$$

# TRIGGERING OF FLIP-FLOPS

Flip-fl ops are synchronous sequential circuits. This type of circuit works with the application of a synchronization mechanism, which is termed as a *clock*. Based on the specific interval or point in the clock during or at which triggering of the flip-flop takes place, it can be classified into two different types—***level triggering*** **and** ***edge triggering***. A clock pulse starts from an initial value of 0, goes momentarily to 1, and after a short interval, returns to the initial value.

## Level Triggering of Flip-flops

If a flip-flop gets enabled when a clock pulse goes HIGH and remains enabled throughout the duration of the clock pulse remaining HIGH, the flip-flop is said to be a *level triggered flip-flop*. If the flip-flop changes its state when the clock pulse is positive, it is termed as a *positive level triggered flip-flop*. On the other hand, if a NOT gate is introduced in the clock input terminal of the flip-flop, then the flip-flop changes its state when the clock pulse is negative, it is termed as a *negative level triggered flip-flop.* The main drawback of level triggering is that, as long as the clock pulse is active, the flip-flop changes its state more than once or many times for the change in inputs. If the inputs do not change during one clock pulse, then the output remains stable. On the other hand, if the frequency of the input change is higher than the input clock frequency, the output of the flip-flop undergoes multiple changes as long as the clock remains active. This can be overcome by using either master-slave flip-flops or the edge-triggered flip-flop.

## Edge-triggering of Flip-flops

A clock pulse goes from 0 to 1 and then returns from 1 to 0. The Figure below shows the two transitions and they are defined as the *positive edge* (0 to 1 transition) and the *negative edge* (1 to 0 transition). The term *edge-triggered* means that the flip-flop changes its state only at either the positive or negative edge of the clock pulse.

# EXCITATION TABLE OF A FLIP-FLOP

The truth table of a flip-flop is also referred to as the characteristic table of a flip-flop, since this table refers to the operational characteristics of the flip-flop. But in designing sequential circuits, we often face situations where the present state(PS) & the next state(NS) of the flip-flop is specified, and we have to find out the input conditions that must prevail for the desired output condition. By present and next states we mean to say the conditions before and after the clock pulse respectively. For example, the output of an S-R flip-flop before the clock pulse is Q*n* = 1 and it is desired that the output does not change when the clock pulse is applied.

Now from the characteristic table of an S-R flip-flop, we obtain the following conditions:

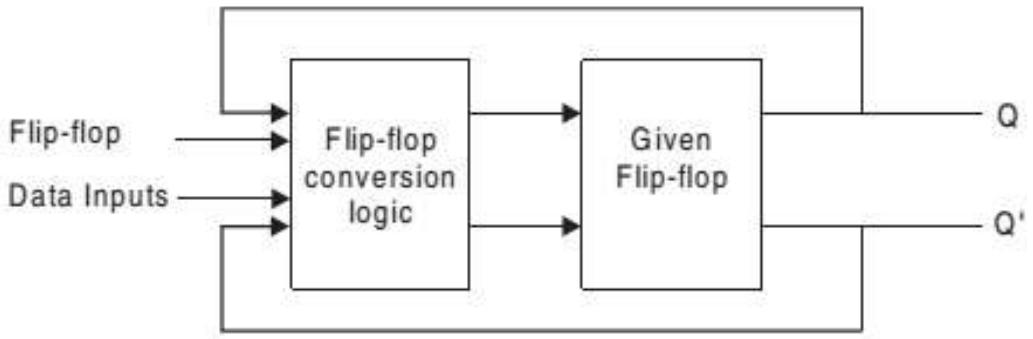
1. S = R = 0 (second row)
2. S = 1, R = 0 (sixth row).

We come to the conclusion from the above conditions that the R input must be 0, whereas the S input may be 0 or 1 (*i.e.*, don't-care). Similarly, for all possible situations, the input conditions can be found out. A tabulation of these conditions is known as an *excitation table*. The table below gives the excitation table for S-R, D, J-K, & T flip-flops. These conditions are derived from the corresponding characteristic tables of the flip-flops.

| *Present* | *Next* | *S-R FF* | | *D-FF* | *J-K FF* | | *T-FF* |
|---|---|---|---|---|---|---|---|
| *State* ($Q_n$) | *State* ($Q_{n+1}$) | $S_n$ | $R_n$ | $D_n$ | $J_n$ | $K_n$ | $T_n$ |
| 0 | 0 | 0 | X | 0 | 0 | X | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | X | 1 |
| 1 | 0 | 0 | 1 | 0 | X | 1 | 1 |
| 1 | 1 | X | 0 | 1 | X | 0 | 0 |

# INTERCONVERSION OF FLIP-FLOPS

In many applications, we are being given a type of flip-flop, whereas we may require some other type. In such cases we may have to convert the given flip-flop to our required flip-flop. Now we may follow a general model for such conversions of flip-flops. The model is shown in below From the model we see that it is required to design the conversion logic for converting new input definitions into input codes that will cause the given flip-flop to work like the desired flip-flop. To design the conversion logic we need to combine the excitation table for both flip-flops and make a truth table with data input(s) and Q as the inputs and the input(s) of the given flip-flop as the output(s).



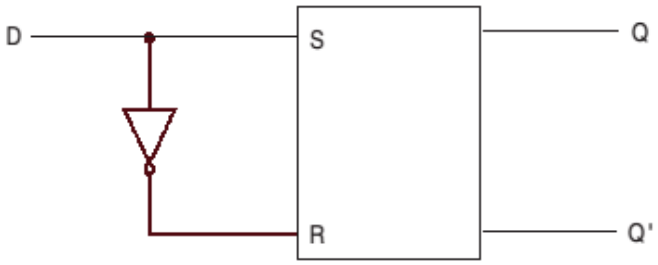
## Conversion of an S-R Flip-flop to a D Flip-flop

The excitation tables of S-R and D flip-flops are given below from which we make the truth table given

| *FF data inputs* | *Output* | *S-R FF inputs* | |
|---|---|---|---|
| *D* | *Q* | *S* | *R* |
| 0 | 0 | 0 | X |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | X | 0 |

From the above table, we make the Karnaugh maps for inputs S and R as shown in Figure below:-


Simplifying with the help of the Karnaugh maps, we obtain S = D and R = D'. Hence the circuit may be designed as in Figure below:-

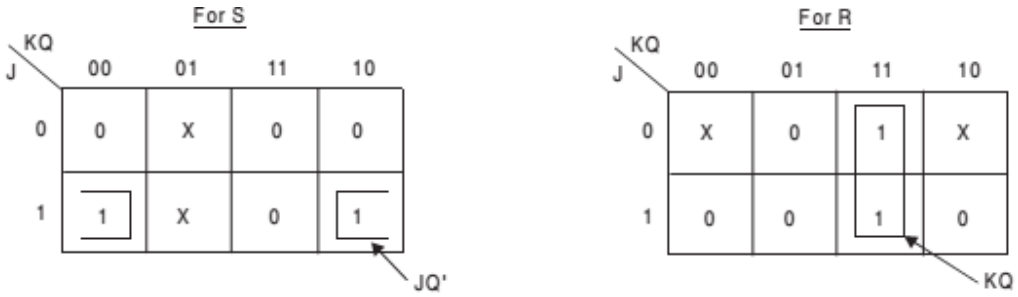



## Conversion of an S-R Flip-fl op to a J-K Flip-flop

The excitation tables of S-R and J-K flip-flops, as we studied before, from which we make the truth table given in below.

| *FF data inputs* | | *Output* | *S-R FF inputs* | |
|---|---|---|---|---|
| *J* | *K* | *Q* | *S* | *R* |
| 0 | 0 | 0 | 0 | X |
| 0 | 1 | 0 | 0 | X |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | X | 0 |
| 1 | 0 | 1 | X | 0 |

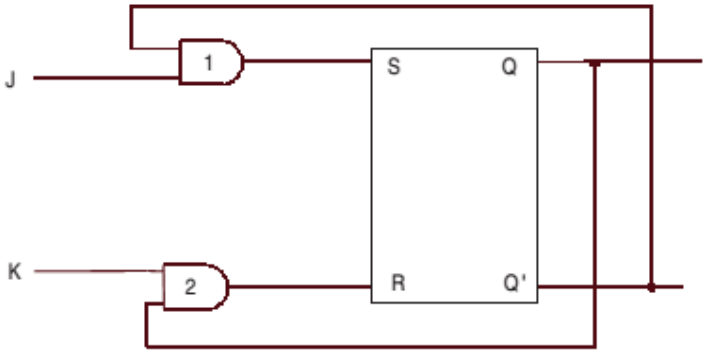
From the above truth table, the Karnaugh map is prepared as shown in Figure below:-




Hence we get the Boolean expression for S and R as

**S = JQ'**
**& R = KQ.**

Hence the circuit may be realized as in below:-

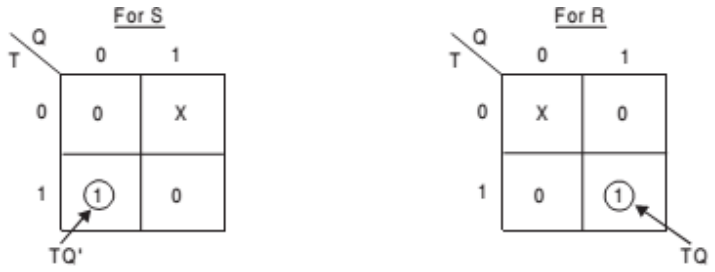



## Conversion of an S-R Flip-flop to a T Flip-flop

The excitation tables of S-R and T flip-flops, as we studied before, from which we make the truth table given in below:-

| *FF data inputs* | *Output* | *S-R FF inputs* | |
|---|---|---|---|
| *T* | *Q* | *S* | *R* |
| 0 | 0 | 0 | X |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | X | 0 |

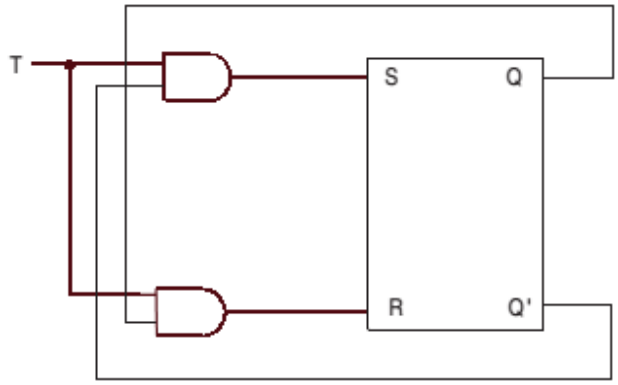
From the above truth table, the Karnaugh map is prepared as shown in Figure below:-



Hence we get the Boolean expression for S and R as:-

**S = TQ' and R = TQ**

Hence the circuit may be realized as in below:-

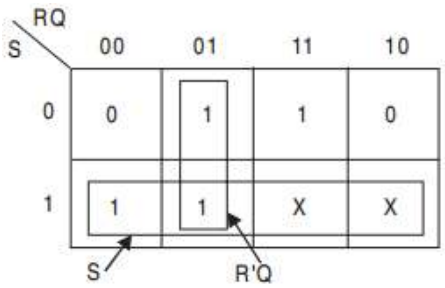



## Conversion of a D Flip-flop to an S-R Flip-flop

The excitation tables of S-R and D flip-flops, as we studied before, from which we make the truth table given in below:-

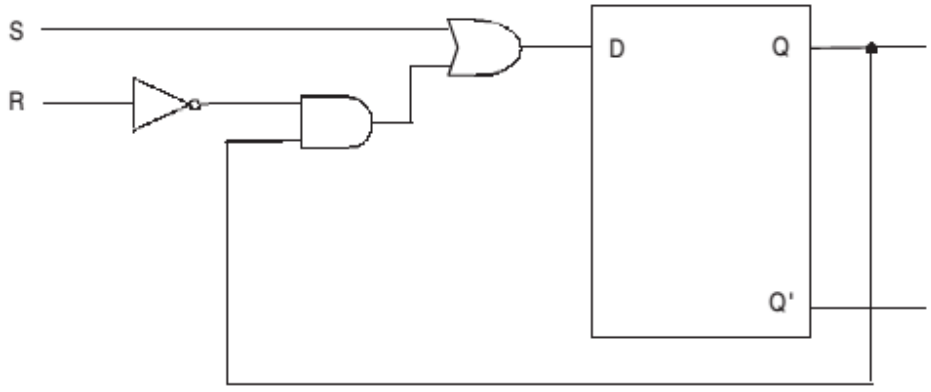
| *FF data inputs* | | *Output* | *D FF inputs* |
|---|---|---|---|
| *S* | *R* | *Q* | *D* |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 |

From the above truth table, the Karnaugh map is prepared as shown in Figure below:-




Hence we get the Boolean expression for S and R as:- **D = S + R'Q**

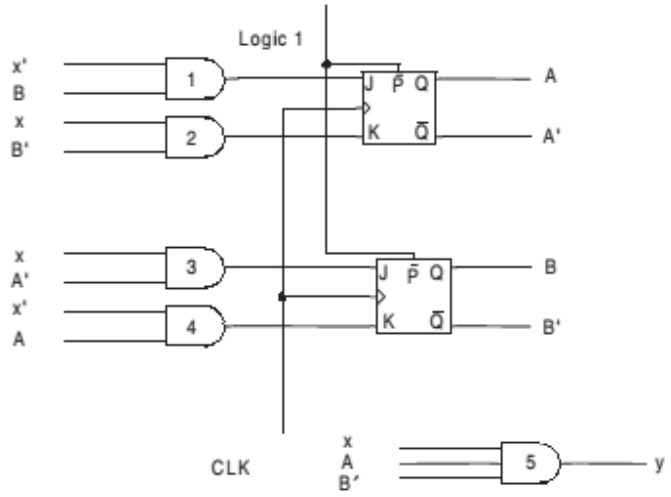
Hence the circuit may be realized as in below:-




Similar procedure is applied for all type of Flip-Flop conversion and is left as an assignment for the student.

# ANALYSIS OF SEQUENTIAL CIRCUITS

The behavior of a sequential circuit is determined from the inputs, the outputs, and the states of the flip-flops. Both the outputs and the next state are a function of the inputs and the present state. The analysis of sequential circuits consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states. Boolean expressions can be written that describe the behavior of the sequential circuits. We first introduce a specific example of a clocked sequential circuit given below to understand its behavior.




## State Table

The time sequence of inputs, outputs and flip-flop states may be enumerated in a state table. The state table for the circuit in Figure above is shown in the table in below. Here in the table there are three sections designated as present state, next state and output. The present state designates the states of the flip-flops before the occurrence of the clock pulse. The next state designates the states of the flip-flops after the application of the clock pulse. The output section shows the values of the output variables during the present state. Again, both the output and the next state sections have two columns, one for $x = 0$ and the other for $x = 1$.

The analysis of the circuit can start from any arbitrary state. In our example, we start the analysis from initial state 00. When the present state is 00, A = 0 and B = 0. From the logic diagram, with $x = 0$, we find both AND gates 1 and 2 produce logic 0 signal and hence the next state remains unchanged. Also, B flip-flop for both AND gates 3 and 4 produce logic 0 signal and hence the next state of B also remains unchanged. Hence, with the clock pulse, flip-flop A and B are both in the memory state, making the next state 00. Similarly, with A = 0 and B = 0, with $x = 1$, we find that gate 1 produces logic 0, whereas gate 2 produces logic 1. Again, with the same condition, gate 3 produces logic 1 whereas gate 4 produces logic 0. Hence, with the clock pulse, flip-flop A is cleared and B is set, making the next state 01. This information is listed in the first row of the state table.

| *Present* | *Next state* | | *Output* | |
|---|---|---|---|---|
| *state* | *x = 0* | *x = 1* | *x = 0* | *x = 1* |
| *AB* | *AB* | *AB* | *y* | *y* |
| 00 | 00 | 01 | 0 | 0 |
| 01 | 11 | 01 | 0 | 0 |
| 10 | 10 | 00 | 0 | 1 |
| 11 | 10 | 11 | 0 | 0 |

In a similar manner, we can derive the other conditions of the state table also. When the present state is 01, i.e., A = 0 & B = 1. From the logic diagram, with $x$ = 0, we find gate 1 produces logic 1 signal and gate 2 produces logic 0. For B flip-flop both gates 3 & 4 produce logic 0 signal & hence the next state of B remains unchanged. Hence, with the clock pulse, flip-flop A is set and B remains in the memory state, making the next state 11. Similarly, with A = 0 and B = 1, with $x$ = 1, we find that both gates 1 and 2 produce logic 0. Again, with the same condition, both gates 3 and 4 produce logic 0. Hence, with the clock pulse, both flip-flops A and B remain in the memory state, making the next state 01. This information is listed in the second row of the state table.

When the present state is 10, A = 1 and B = 0. From the logic diagram, with $x$ = 0, we find both gates 1 and 2 produce logic 0. For B flip-flop gate 3 produces logic 0 signal but gate 4 produces logic 1. Hence, with the clock pulse, flip-flop A remains in the memory state and B is reset, making the next state 10. Similarly, with A = 1 and B = 0, with $x$ = 1, we find that gate 1 produces logic 0, whereas gate 2 produces logic 1. Again, with the same condition, both gates 3 and 4 produce logic 0. Hence, with the clock pulse, A is reset and B remains in the memory state, making the next state 00. This information is listed in the third row of the state table.

Finally when the present state is 11, A = 1 and B = 1. From the logic diagram, with $x$ = 0, we find gate 1 produces logic 1 and gate 2 produces logic 0. For B flip-flop gate 3 produces logic 0 signal but gate 4 produces logic 1. Hence, with the clock pulse, flip-flop A remains in the memory state and B is reset, making the next state 10. Similarly, with A = 1 and B = 1, with $x$ = 1, we find that both gates 1 and 2 produce logic 0. Again, with the same condition, both gates 3 and 4 produce logic 0. Hence, with the clock pulse, both A and B remain in the memory state, making the next state 11. This information is listed in the last row of the state table.

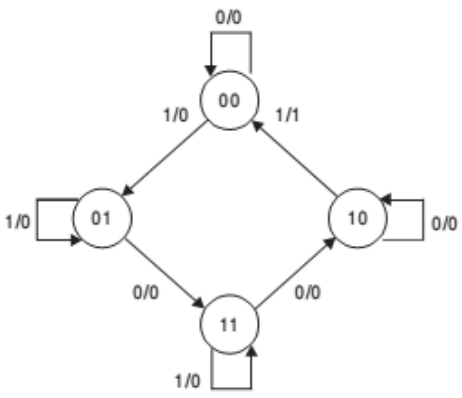
The entries in the output section are easier to derive. In this example, output y = 1 only when $x$ = 1, A = 1, and B = 0. Hence the output columns are marked with 0s except when the present state is 10 and input $x$ = 1, for which $y$ is marked as 1.

The state table of any sequential circuit is obtained by the same procedure used in the example. In general, a sequential circuit with $m$ flip-flops and $n$ input variables will have $2m$ rows, one for each state. The next state and output sections will have $2n$ columns, one for each input combination.

The external output of a sequential circuit may come from memory elements or logic gates. The output section is only included in the state table if there are outputs from logic gates. Any external output taken directly from a flip-flop is already listed in the present state of the state table.

## State Diagram

All the information available in the state table may be represented graphically in the state diagram.



In the diagram, a state is represented by a circle and the transitions between states are indicated by direct arrows connecting the circles. The binary number inside each circle identifies the state the circle represents. The direct arrows are labeled with two binary numbers separated by a /. The number before the / represents the value of the external input, which causes the state transition, and the number after the / represents the value of the output during the present state. For example, the directed arrow from the state 11 to 10 while $x = 0$ and $y = 0$, and that on the termination of the next clock pulse, the circuit goes to the next state 10. A directed arrow connecting a circle with itself indicates that no change of the state occurs.

There is no difference between a state table and a state diagram except in the manner of representation. The state table is easier to derive from a given logic diagram and the state diagram directly follows the state table. The state diagram gives a pictorial form of the state transitions and hence is easier to interpret.

## State Equation

A state equation is an algebraic expression that specifies the conditions for a flip-flop state transition. The left side of the equation denotes the next state of the flip-flop and the right side a Boolean function that specifies the present state conditions that make the next state equal to 1. The state equation is derived directly from a state table. For example, the state equation for flip-flop A can be derived from the table in Figure 7.89. From the next state columns we find that flip-flop A goes to the 1 state four times: when $x = 0$ and AB = 01 or 10 or 11, or when $x = 1$ and AB = 11. This can be expressed algebraically in a state equation as follows:

$$\mathbf{A}(t+1) = \mathbf{A'B + AB' + AB})x' + \mathbf{AB}x$$

Similarly, from the next state columns we find that flip-flop B goes to the 1 state four times: when $x = 0$ and AB = 01 or when $x = 1$ & AB = 00 or 01 or 11. This can be expressed algebraically in a state equation as follows:

$$\mathbf{B}(t+1) = \mathbf{A'B}x' + \mathbf{(A'B' + A'B + AB)}x$$

The right-hand side of the state equation is a Boolean function for the present state. When this function is equal to 1, the occurrence of a clock pulse causes flip-flop A or flip-flop B to have a next state of 1. When this function is equal to 0, the occurrence of a clock pulse causes flip-flop A or flip-flop B to have a next state of 0. The LHS of the equation identifies the flip-flop by its letter symbol, followed by the time function designation ($t$ +1), to emphasize that this value is to be reached by the flip-flop one pulse sequence later The state equation for flip-flop A and B are simplified algebraically below. Hence, we get

$$\begin{aligned} A(t+1) &= (A'B + AB' + AB)x' + ABx \\ &= (Bx')A' + AB'x' + AB \\ &= (Bx')A' + (B + B'x')A \\ &= (Bx')A' + (B + x')A \\ &= (Bx')A' + (B'x)A. \end{aligned}$$

If we let $Bx' = J$ and $B'x = K$, we obtain the relationship: $\mathbf{A}(t+1) = \mathbf{JA' + KA}$.
which is the characteristic equation of the J-K flip-flop. This relationship between the state equation and the characteristic equation can be justified from inspection of the logic diagram in the figure example of a clocked sequential circuit. In it we find that the J input of flip-flop A is equal to the Boolean function $Bx'$ and the K input is equal to $B'x$.

Similarly, for flip-flop B we get

$$\begin{aligned} B(t+1) &= A'Bx' + (A'B' + A'B + AB)x \\ &= (A'x)B' + A'Bx' + Bx \\ &= (A'x)B' + (x + A'x')B \\ &= (A'x)B' + (x + A')B \\ &= (A'x)B' + (Ax')B. \end{aligned}$$

If we let $A'x = J$ and $Ax' = K$, we obtain the relationship: $\mathbf{B(t + 1) = JB' + KB}$ **,** which is the characteristic equation of the J-K flip-flop. In the diagram in example of a clocked sequential circuit, we find that the J input of flip-flop B is equal to the Boolean function $A'x$ and the K input is equal to $Ax'$.

## DESIGN PROCEDURE OF SEQUENTIAL CIRCUITS

The design of a sequential circuit follows certain steps. The steps may be listed as follows:

1. The word description of a circuit may be given accompanied with a state diagram, or timing diagram, or other pertinent information.
2. Then from the given state diagram the state table has to be prepared.
3. If the state reduction mechanism is possible, then the number of states may be reduced.
4. After state reduction, assign binary values to the states if the states contain letter symbols.
5. Then the number of flip-flops required is to be determined. Each flip-flop is assigned a letter symbol.
6. Then the choice has to be made regarding the type of flip-flop to be used.
7. With the help of a state table and the flip-flop excitation table the circuit excitation and the output tables have to be determined.
8. Then using some simplification technique *e.g.*, a Karnaugh map or some other method, the circuit output functions and the flip-flop input functions have to be determined.
9. Then the logic diagram has to be drawn.

Although certain steps have been specified for designing the sequential circuit, the procedure can be shortened with experience. A sequential circuit is made up of flip-flops and combinational gates. One of the most important parts is the choice of flip-flop. From the excitation table of different flip-flops we see that the J-K flip-flop excitation table contains the maximum number of don't-care conditions. Hence, for designing any sequential circuit, it will be most simplified if the circuit is designed with, J-K flip-flop.

The number of flip-flops is determined by the number of states. A circuit may have unused binary states if the total number of states is less than $2m$. The unused states are taken as don't-care conditions during the design of the combinational part of the circuit.

Any design process must consider the problem of minimizing the cost of the final circuit. The most obvious cost reductions are reductions in the number of flip-flops and the number of gates. The reduction of the number of flip-flops in a sequential circuit is referred to as the state reduction. Since $m$ flip-flops produce $2m$ states, a reduction in the number of states may (or may not) result in a reduction of the number of flip-flops. State

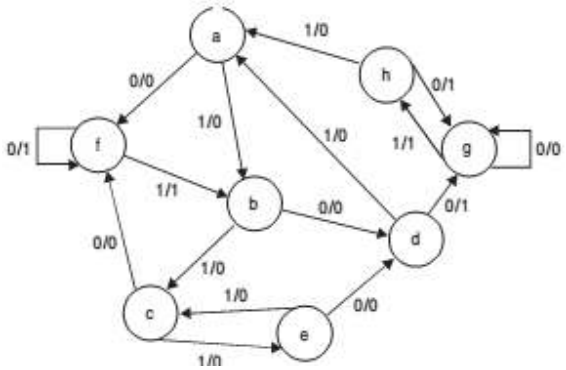
reduction algorithms are concerned with procedures for reducing the number of states in a state table while keeping the external input-output requirements unchanged. An algorithm for the state reduction is given here. If two states in a state table are equivalent, one of them can be removed without altering the input-output relationships.

## SEQUENTIAL LOGIC CIRCUITS

Now two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit to the same state or to an equivalent state. We will discuss the state reduction problem with an example in this section later on.

In certain cases the states are specified in letter symbols. In such cases there comes another factor, called state assignment. State assignment procedures are concerned with methods for assigning binary values to states in such a way as to reduce the cost of the combinational circuit that drives the flip-flop. For any problem there may be a number of different state assignments leading to different combinational parts of the sequential circuit. The most common criterion is that the chosen assignment should result in a simple combinational circuit for the flip-flop inputs. However, to date, there are no state assignment procedures that guarantee a minimal-cost combinational circuit.

We now wish to design the clocked sequential circuit whose state diagram is given below:-




The state table for the state diagram shown above is shown in the table in Figure below.

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

We now look for two equivalent states, & find that $d$ & $h$ are two such states; they both go to $g$ & $a$ and have outputs of 1 and 0 for $x = 0$ & $x = 1$, respectively. Therefore, states $d$ and $h$ are equivalent; one can be removed. Similarly, we find that $b$ and $e$ are again two such states; they both go to $d$ and $c$ and have outputs of 0 and 0 for

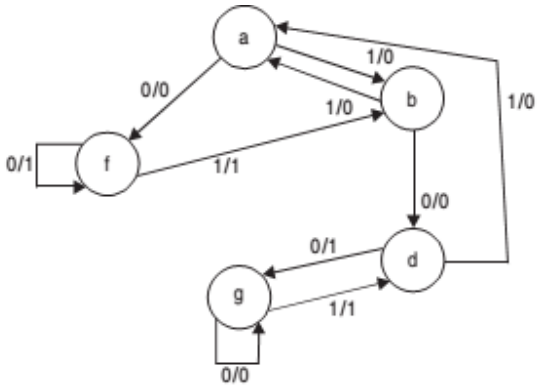
$x = 0$ and $x = 1$, respectively. Therefore, states $b$ and $e$ are also equivalent; and one can be removed. The procedure of removing a state and replacing it by its equivalent is demonstrated in the table in Figure below. From the below table we find that present state $c$ now has next states $f$ and $b$ and outputs 0 and 0 for $x = 0$ and $x = 1$, respectively. The same next states and outputs appear in the row with present state $a$. Therefore, states $a$ and $c$ are equivalent; state $c$ can be removed and replaced by $a$.

| *Present state* | *Next state* | | *Output* | |
|---|---|---|---|---|
| | $x = 0$ | $x = 1$ | $x = 0$ | $x = 1$ |
| $a$ | $f$ | $b$ | 0 | 0 |
| $b$ | $d$ | ~~$e$~~ $a$ | 0 | 0 |
| ~~$c$~~ | $f$ | ~~$e$~~ $b$ | 0 | 0 |
| $d$ | $g$ | $a$ | 1 | 0 |
| ~~$e$~~ | $d$ | $c$ | 0 | 0 |
| $f$ | $f$ | $b$ | 1 | 1 |
| $g$ | $g$ | ~~$h$~~ $d$ | 0 | 1 |
| ~~$h$~~ | $g$ | $a$ | 1 | 0 |

The final reduced state table is shown in below:-

| *Present state* | *Next state* | | *Output* | |
|---|---|---|---|---|
| | $x = 0$ | $x = 1$ | $x = 0$ | $x = 1$ |
| $a$ | $f$ | $b$ | 0 | 0 |
| $b$ | $d$ | $a$ | 0 | 0 |
| $d$ | $g$ | $a$ | 1 | 0 |
| $f$ | $f$ | $b$ | 1 | 1 |
| $g$ | $g$ | $d$ | 0 | 1 |

The state diagram for the reduced state table consists of only five states and is shown in Figure below:-




We now assign the different states the binary values. As we have already discussed, there may be a variety of state assignments. Some of them are shown in the below table. Among them we may choose any of them and accordingly design the circuit.

| State | Assignment 1 | Assignment 2 | Assignment 3 | Assignment 4 |
|---|---|---|---|---|
| *a* | 000 | 001 | 111 | 011 |
| *b* | 001 | 010 | 001 | 101 |
| *d* | 010 | 011 | 110 | 111 |
| *f* | 011 | 100 | 101 | 001 |
| *g* | 100 | 101 | 010 | 000 |

In the table in Figure below, we have used binary assignment 1 to substitute the letter symbols of the five states. It is obvious that a different binary assignment will result in a state table, with completely new binary values for the states while the input-output relationships will remain the same. We will now show the procedure for obtaining the excitation table and the combinational gate structure.

| Present state | Next state | | Output | |
|---|---|---|---|---|
| | *x = 0* | *x = 1* | *x = 0* | *x = 1* |
| 000 | 011 | 001 | 0 | 0 |
| 001 | 010 | 000 | 0 | 0 |
| 010 | 100 | 000 | 1 | 0 |
| 011 | 011 | 001 | 1 | 1 |
| 100 | 100 | 010 | 0 | 1 |

The derivation of the excitation table is facilitated if we arrange the state table in a different form. This form is shown in the below table, where the present state and the input variables are arranged in the form of a truth table. As we have previously said, we may use any flip-flop, but the simplest form of the circuit is possible with J-K flip-flops. So we now design the circuit using J-K flip-flops.

| Present state | | | Input | Next state | | | Flip-flop inputs | | | | | | Output |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *A* | *B* | *C* | *x* | *A* | *B* | *C* | *JA* | *KA* | *JB* | *KB* | *JC* | *KC* | *y* |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | X | 1 | X | 1 | X | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X | 0 | X | 1 | X | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | X | 1 | X | X | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | X | 0 | X | X | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | X | X | 1 | 0 | X | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | X | X | 1 | 0 | X | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | X | X | 0 | X | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | X | X | 1 | X | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | X | 0 | 0 | X | 0 | X | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | X | 1 | 1 | X | 0 | X | 1 |

There are three unused states in this circuit: binary states 101, 110, and 111. When an input of 0 or 1 is included with these unused states, we obtain six don't-care terms. These six binary combinations are not listed in the table under the present state or input and are treated as don't-care terms.

In below figure Karnaugh maps are prepared for JA, KA, JB, KB, JC, and KC.

For JA

| AB \ Cx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 1 | 0 | 0 | 0 |
| 11 | X | X | X | X |
| 10 | X | X | X | X |

For KA

| AB \ Cx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | X | X | X | X |
| 01 | X | X | X | X |
| 11 | X | X | X | X |
| 10 | 0 | 1 | X | X |

From the Karnaugh maps for JA and KA, we obtain

JA = BC'$x$' and

KA = $x$.

For JB

| AB \ Cx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 1 |
| 01 | X | X | X | X |
| 11 | X | X | X | X |
| 10 | 0 | 1 | X | X |

For KB

| AB \ Cx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | X | X | X | X |
| 01 | 1 | 1 | 1 | 0 |
| 11 | X | X | X | X |
| 10 | 0 | X | X | X |

The Boolean expressions are derived for JB and KB from the Karnaugh maps as

JB = A$x$ + A'$x$' and

KB = C' + $x$.

For JC

| AB \ Cx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | X | X |
| 01 | 0 | 0 | X | X |
| 11 | X | X | X | X |
| 10 | 0 | 0 | X | X |

For KC

| AB \ Cx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | X | X | 1 | 1 |
| 01 | X | X | 0 | 0 |
| 11 | X | X | X | X |
| 10 | X | X | X | X |

Similarly, the expressions for JC and KC we obtain as

JC = A'B' and

KC = B'.

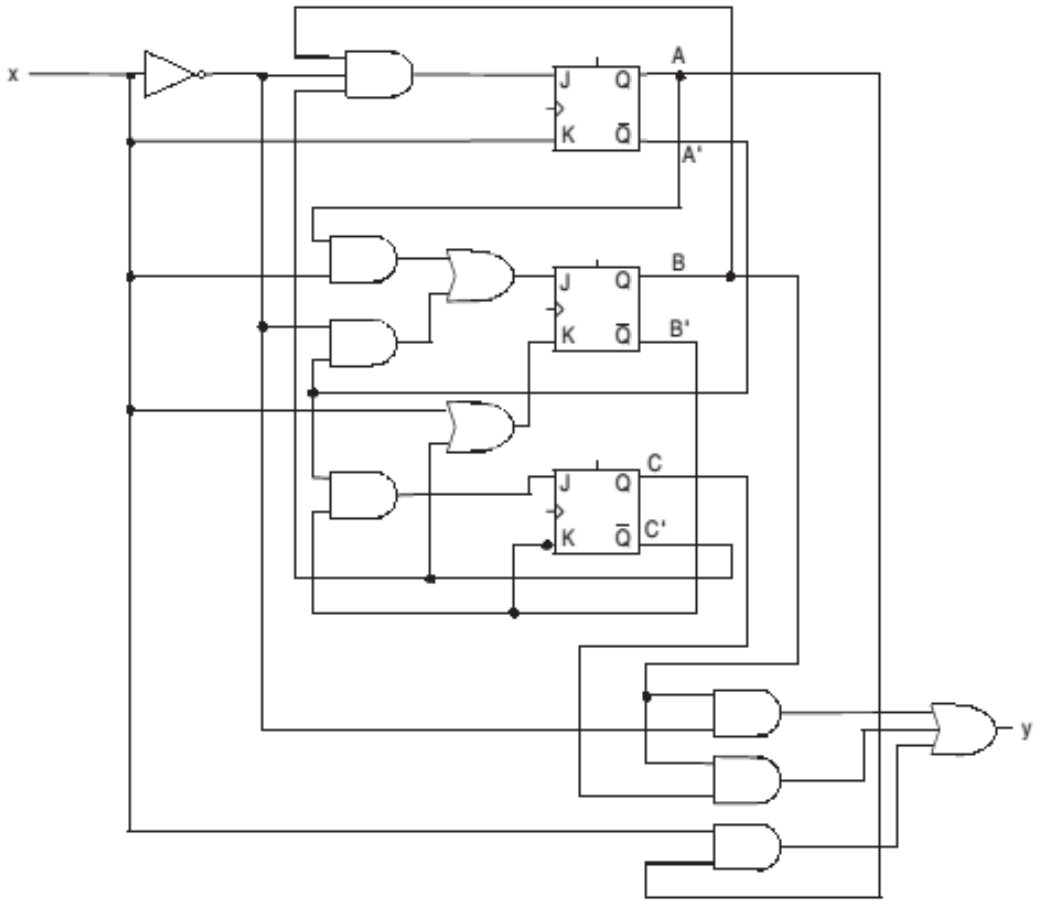
A Karnaugh map has been also prepared below for output *y* and the Boolean expression for *y* is obtained as

**Y = B*x*' + BC + A*x***

For Y

| Cx / AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 1 | 0 | 1 | 1 |
| 11 | X | X | X | X |
| 10 | 0 | 1 | X | X |

The circuit diagram of the desired sequential logic network is shown in Figure below:-



# REGISTERS

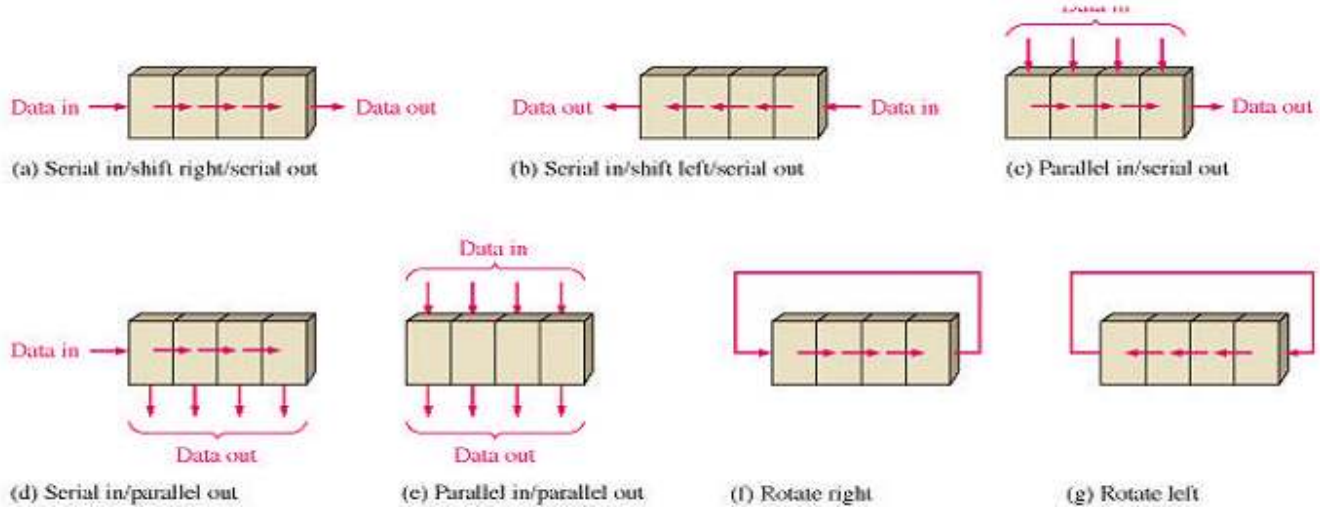
*A register* is a group of binary storage cells capable of holding binary information. A group of flip-flops constitutes a register, since each flip-flop can work as a binary cell. An *n*-bit register, has n flip-flops and is capable of holding n-bits information. In addition to flip-flops a register can have a combinational part that performs data-processing tasks.

***Register:***

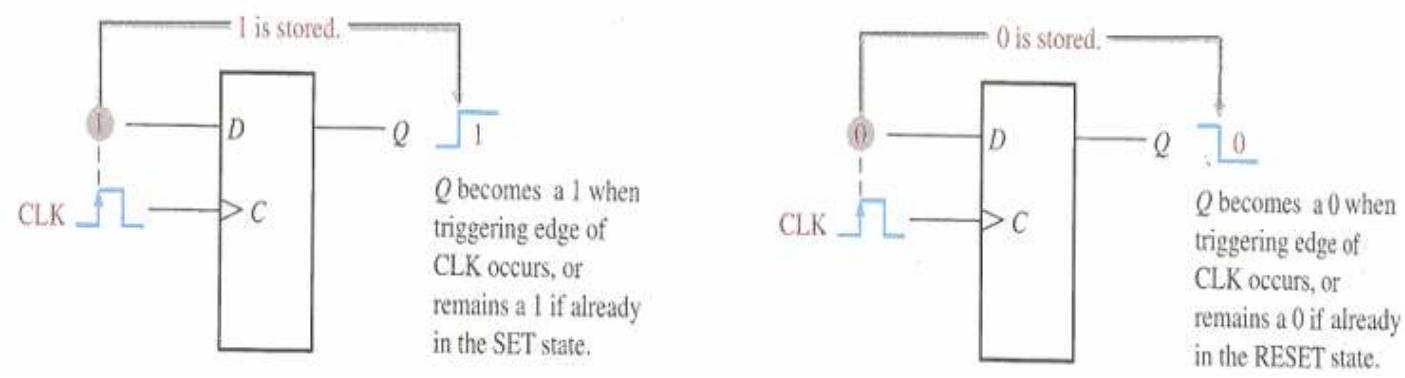
- A set of n flip-flops
- Each flip-flop stores one bit
- Two basic functions: data storage and data movement.

***Shift Register:*** A register that allows each of the flip-flops to pass the stored information to its adjacent neighbor.
***Counter:*** A register that goes through a predetermined sequence of states.

## Basic data movement operation in shift registers

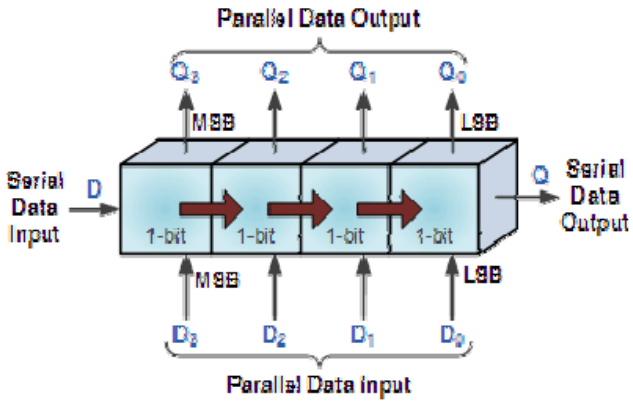



## Storage Capacity of a register



The storage capacity of a register is the total number of bits (1 or 0) of digital data it can retain. Each stage (flip flop) in a shift register represents one bit of storage capacity. Therefore the number of stages in a register determines its storage capacity.

The effect of data movement from left to right through a shift register can be presented graphically as:




# Shift Register

A shift register is a storage device that used to store binary data. When a number of flip flop are connected in series it is called a register. A single flip flop is supposed to stay in one of the two stable states 1 or 0 or in other words the flip flop contains a number 1 or 0 depending upon the state in which it is. A register will thus contain a series of bits which can be termed as a word or a byte.

If in these registers the connection is done in such a way that the output of one of the flip flop forms in input to other, it is known as a **shift register**. The data in a shift register is moved serially (one bit at a time).

The shift register can be built using RS, JK or D flip-flops various types of shift registers are available some of them are given as under.

1. Shift Left Register
2. Shift Right Register
3. Shift Around Register
4. Bi-directional Shift Register

There are two ways to shift data into a register (serial or parallel) and similarly two ways to shift the data out of the register. This leads to the construction of four basic types of registers:-

1. Serial in/Serial out (SISO)
2. Serial in/Parallel out (SIPO)
3. Parallel in/Serial out (PISO)
4. Parallel in/Parallel out (PIPO)

## SERIAL-IN–-SERIAL-OUT SHIFT REGISTER

From the name itself it is obvious that this type of register accepts data serially, *i.e.*, one bit at a time at the single input line. The output is also obtained on a single output line in a serial fashion. The data within the register may be shifted from left to right using *shift-left* register, or may be shifted from right to left using *shift-right* register.

# Shift-right Register

A shift-right register can be constructed with either J-K or D flip-flops as shown in Figure 8.3. A J-K flip-flop based shift register requires connection of both J and K inputs. Input data are connected to the J and K inputs of the left most (lowest order) flip-fl op. To input a 0, one should apply a 0 at the J input, *i.e.*, J = 0 and K = 1 and vice versa. With the application of a clock pulse the data will be shifted by one bit to the right.

In the shift register using D flip-flop, D input of the left most flip-flop is used as a serial input line. To input 0, one should apply 0 at the D input and vice versa.

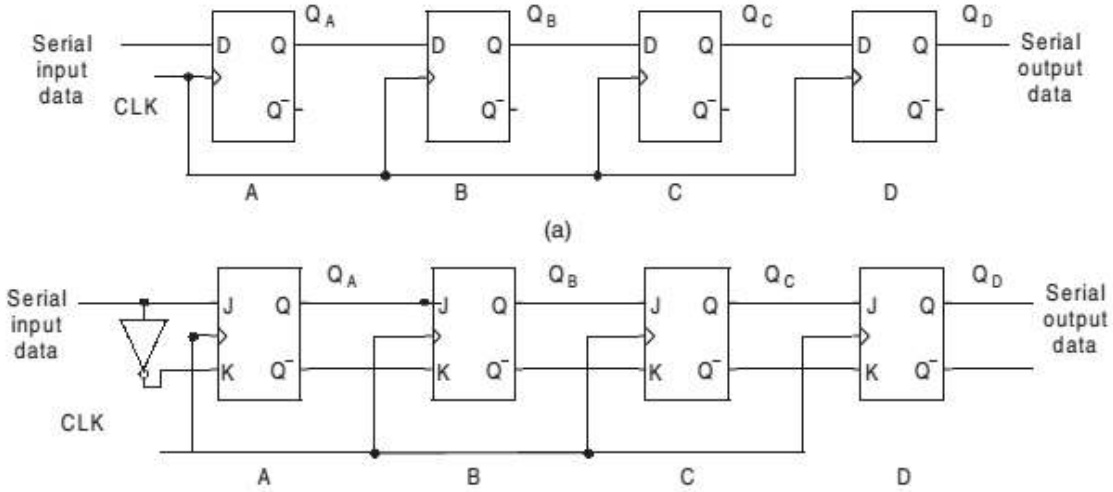



**Figure of Shift-right register (a) using D flip-flops, (b) using J-K flip-flops.**

The clock pulse is applied to all the flip-flops simultaneously. When the clock pulse is applied, each flip-flop is either set or reset according to the data available at that point of time at the respective inputs of the individual flip-flops. Hence the input data bit at the serial input line is entered into flip-flop A by the first clock pulse. At the same time, the data of stage A is shifted into stage B and so on to the following stages. For each clock pulse, data stored in the register is shifted to the right by one stage. New data is entered into stage A, whereas the data present in stage D are shifted out (to the right).

**Operation of the Shift-right Register:-**

| *Timing pulse* | $Q_A$ | $Q_B$ | $Q_C$ | $Q_D$ | *Serial output at* $Q_D$ |
|---|---|---|---|---|---|
| Initial value | 0 | 0 | 0 | 0 | 0 |
| After 1st clock pulse | 1 | 0 | 0 | 0 | 0 |
| After 2nd clock pulse | 1 | 1 | 0 | 0 | 0 |
| After 3rd clock pulse | 0 | 1 | 1 | 0 | 0 |
| After 4th clock pulse | 1 | 0 | 1 | 1 | 1 |

For example, consider that all the stages are reset and a logical input 1011 is applied at the serial input line connected to stage A. The data after four clock pulses is shown in above Table.

Let us now illustrate the entry of the 4-bit number 1011 into the register, beginning with the right-most bit. A 1 is applied at the serial input line, making D = 1. As the first clock pulse is applied, flip-flop A is SET, thus

storing the 1. Next, a 1 is applied to the serial input, making D = 1 for flip-flop A and D = 1 for flip-flop B also, because the input of flip-flop B is connected to the $Q_A$ output.

When the second clock pulse occurs, the 1 on the data input is "shifted" to the flip-flop A and the 1 in the flip-flop A is "shifted" to flip-flop B. The 0 in the binary number is now applied at the serial input line, and the third clock pulse is now applied. This 0 is entered in flip-flop A and the 1 stored in flip-flop A is now "shifted" to flip-flop B and the 1 stored in flip-flop B is now "shifted" to flip-flop C. The last bit in the binary number that is the 1 is now applied at the serial input line and the fourth clock pulse is now applied. This 1 now enters the flip-flop A and the 0 stored in flip-flop A is now "shifted" to flip-flop B and the 1 stored in flip-flop B is now "shifted" to flip-flop C and the 1 stored in flip-flop C is now "shifted" to flip-flop D. Thus the entry of the 4-bit binary number in the shift-right register is now completed.

From the third column of above Table we can get the serial output of the data that is being entered in the register. We find that after the first, second, and the third clock pulses the output at the serial output line *i.e.*, $Q_D$ is 0. After the fourth clock pulse the output at the serial output line is 1. If we want to get the total data that we have entered in the register in a serial manner from $Q_D$, then we have to apply another three clock pulses. After the fifth clock pulse we will gate another 1 at $Q_D$. After the sixth clock pulse the output at $Q_D$ will be 0 and after the seventh clock pulse the output at $Q_D$ will be 1. In this process of the fifth, sixth, and the seventh clock pulses if no data is being supplied at the serial input line then the A, B, and C flip-flops will again be RESET with output 0.

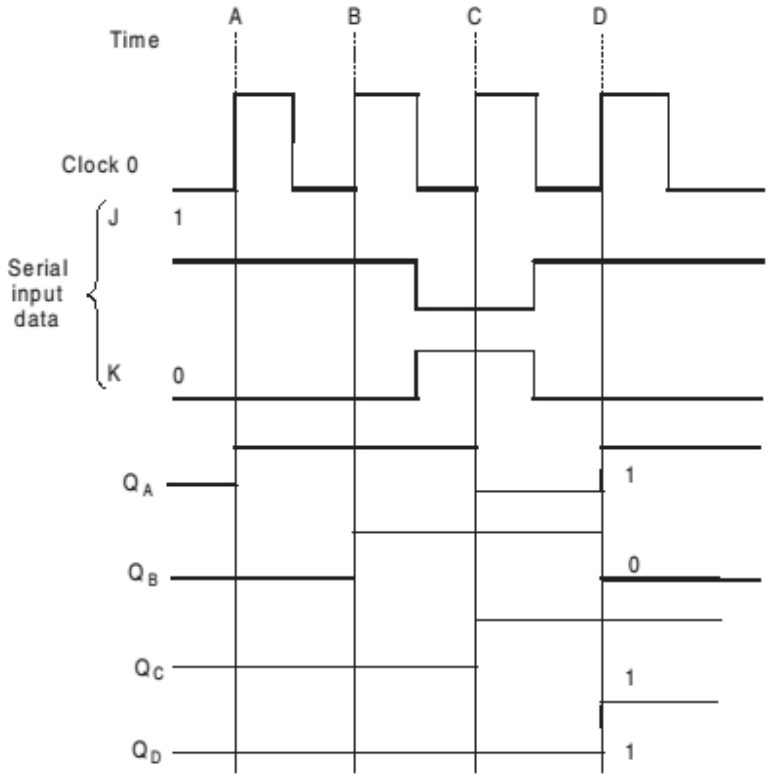



**Figure:-**Waveforms of 4-bit serial input shift-right register.

The waveforms shown above illustrate the entry of a 4-bit number 1011. For a J-K flip-flop, the data bit to be shifted into the flip-flop must be present at the J and K inputs when the clock transitions from low to high occur. Since the data bit is either 1 or 0, there can be two different cases:

1. To shift a 1 into the flip-flop, J = 1 and K = 0,
2. To shift a 0 into the flip-flop, J = 0 and K = 1.

**At time A:** All the flip-flops are reset. At the serial data input line a 1 is given and with the first clock pulse this 1 is shifted at $Q_A$ making $Q_A = 1$. At the same time the 0 in $Q_A$ is shifted to $Q_B$, and the 0 in $Q_B$ is shifted to $Q_C$ and the 0 in $Q_C$ is shifted to $Q_D$. Hence the flip-flop outputs just after time A are $Q_AQ_BQ_CQ_D = 1000$.

2. **At time B:** The flip-flop A contains 1, & all other flip-flop contains 0. Now, again, 1 is given at the serial data input line. With the 2nd clock pulse this 1 is shifted to $Q_A$. The 1 in $Q_A$ is shifted to $Q_B$ & the 0 in $Q_B$ is shifted to $Q_C$ and the 0 in $Q_C$ is shifted to $Q_D$. Hence the flip-flop outputs just after time B are $Q_AQ_BQ_CQ_D = 1100$.

3. **At time C:** The flip-flop A & B contain 1, & all other flip-flops contain 0. Now a 0 is given at the serial data input line. With the 3rd clock pulse this 0 is shifted to $Q_A$. The 1 in $Q_A$ is shifted to $Q_B$ & the 1 in $Q_B$ is shifted to $Q_C$ & the 0 in $Q_C$ is shifted to $Q_D$. Hence the flip-flop outputs just after time C are $Q_AQ_BQ_CQ_D = 0110$.

4. **At time D:** The flip-flop B and flip-flop C contain 1, and all other flip-flops contain 0. Now another 1 is given at the serial data input line. With the fourth clock pulse this 1 is shifted to $Q_A$. The 0 in $Q_A$ is shifted to $Q_B$ and the 1 in $Q_B$ is shifted to $Q_C$ and the 1 in $Q_C$ is shifted to $Q_D$. Hence the flip-flop outputs just after time C are $Q_AQ_BQ_CQ_D = 1011$. To summarize, we have shifted 4 data bits in a serial manner into four flip-flops. These 4 data bits could represent a 4-bit binary number 1011, assuming that we began shifting with the LSB first. Notice that the LSB is in D and the MSB is in A. These four flip-flops could be defined as a 4-bit shift register.

## Shift-left Register

A shift-left register can also be constructed with either J-K or D flip-flops as shown in Figure below. Let us now illustrate the entry of the 4-bit number 1110 into the register, beginning with the right-most bit. A 0 is applied at the serial input line, making D = 0. As the first clock pulse is applied, flip-fl op A is RESET, thus storing the 0. Next a 1 is applied to the serial input, making D = 1 for flip-flop A and D = 0 for flip-flop B, because the input of flip-flop B is connected to the $Q_A$ output.

When the second clock pulse occurs, the 1 on the data input is “shifted” to the flip-flop A and the 0 in the flip-flop A is “shifted” to flip-flop B. The 1 in the binary number is now applied at the serial input line, and the third clock pulse is now applied. This 1 is entered in flip-flop A and the 1 stored in flip-flop A is now “shifted” to flip-flop B and the 0 stored in flip-flop B is now “shifted” to flip-flop C. The last bit in the binary number that is the 1 is now applied at the serial input line and the fourth clock pulse is now applied. This 1 now enters the flip-flop A and the 1 stored in flip-flop A is now “shifted” to flip-flop B and the 1 stored in flip-flop B is now “shifted” to flip-flop C and the 0 stored in flip-flop C is now “shifted” to flip-flop D. Thus the entry of the 4-bit binary number in the shift-right register is now completed.

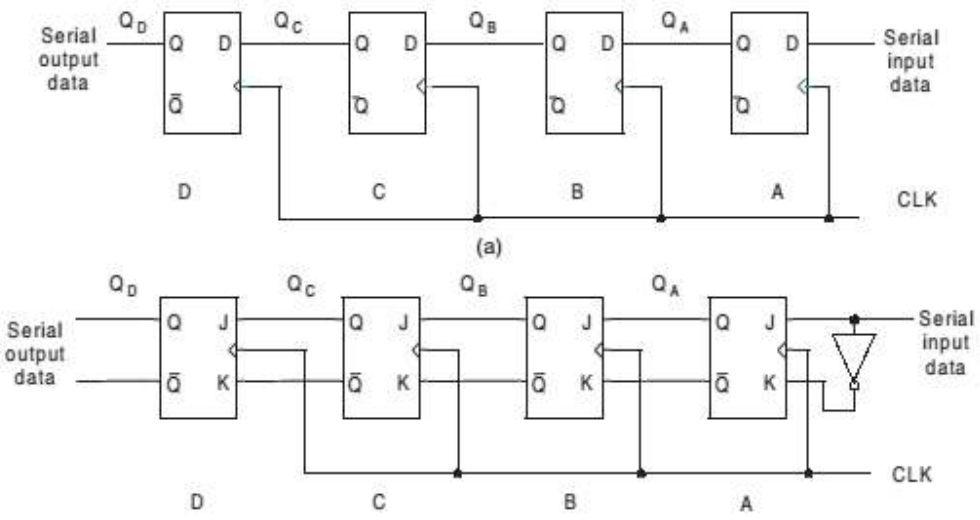



**Figure:-** Shift-left register (*a*) using D flip-flops, (*b*) using J-K flip-flops.

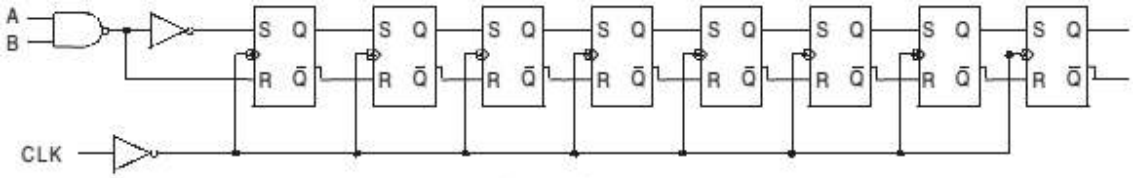
| *Timing pulse* | $Q_D$ | $Q_C$ | $Q_B$ | $Q_A$ | *Serial output at* $Q_D$ |
|---|---|---|---|---|---|
| Initial value | 0 | 0 | 0 | 0 | 0 |
| After 1st clock pulse | 0 | 0 | 0 | 0 | 0 |
| After 2nd clock pulse | 0 | 0 | 0 | 1 | 0 |
| After 3rd clock pulse | 0 | 0 | 1 | 1 | 0 |
| After 4th clock pulse | 0 | 1 | 1 | 1 | 0 |

## 8-bit Serial-in–Serial-out Shift Register

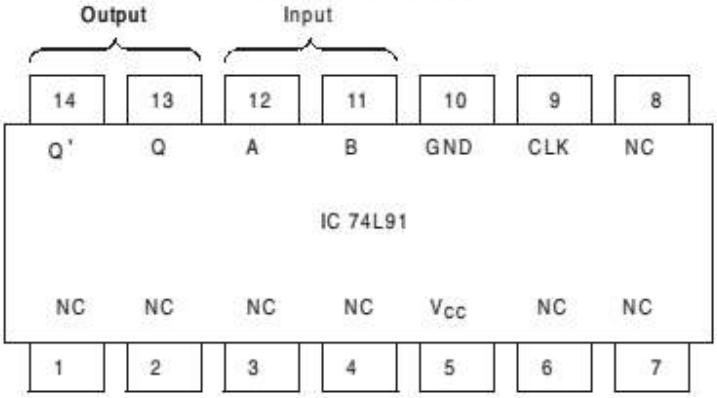
The pinout and logic diagram of IC 74L91 is shown in Figure below. This IC is actually an example of an 8-bit serial-in–serial-out shift register. There are eight S-R flip-flops connected to provide a serial input as well as a serial output. The clock input at each flip-flop is negative edge-triggered. However, the applied clock signal is passed through an inverter. Hence the data will be shifted on the positive edges of the input clock pulses.

An inverter is connected in between R and S on the first flip-flop. This means that this circuit functions as a D-type flip-flop. So the input to the register is a single liner on which the data can be shifted into the register appears serially. The data input is applied at either A (pin 12) or B (pin 11). The data level at A (or B) is complemented by the NAND gate and then applied to the R input of the first flip-flop. The same data level is complemented by the NAND gate and then again complemented by the inverter before it appears at the S input. So, a 0 at input A will *reset* the first flip-flop (in other words this 0 is shifted into the first flip-flop) on a positive clock transition.

The NAND gate with A and B inputs provide a gating function for the input data stream if required, if gating is not required, simply connect pins 11 and 12 together and apply the input data stream to this connection.




(a) Logic diagram.




(b) Pinout diagram of IC 74L91.

# SERIAL-IN–PARALLEL-OUT REGISTER

In this type of register, the data is shifted in serially, but shifted out in parallel. To obtain the output data in parallel, it is required that all the output bits are available at the same time. This can be accomplished by connecting the output of each flip-flop to an output pin. Once the data is stored in the flip-flop the bits are available simultaneously. The basic configuration of a serial-in–parallel-out shift register is shown in below.

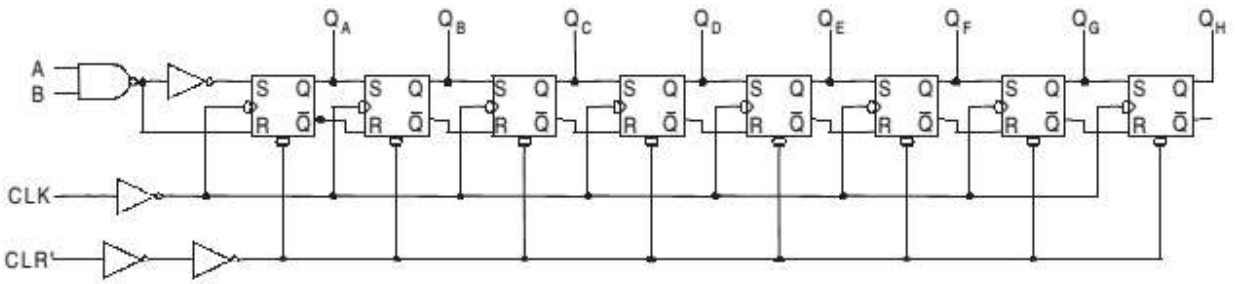
## 8-bit Serial-in–Parallel-out Shift Register

The pinout and logic diagram of IC 74164 is shown in Figure below. IC 74164 is an example of an 8-bit SIPO shift register. There are eight S-R flip-flops, which are all sensitive to negative clock transitions. The logic diagram in Figure below is almost the same as shown in SISO with only two exceptions: (1) each flip-flop has an asynchronous CLEAR input; and (2) the true side of each flip-flop is available as an output—thus all 8 bits of any number stored in the register are available simultaneously as an output (this is a parallel data output).

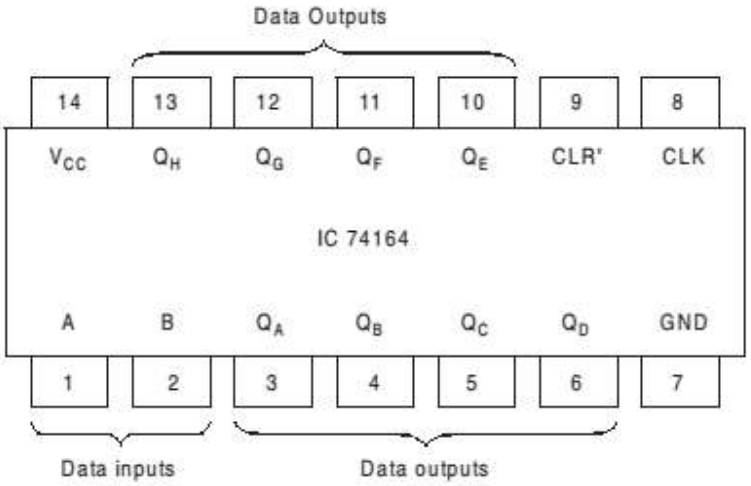
Hence, a low level at the CLR input to the chip (pin 9) is applied through an amplifier and will reset every flip-flop. As long as the CLR input to the chip is LOW, the flip-flop outputs will all remain low. It means that, in effect, the register will contain all zeros.

Shifting of data into the register in a serial fashion is exactly the same as the IC 74L91. Data at the serial input may be changed while the clock is either low or high, but the usual hold and setup times must be observed. The data sheet for this device gives hold time as 0.0 ns and setup time as 30 ns.

Now we try to analyze the gated serial inputs A and B. Suppose that the serial data is connected to B; then A can be used as a control line. Here's how it works:




(a) Logic diagram.




(b) Pinout diagram of IC 74164.

**A is held high:** The NAND gate is enabled and the serial input data passes through the NAND gate inverted. The input data is shifted serially into the register.
**A is held low:** The NAND gate output is forced high, the input data steam is inhibited, and the next clock pulse will shift a 0 into the first flip-flop. Each succeeding positive clock pulse will shift another 0 into the register. After eight clock pulses, the register will be full of zeros.

## PARALLEL-IN–SERIAL-OUT REGISTER

In the preceding two cases the data was shifted into the registers in a serial manner. Here we develop an idea for the parallel entry of data into the register. Here the data bits are entered into the flip-flops simultaneously, rather than a bit-by-bit basis.

A 4-bit parallel-in–serial-out register is illustrated in Figure below. A, B, C, and D are the four parallel data input lines and *SHIFT* / $\overline{LOAD}$ $(SH / \overline{LD})$ is a control input that allows the four bits of data at A, B, C, and D inputs to enter into the register in parallel or shift the data in serial. When *SHIFT* / $\overline{LOAD}$ is HIGH, AND gates $G_1$, $G_3$ & $G_5$ are enabled, allowing the data bits to shift right from one stage to the next. When *SHIFT* / $\overline{LOAD}$ is LOW, AND gates $G_2$, $G_4$, and $G_6$ are enabled, allowing the data bits at the parallel inputs. When a clock pulse is applied, the flip-flops with D = 1 will be set and the flip-flops with D = 0 will be reset, thereby storing all the four bits simultaneously. The OR gates allow either the normal shifting operation or the parallel data-entry operation, depending on which of the AND gates are enabled by the level on the *SHIFT* / $\overline{LOAD}$ input.

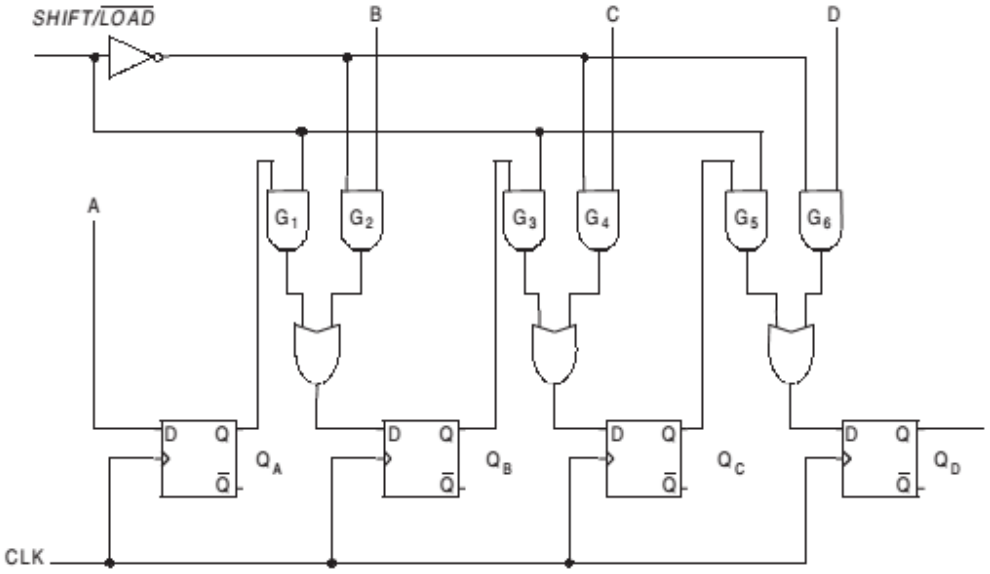
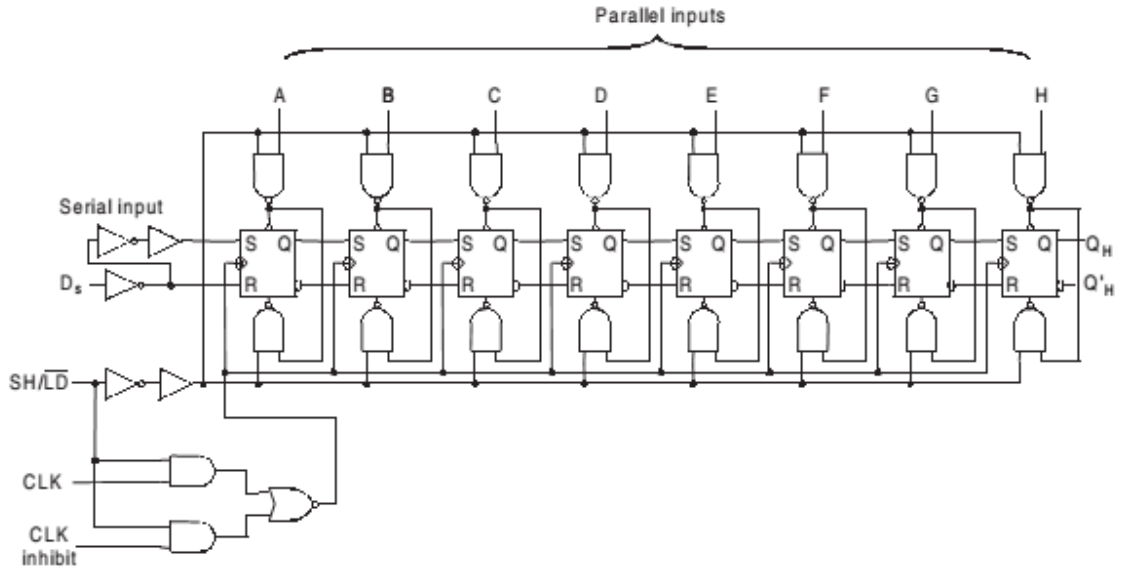



**Figure:-** A 4-bit parallel-in–serial-out shift register.

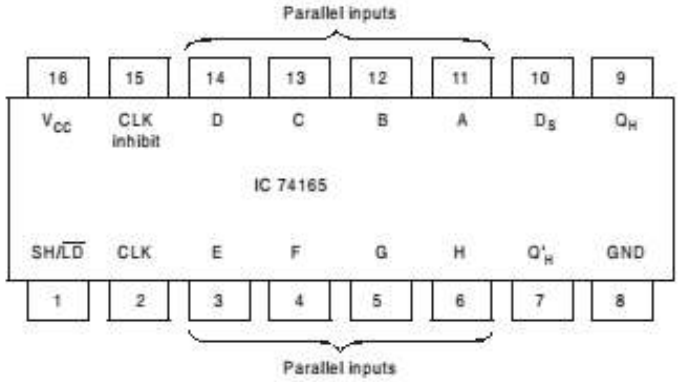
## 8-bit Parallel-in–Serial-out Shift Register

The pinout and logic diagram of IC 74165 is shown in Figure below. IC 74165 is an example of an 8-bit serial/parallel-in and serial-out shift register. The data can be loaded into the register in parallel and shifted out serially at QH using either of two clocks (CLK or CLK inhibit). It also contains a serial input, DS through which the data can be serially shifted in.

When the input *SHIFT* / $\overline{LOAD}$ ($SH$ / $\overline{LD}$) is LOW, it enables all the NAND gates for parallel loading. When an input data bit is a 0, the fl ip-fl op is asynchronously RESET by a LOW output of the lower NAND gate. Similarly, when the input data bit is a 1, the flip-flop is asynchronously SET by a LOW output of the upper NAND gate. The clock is inhibited during parallel loading operation. A HIGH on the *SHIFT* / $\overline{LOAD}$ input enables the clock causing the data in the register to shift right. With the low to high transitions of either clock, the serial input data (DS) are shifted into the 8-bit register.



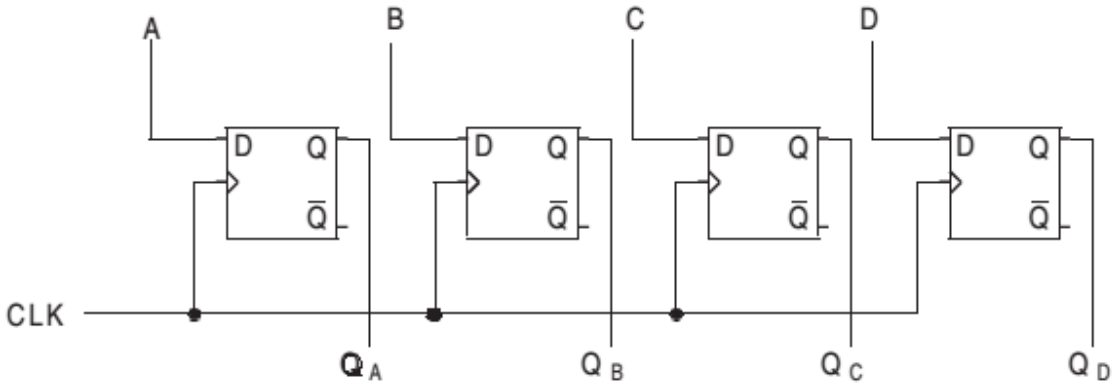

(a) Logic diagram.



(b) Pinout diagram of IC 74165.

# PARALLEL-IN–PARALLEL-OUT REGISTER

The parallel input of data has already been discussed in the preceding section of parallel-in–serial-out shift register. Also, in this type of register there is no interconnection between the flip-flops since no serial shifting is required. Hence, the moment the parallel entry of the data is accomplished the data will be available at the parallel outputs of the register. A simple parallel-in–parallel out shift register is shown in Figure below.




Here the parallel inputs to be applied at A, B, C, and D inputs are directly connected to the D inputs of the respective fl ip-fl ops. On applying the clock transitions, these inputs are entered into the register and are immediately available at the outputs $Q_1$, $Q_2$, $Q_3$, and $Q_4$.

# UNIVERSAL REGISTER

A register that is capable of transferring data in only one direction is called a '***unidirectional shift register***' whereas the register that is capable of transferring data in both left and right direction is called a '***bidirectional shift register***'. Now if the register has both the shift-right and shift-left capabilities, along with the necessary input and output terminals for parallel transfer, then it is called a ***shift register*** **with** ***parallel load*** or '***universal shift register***'.

The most general shift register has all the capabilities listed below. Others may have only some of these functions, with at least one shift operation.

1) A shift-right control to enable the shift-right operation and the serial input and output lines associated with the shift-right.
2) A shift-left control to enable the shift-left operation and the serial input and output lines associated with the shift-left.

3) A parallel-load control to enable a parallel transfer and the *n* input lines associated with the parallel transfer.
4) *n* parallel output lines.
5) A *clear* control to clear the register to 0.
6) A *CLK* input for clock pulses to synchronize all operations.
7) A control state that leaves the information in the register unchanged even though clock pulses are continuously applied.

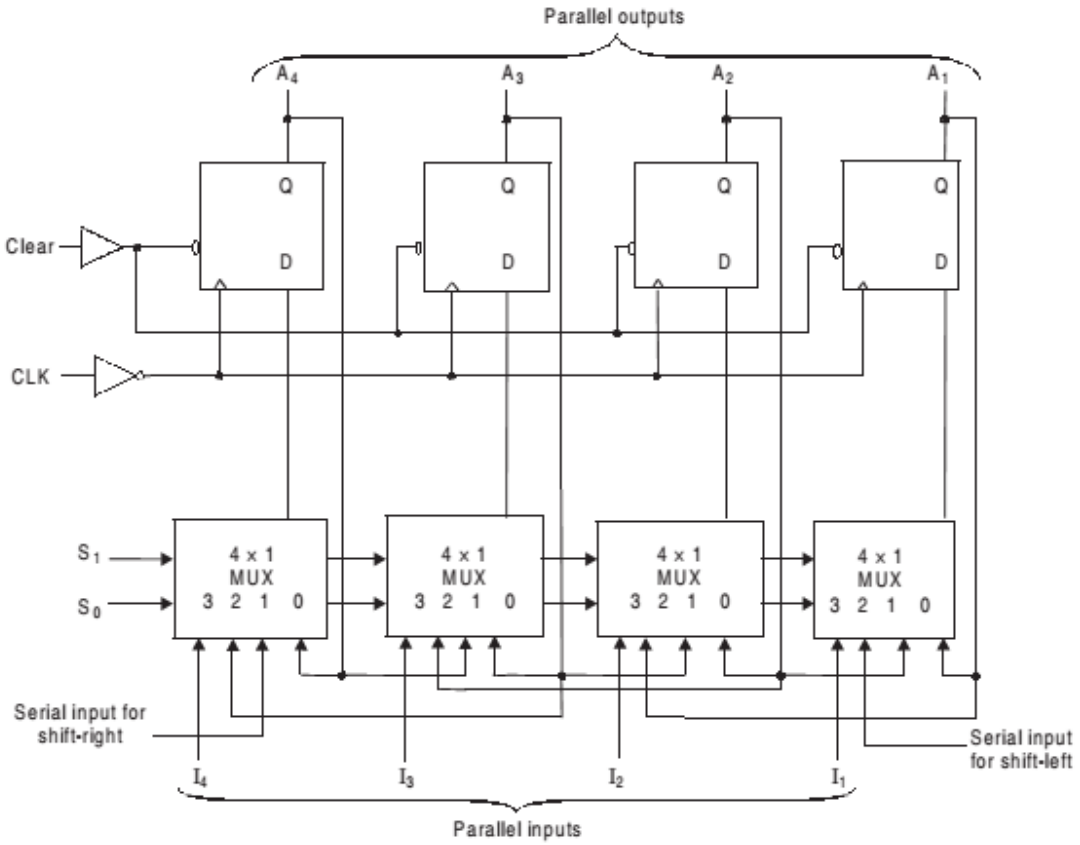



**Figure:-** 4-bit universal shift register.

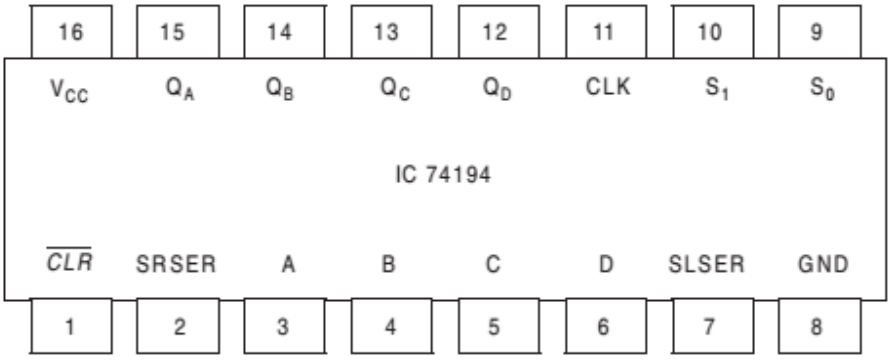
The diagram of a shift-register with all the capabilities listed above is shown in Figure above. This is similar to IC type 74194. Though it consists of four D flip-flops, S-R flip-flops can also be used with an inverter inserted between the S and R terminals. The four multiplexers drawn are also part of the register. The four multiplexers have two common selection lines $S_1$ and $S_0$. When $S_1S_0 = 00$, the input 0 is selected for each of the multiplexers. Similarly, when $S_1S_0 = 01$, the input 1, when $S_1S_0 = 10$, the input 2 and for $S_1S_0 = 11$, the input 3, is selected for each of the multiplexers.

The S1 and S0 inputs control the mode of operation of the register as specified in the entries of functions in the below Table. When $S_1S_0 = 00$, the present value of the register is applied to the D inputs of the flip-flops. Hence this condition forms a path from the output of each flip-flop into the input of the same flip-flop. The next clock pulse transition transfers into each flip-flop the binary value held previously & no change of state occurs. When $S_1S_0 = 01$, terminals 1 of each of the multiplexer inputs have a path to the D inputs of each of the flip-flops. This causes a shift-right operation, with the serial input transferred into flip-flop $A_4$. Similarly, with S1S0 = 10, a shift-left operation results, with the other serial input going into flip-flop $A_1$. Finally, when S1S0 = 11, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock pulse.

| *Mode control* | | *Register operation* |
|---|---|---|
| $S_1$ | $S_0$ | |
| 0 | 0 | No change |
| 0 | 1 | Shift-right |
| 1 | 0 | Shift-left |
| 1 | 1 | Parallel load |

**Table:-** Function table for the universal register

A universal register is a general-purpose register capable of performing three operations: shift-right, shift-left, and parallel load. Not all shift registers available in MSI circuits have all these capabilities. The particular application dictates the choice of one MSI circuit over another. As we have already mentioned IC 74194 is a 4-bit bidirectional shift register with parallel load. The pinout diagram of IC 74194 is shown in Figure below:-




The parallel loading of data is accomplished with a positive transition of the clock and by applying the four bits of data to the parallel inputs and a HIGH to the $S_1$ and $S_0$ inputs. Similarly, shift-right is accomplished synchronously with the positive edge of the clock when $S_0$ is HIGH and $S_1$ is LOW. In this mode the serial data is entered at the shift right serial input. In the same manner, when $S_0$ is LOW and $S_1$ is HIGH, data bits shift left synchronously with the clock pulse and new data is entered at the shift-left serial input.

# SHIFT REGISTER COUNTERS

Shift registers may be arranged to form different types of counters. These shift registers use *feedback*, where the output of the last flip-flop in the shift register is fed back to the first flip-flop. Based on the type of this feedback connection, the shift register counters are classified as (*i*) ring counter and (*ii*) twisted ring or Johnson or Shift counter.

## Ring Counter

It is possible to devise a counter-like circuit in which each flip-flop reaches the state Q = 1 for exactly one count, while for all other counts Q = 0. Then Q indicates directly an occurrence of the corresponding count. Actually, since this does not represent binary numbers, it is better to say that the outputs of the flip-flops represent a code. Such a circuit is shown in Figure below, which is known as a ***ring counter***. The Q output of the last stage in the shift register is fed back as the input to the first stage, which creates a ring-like structure. Hence a ring *counter* is a circular shift register with only one flip-flop being set at any particular time and all

others being cleared. The single bit is shifted from one flip-flop to the other to produce the sequence of timing signals. Such encoding where there is a single 1 and the rest of the code variables are 0, is called a *one-hot code*.

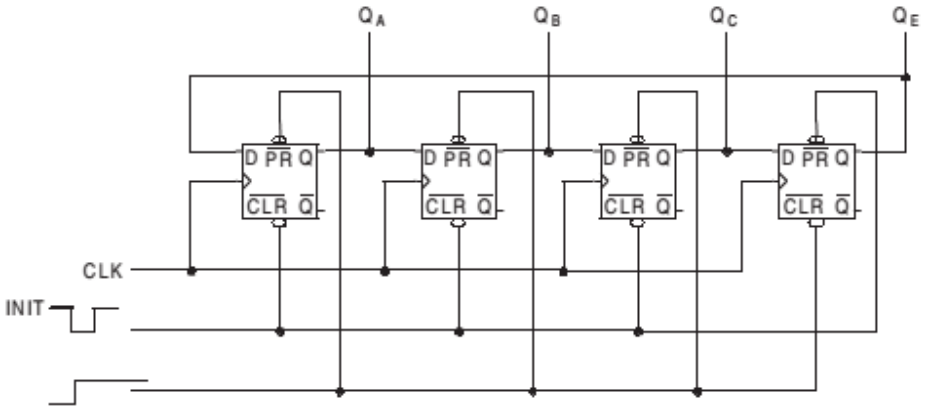



**Figure:-** A 4-bit ring counter using D flip-flops

The circuit shown in Figure above consists of four flip-flops and their outputs are $Q_A$,$Q_B$, $Q_C$, and $Q_E$ respectively. The PRESET input of the last flip-fl op and the CLEAR inputs of the other three flip-flops are connected together. Now, by applying a LOW pulse at this line, the last flip-flop is SET and all the others are RESET, *i.e.*, $Q_AQ_BQ_CQ_E = 0001$. Hence, from the circuit it is clear that $D_A = 1$, $D_B = 0$, $D_C = 0$, and $D_E = 0$. Therefore, when a clock pulse is applied, the 1st flip-flop is set to 1, while the other three flip-flops are reset to 0 *i.e.*, the output of the ring counter is $Q_AQ_BQ_CQ_E = 1000$. Similarly, when the 2nd clock pulse is applied, the 1 in the first flip-flop is shifted to the second flip-flop & the output of the counter becomes $Q_AQ_BQ_CQ_E = 0100$; on occurrence of the 3rd clock pulse, the output will be $Q_AQ_BQ_CQ_E = 0010$; on occurrence of the fourth clock pulse the output becomes $Q_AQ_BQ_CQ_E = 0001$, *i.e.*, the initial state. Thus, the 1 is shifted around the register as long as the clock pulses are applied. The truth table that describes the operation of the above 4-bit ring counter is shown in Table below:-

| *INIT* | *CLK* | $Q_A$ | $Q_B$ | $Q_C$ | $Q_E$ |
|---|---|---|---|---|---|
| L | X | 0 | 0 | 0 | 1 |
| H | ↑ | 1 | 0 | 0 | 0 |
| H | ↑ | 0 | 1 | 0 | 0 |
| H | ↑ | 0 | 0 | 1 | 0 |
| H | ↑ | 0 | 0 | 0 | 1 |

## Johnson Counter

A *k*-bit ring counter circulates a single bit among the flip-flops to provide *k* distinguishable states. The number of sates can be doubled if the shift register is connected as a *switch-tail* ring counter. A switch-tail ring counter is a circular shift register with the complement of the last flip-flop being connected to the input of the first flip-flop. Figure below shows such a type of shift register. The circular connection is made from the complement of the rightmost flip-flop to the input of the leftmost flip-flop. The register shifts its contents once to the right with every clock pulse, and at the same time, the complement value of the E flip-flop is transferred into the A flip-flop. Starting from a cleared state, the switch-tail ring counter goes through a sequence of eight states as listed in Table below. In general a *k*-bit switch-tail counter will go through 2k states. Starting with all 0s each shift operation inserts 1s from the left until the register is filled with all 1s. In the following sequences, 0s are inserted from the left until the register is again filled with all 0s.

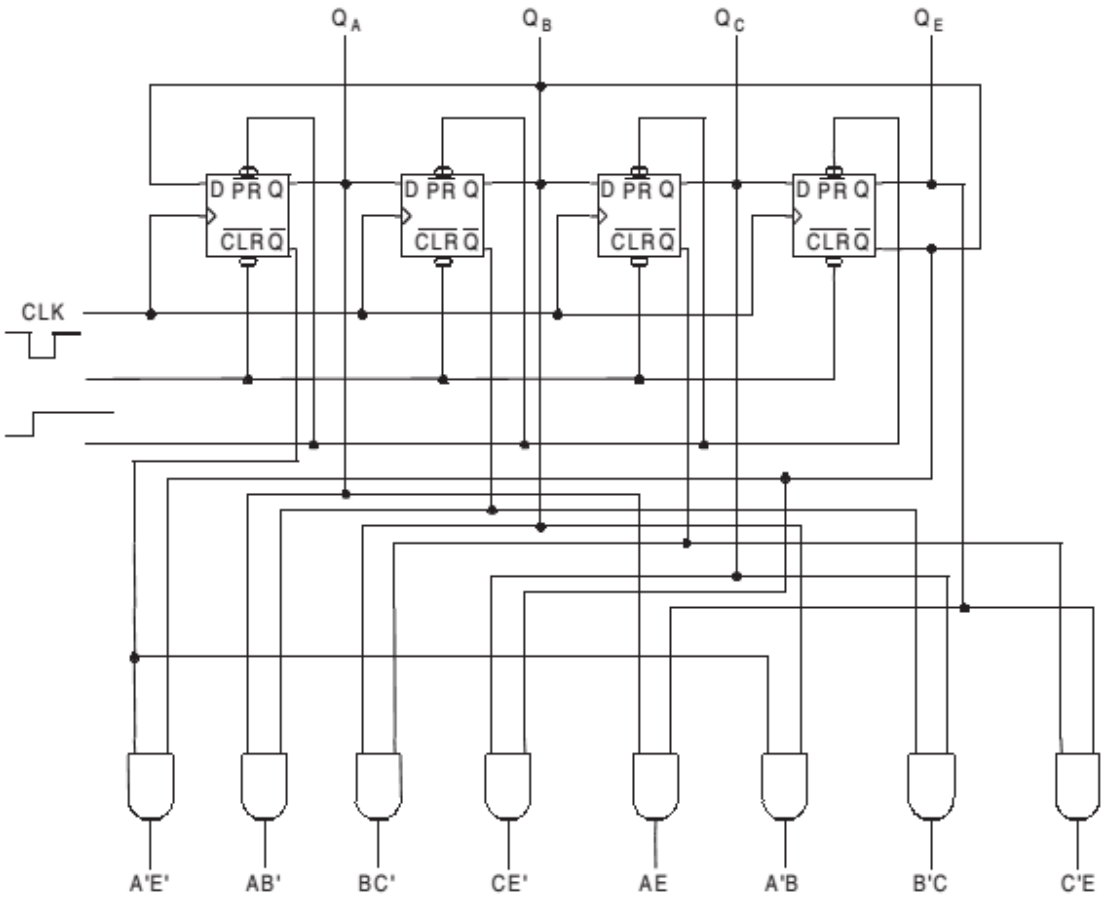



**Figure:-** A 4-bit Johnson counter using D flip-flops and decoding gates.

A *Johnson* or *moebius counter* is a switch-tail ring counter with 2k decoding gates to provide outputs for 2*k* timing signals. The decoding gates are also shown in Figure above. Since each gate is enabled during one particular state sequence, the outputs of the gates generate eight timing sequences in succession.

The decoding of a *k*-bit switch-tail ring counter to obtain 2*k* timing sequences follows a regular pattern. The all-0s state is decoded by taking the complement of the two extreme flip-flop outputs. The all-1s state is decoded by taking the normal outputs of the two extreme flip-flops. All other states are decoded from an adjacent 1, 0 or 0, 1 pattern in the sequence.
For example, sequence 6 has an adjacent 0 and 1 pattern in flip-flops A and B. the decoded output is then obtained by taking the complement A and the normal of B, or the A′B.

| *Sequence number* | *Flip-flop outputs* | | | |
|---|---|---|---|---|
| | *A* | *B* | *C* | *E* |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 |
| 4 | 1 | 1 | 1 | 0 |
| 5 | 1 | 1 | 1 | 1 |
| 6 | 0 | 1 | 1 | 1 |
| 7 | 0 | 0 | 1 | 1 |
| 8 | 0 | 0 | 0 | 1 |

**Table :-** Count sequence of a 4-bit Johnson counter

One disadvantage of the circuit in Figure 8.16 is that, if it finds itself in an unused state, it will persist in moving from one invalid state to another and never find its way to a valid state. The difficulty can be corrected by modifying the circuit to avoid this undesirable condition. One correcting procedure is to disconnect the output from flip-flop B that goes to the D input of flip-flop C, and instead enable the input of flip-flop C by the function:

**DC = (A + C)B,** where DC is the flip-flop input function for the D input of the flip-flop C.

Johnson counters can be constructed for any number of timing sequences. The number of flip-flops needed is one-half the number of timing signals. The number of decoding gates is equal to the number of timing sequences and only 2-input gates are employed. Ring counter does not require any decoding gates, since in ring counter only one flip-flop will be in the set condition at any time.

## Asynchronous and Synchronous Shift Registers

Asynchronous circuits changes state each time the input changes the state, while synchronous circuit changes state only when triggered by a momentary change in the input signal. This momentary change is called triggering.

Shift registers are made of flip flops and their operation depends upon the state at the flip flops. Flip flops changes their states due to triggering when flip flop change their state on the base of input pulse then it is called Edge triggering. In edge triggering flip flop change its state on the basses of Leading edge or trailing edge. When flip flop works on the bases of change in DC level, that is called Asynchronous Triggering. And the shift registers work on this principle is called Asynchronous shift registers. On the other hand, shift registers changes their state only when triggered by clock pulse are called Synchronous shift registers these type of shift registers usually used in counters.

# Counters

Counting is frequently required in digital computers and other digital systems to record the number of events occurring in a specified interval of time. Normally an electronic counter is used for counting the number of pulses coming at the input line in a specified time period. The counter must possess memory since it has to remember its past states. As with other sequential logic circuits counters can be synchronous or asynchronous.

As the name suggests, it is a circuit which counts. The main purpose of the counter is to record the number of occurrence of some input. There are many types of counter both binary and decimal. Commonly used counters are

1. Binary Ripple Counter
2. Ring Counter
3. BCD Counter
4. Decade counter
5. Up down Counter
6. Frequency Counter

# Ripple Counter

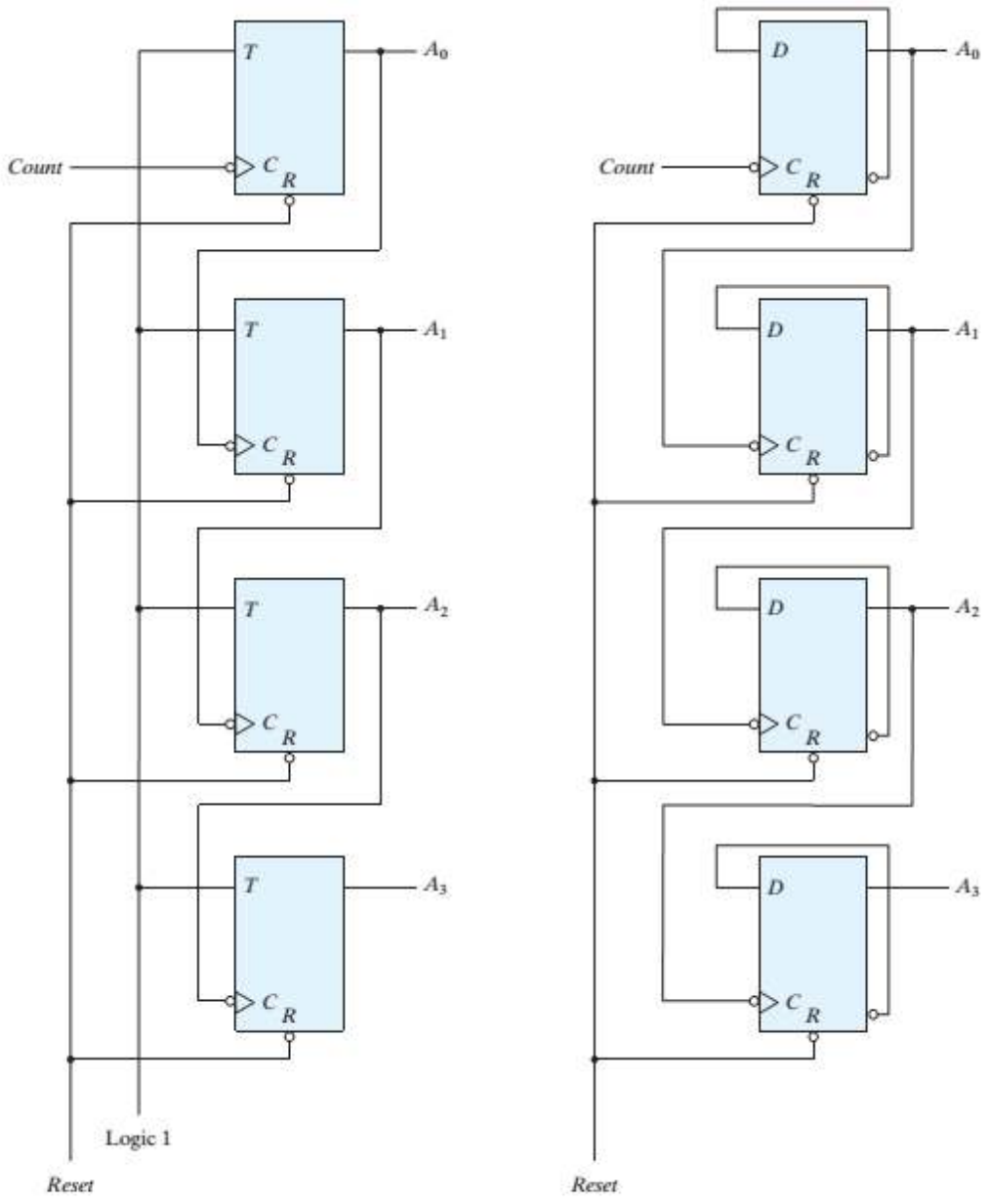
A counter that follows the binary number sequence is called a binary counter. An n -bit binary counter consists of n flip-flops and can count in binary from 0 through 2n - 1. Counters are available in two categories: ripple counters and synchronous counters. In a ripple counter, a flip-flop output transition serves as a source for triggering other flip-flops. In other words, the C input of some or all flip-flops are triggered, not by the common clock pulses, but rather by the transition that occurs in other flip-flop outputs. In a synchronous counter, the C inputs of all flip-flops receive the common clock.

# Binary Ripple Counter

A binary ripple counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the *C* input of the next higher order flip-flop. The flip-flop holding the least significant bit receives the incoming count pulses. A complementing flip-flop can be obtained from a *JK* flip-flop with the *J* and *K* inputs tied together or from a *T* flip-flop. A third possibility is to use a *D* flip-flop with the complement output connected to the *D* input. In this way, the *D* input is always the complement of the present state, and the next clock pulse will cause the flip-flop to complement. The logic diagram of two 4-bit binary ripple counters is shown in Fig. below . The output of each flip-flop is connected to the *C* input of the next flip-flop in sequence. The flip-flop holding the least significant bit receives the incoming count pulses. The *T* inputs of all the flip-flops in (a) are connected to a permanent logic 1, making each flip-flop complement if the signal in its *C* input goes through a negative transition. The bubble in front of the dynamic indicator symbol next to *C* indicates that the flip-flops respond to the negative-edge transition of the input. The negative transition occurs when the output of the previous flip-flop to which *C* is connected goes from 1 to 0.

The count starts with binary 0 and increments by 1 with each count pulse input. After the count of 15, the counter goes back to 0 to repeat the count. The least significant bit,$A_0$, is complemented with each count pulse input. Every time that $A_0$ goes from 1 to 0, it complements$A_1$. Every time that$A_1$ goes from 1 to 0, it

complements $A_2$. Every time that $A_2$ goes from 1 to 0, it complements $A_3$, and so on for any other higher order bits of a ripple counter. For example, consider the transition from count 0011 to 0100. $A_0$ is complemented with the count pulse. Since $A_0$ goes from 1 to 0, it triggers $A_1$ and complements it. As a result, $A_1$ goes from 1 to 0, which in turn complements $A_2$, changing it from 0 to 1. $A_2$ does not trigger $A_3$, because $A_2$ produces a positive transition and the flip-flop responds only to negative transitions. Thus, the count from 0011 to 0100 is achieved by changing the bits one at a time, so the count goes from 0011 to 0010, then to 0000, and finally to 0100. The flip-flops change one at a time in succession, and the signal propagates through the counter in a ripple fashion from one stage to the next.



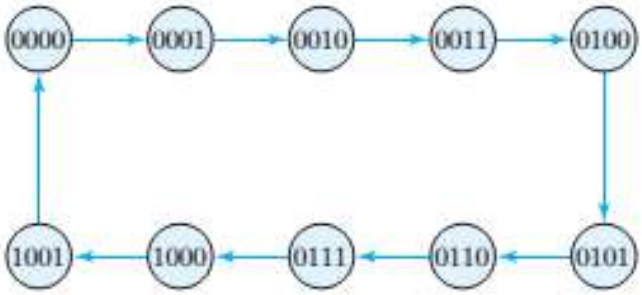

A binary counter with a reverse count is called a *binary countdown counter*. In a countdown counter, the binary count is decremented by 1 with every input count pulse. The count of a four-bit countdown counter starts from binary 15 and continues to binary counts 14, 13, 12, . . . , 0 and then back to 15. A list of the count sequence of a binary countdown counter shows that the least significant bit is complemented with every count pulse. Any other bit in the sequence is complemented if its previous least significant bit goes from 0 to 1. Therefore, the

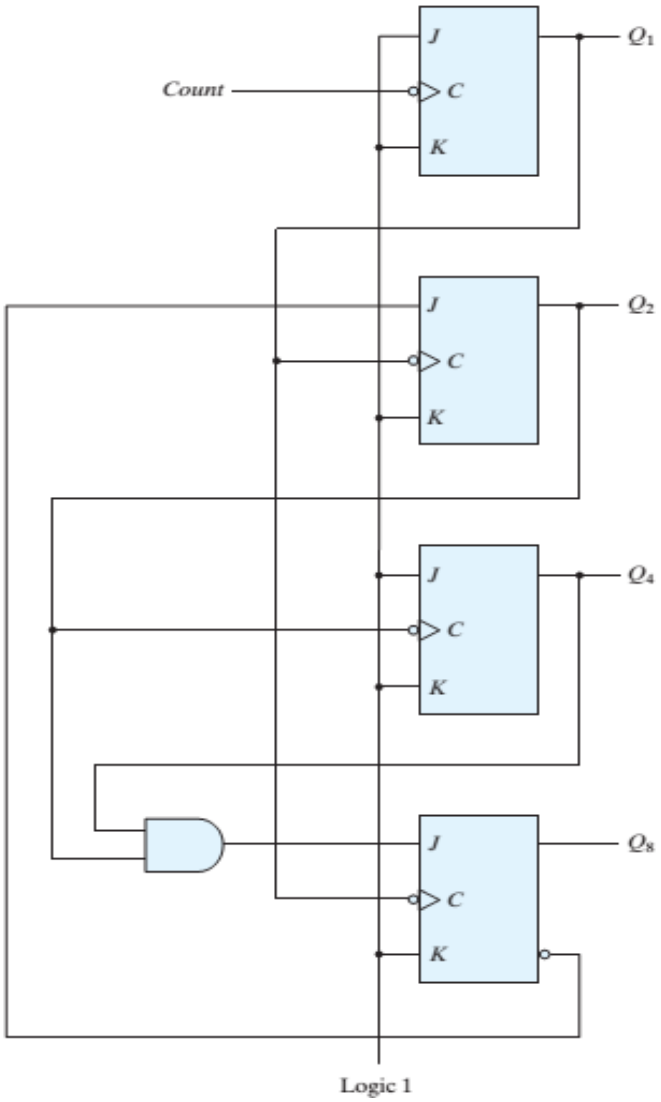
diagram of a binary countdown counter looks the same as the binary ripple counter in Fig. above , provided that all flip-flops trigger on the positive edge of the clock. (The bubble in the *C* inputs must be absent.) If negative-edge-triggered flip-flops are used, then the *C* input of each flip-flop must be connected to the complemented output of the previous flip-flop. Then, when the true output goes from 0 to 1, the complement will go from 1 to 0 and complement the next flip-flop as required.

## BCD Ripple Counter

A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9.Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits. The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit. If BCD is used, the sequence of states is as shown in the state diagram in the side Fig.:-

**(Fig:-** State diagram for a decimal BCD counter)



A decimal counter is similar to a binary counter, except that the state after 1001 (the code for decimal digit 9) is 0000 (the code for decimal digit 0).

The logic diagram of a BCD ripple counter using *JK* flip-flops is shown in Figure below. The four outputs are designated by the letter symbol *Q*, with a numeric subscript equal to the binary weight of the corresponding bit in the BCD code. Note that the output of $Q_1$ is applied to the *C* inputs of both $Q_2$ and $Q_8$ and the output of $Q_2$ is applied to the *C* input of$Q_4$. The *J* and *K* inputs are connected either to a permanent 1 signal or to outputs of other flip-flops.

A ripple counter is an asynchronous sequential circuit. Signals that affect the flip-flop transition depend on the way they change from 1 to 0. The operation of the counter can be explained by a list of conditions for flip-flop transitions. These conditions are derived from the logic diagram and from knowledge of how a *JK* flip-flop operates. Remember that when the *C* input goes from 1 to 0, the flip-flop is set if $J = 1$, is cleared if $K = 1$, is complemented if $J = K = 1$, and is left unchanged if $J = K = 0$.

To verify that these conditions result in the sequence required by a BCD ripple counter, it is necessary to verify that the flip-flop transitions indeed follow a sequence of states as specified by the state diagram as mentioned above .$Q_1$ changes state after each clock pulse.$Q_2$ complements every time$Q_1$ goes from 1 to 0, as long as $Q_8 =$ 0. When$Q_8$ becomes 1,$Q_2$ remains at 0.$Q_4$ complements every time$Q_2$ goes from 1 to 0.$Q_8$ remains at 0 as long as$Q_2$ or$Q_4$ is 0. When both$Q_2$ and$Q_4$ become 1,$Q_8$ complements when$Q_1$ goes from 1 to 0.$Q_8$ is cleared on the next transition of$Q_1$.

The BCD counter of Fig. above is a *decade* counter, since it counts from 0 to 9. To count in decimal from 0 to 99, we need a two-decade counter. To count from 0 to 999, we need a three-decade counter. Multiple decade counters can be constructed by connecting BCD counters in cascade, one for each decade. A three-decade counter is shown in Fig. below:-

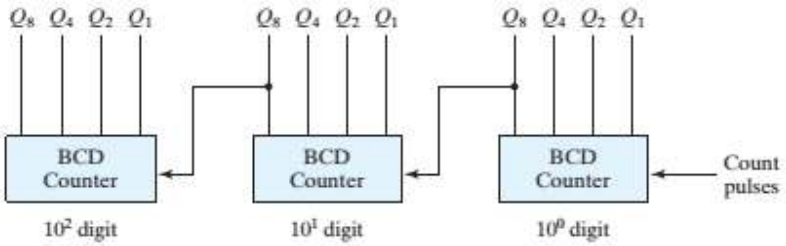



**Fig: -** Block diagram of a three-decade decimal BCD counter

The inputs to the second and third decades come from$Q_8$ of the previous decade. When$Q_8$ in one decade goes from 1 to 0, it triggers the count for the next higher order decade while its own decade goes from 9 to 0.

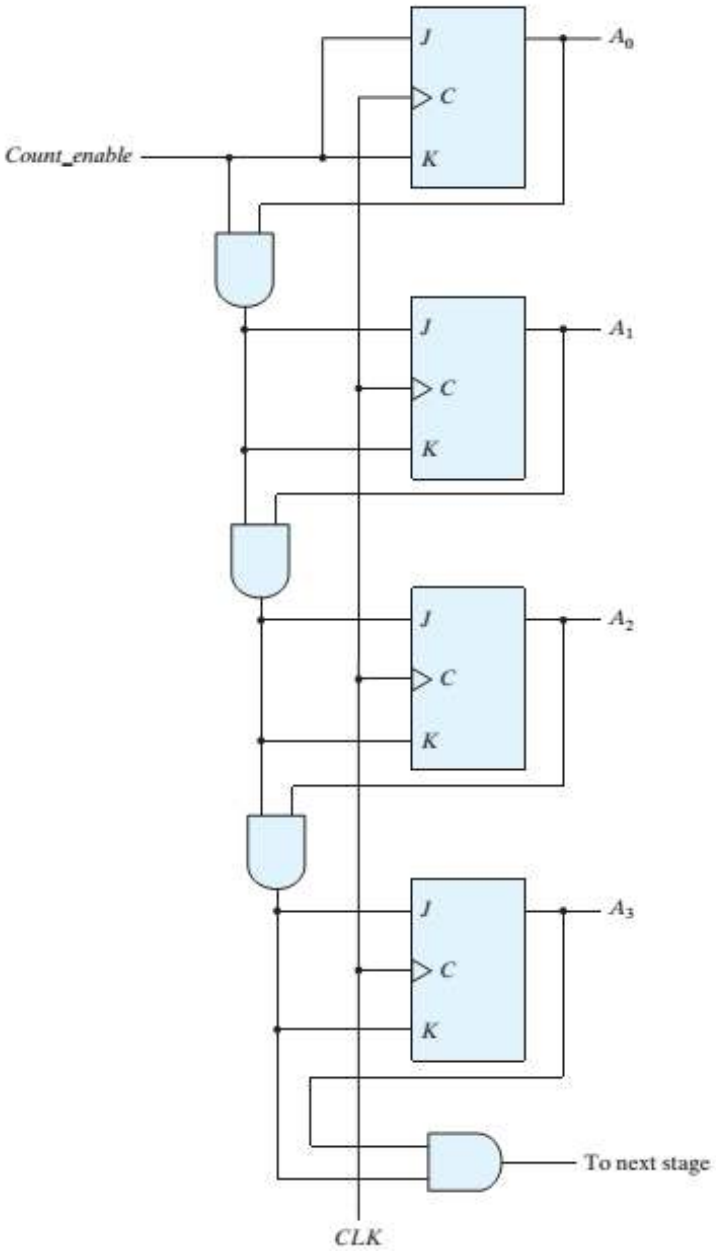
## Synchronous counters

Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops. A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter. The decision whether a flip-flop is to be complemented is determined from the values of the data inputs, such as *T* or *J* and *K* at the time of the clock edge. If $T = 0$ or $J = K = 0$, the flip-flop does not change state. If $T = 1$ or $J = K = 1$, the flip-flop complements. Here we present some typical synchronous counters and explain their operation.

# Binary Counter

The design of a synchronous binary counter is so simple that there is no need to go through a sequential logic design process. In a synchronous binary counter, the flip-flop in the least significant position is complemented with every pulse. *A flip-flop in any other position is complemented when all the bits in the lower significant positions are equal to 1.* For example, if the present state of a four-bit counter is$A_3A_2A_1A_0$ = 0011, the next count is 0100.$A0$ is always complemented.$A1$ is complemented because the present state of $A0$ = 1.$A_2$ is complemented because the present state of $A_1A_0$ = 11. However, $A_3$ is not complemented, because the present state of $A_2A_1A_0$ = 011, which does not give an all-1's condition.

Synchronous binary counters have a regular pattern and can be constructed with complementing flip□flops and gates. The regular pattern can be seen from the 4□bit counter depicted in Fig. below.



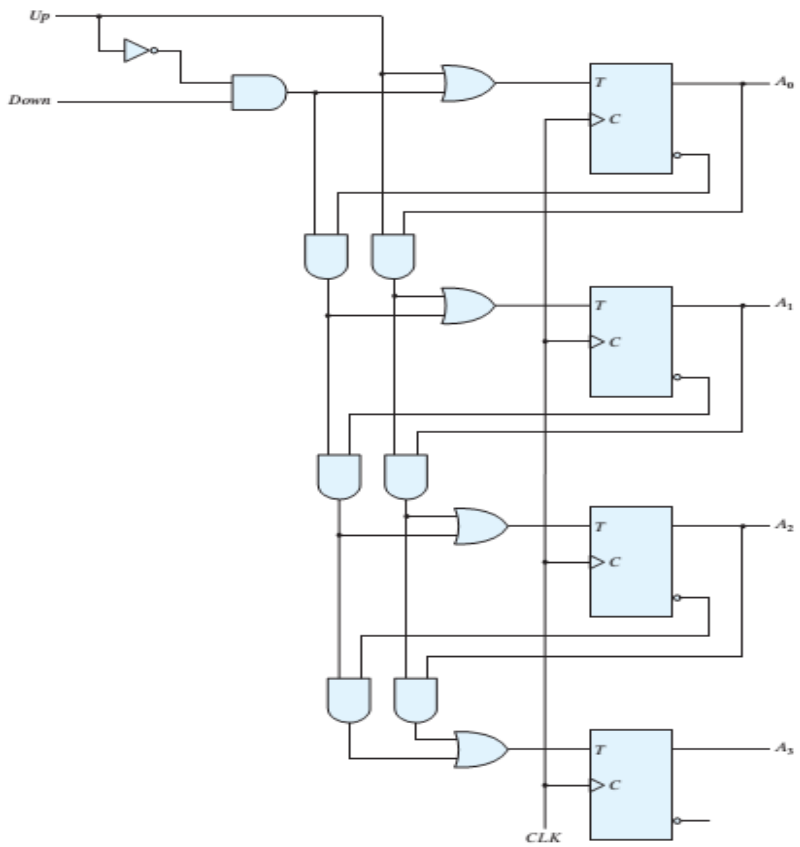

( **Figure: -** Four□bit synchronous binary counter )

The *C* inputs of all flip□flops are connected to a common clock. The counter is enabled by *Count enable*. If the enable input is 0, all *J* and *K* inputs are equal to 0 and the clock does not change the state of the counter. The first stage, *A*0, has its *J* and *K* equal to 1 if the counter is enabled. The other *J* and *K* inputs are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled. The chain of AND gates generates the required logic for the *J* and *K* inputs in each stage. The counter can be extended to any number of stages, with each stage having an additional flip□flop and an AND gate that gives an output of 1 if all previous flip□flop outputs are 1.

Note that the flip□flops trigger on the positive edge of the clock. The polarity of the clock is not essential here, but it is with the ripple counter. The synchronous counter can be triggered with either the positive or the negative clock edge. The complementing flip□flops in a binary counter can be of either the *JK* type, the *T* type, or the *D* type with XOR gates.

## Up–Down Binary Counter

A synchronous countdown binary counter goes through the binary states in reverse order, from 1111 down to 0000 and back to 1111 to repeat the count. It is possible to design a countdown counter in the usual manner, but the result is predictable by inspection of the downward binary count. The bit in the least significant position is complemented with each pulse. *A bit in any other position is complemented if all lower significant bits are equal to 0.* For example, the next state after the present state of 0100 is 0011. The least significant bit is always complemented. The second significant bit is complemented because the first bit is 0. The third significant bit is complemented because the first two bits are equal to 0. But the fourth bit does not change, because not all lower significant bits are equal to 0.A countdown binary counter can be constructed as shown in previous Fig., except that the inputs to the AND gates must come from the complemented outputs, instead of the normal outputs, of the previous flip□flops. The two operations can be combined in one circuit to form a counter capable of counting either up or down. The circuit of a 4bit up–down binary counter using *T* flip□flops is shown in Fig. below.



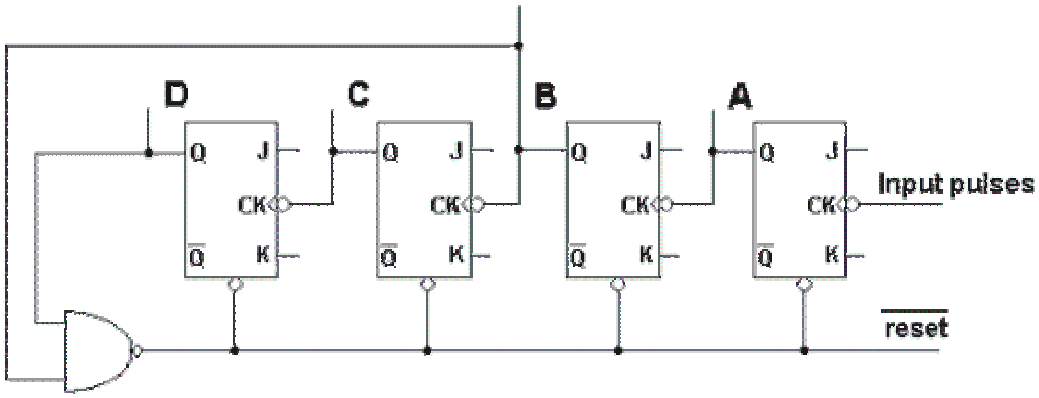
It has an up control input and a down control input. When the up input is 1, the circuit counts up, since the *T* inputs receive their signals from the values of the previous normal outputs of the flip□flops. When the down input is 1 and the up input is 0, the circuit counts down, since the complemented outputs of the previous flip□flops are applied to the *T* inputs. When the up and down inputs are both 0, the circuit does not change state and remains

## Decade Counter

A decade counter is the one which goes through 10 unique combinations of outputs and then resets as the clock proceeds. We may use some sort of a feedback in a 4-bit binary counter to skip any six of the sixteen possible output states from 0000 to 1111 to get to a decade counter. A decade counter does not necessarily count from 0000 to 1001 it could count as 0000,0001, 0010, 1000, 1001, 1010, 1011, 1110, 1111, 0000, 0001 and so on.

Figure below shows a decade counter having a binary count that is always equivalent to the input pulse count. The circuit is essentially, a ripple counter which count up to 16. We desire however, a circuit operation in which the count advance from 0 to 9 and then reset to 0 for a new cycle. This reset is a accomplished at the desired count as follows.

1. With counter REST count = 0000 the counter is ready to stage counter cycle.
2. Input pulses advance counter in binary sequence up to count of a (count = 1001)
3. The next count pulse advance the count to 10 count = 1010. A logic NAND gate decodes the count of 10 providing a level change at that time to trigger the one shot unit which then resets all counter stages. Thus, the pulse after the counter is at count = 9, effectively results in the counter going to count = 0.




**( Figure:** Decade Counter **)**

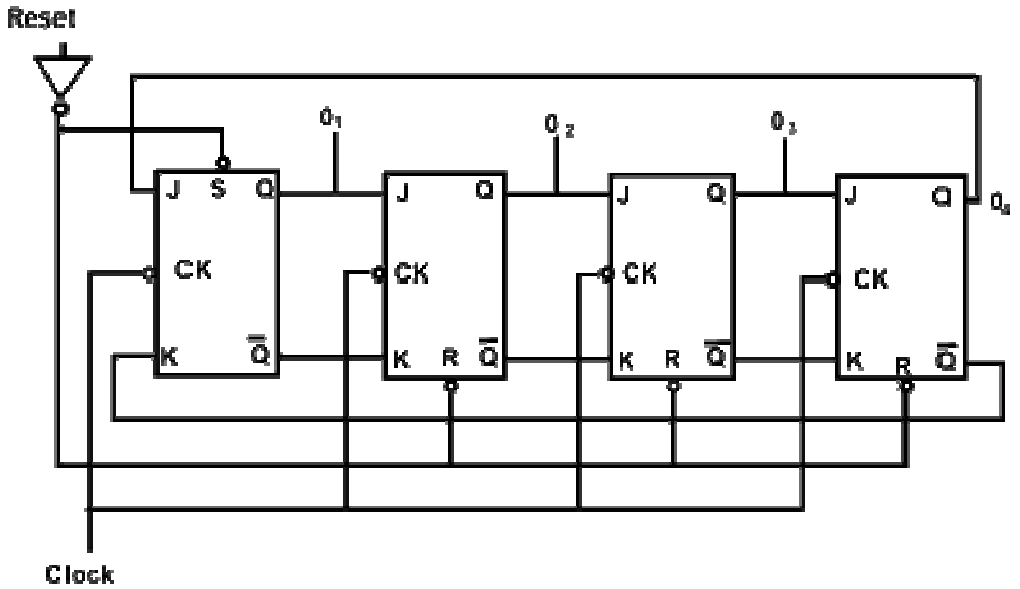
Table below provides a count table showing the binary count equivalent to the decimal count of input pulses. The table also shows that the count goes momentarily count from nine (1001) to ten (1010) before resetting to zero(0000). The NAND gate provides an output of 1 until the count reach ten. The count of ten is decoded (or sensed in this case ) by using logic inputs that are all 1 at the count of ten. When the count becomes ten the NAND gate output goes to logical 0, providing a 1 to 0 logic change to trigger the one shot unit, which then provides a short pulse to reset all counter stages.

The Q signal is used since it is normally high and goes low during the one shot timing period the flip flop in this circuit being reset by a low signal level (active low clearing). The one shot pulse need only be long enough so that slowest counter stage resets. Actually, at this time only the $2^1$and $2^3$ stage need be reset, but all stages are reset to insure that a new cycle at the count 0000.

| Table : Decade Counter Truth Table | | | | |
|---|---|---|---|---|
| **Input Pulses** | **D** | **C** | **B** | **A** |
| **0** | **0** | **0** | **0** | **0** |
| **1** | **0** | **0** | **0** | **1** |
| **2** | **0** | **0** | **1** | **0** |
| **3** | **0** | **0** | **1** | **1** |
| **4** | **0** | **1** | **0** | **0** |
| **5** | **0** | **1** | **0** | **1** |
| **6** | **0** | **1** | **1** | **0** |
| **7** | **0** | **1** | **1** | **1** |
| **8** | **1** | **0** | **0** | **0** |
| **9** | **1** | **0** | **0** | **1** |
| **10** | **1** | **0** | **1** | **0** |
| **0** | **0** | **0** | **0** | **0** |

## Ring Counter

The ring counter is the simplest example of a shift register. The simplest counter is called a Ring counter. The ring counter contains only one logical 1 or 0 which it circulates. The total cycle length is equal to the number of stages. The ring counter is useful in applications where count has to be recognized in order to perform some other logical operation. Since only one output is ever at logic 1 at given time extra logic gates are not required to decode the counts and the flip flop outputs may be used directly to perform the required operation.



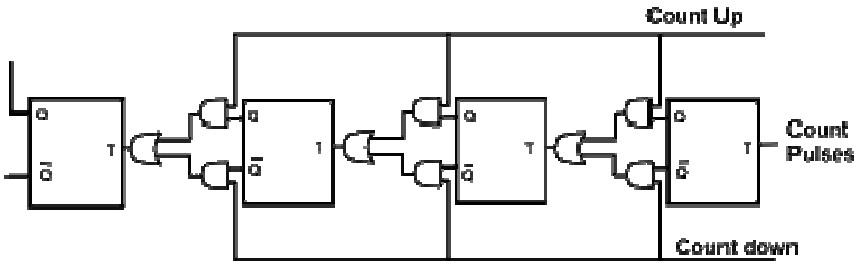

(**Figure:** Simple Ring Counter)

Note that in the above diagram the Reset will reset $Q_2$, $Q_3$ and $Q_4$ but will put $Q_1$ to a logic 1 state. This 1 will circulate when clock pulses are applied.

| Table: Ring Counter Truth Table | | | | |
|---|---|---|---|---|
| **Clock** | **$0_1$** | **$0_2$** | **$0_3$** | **$0_4$** |
| 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 |

## Up-Down Counter

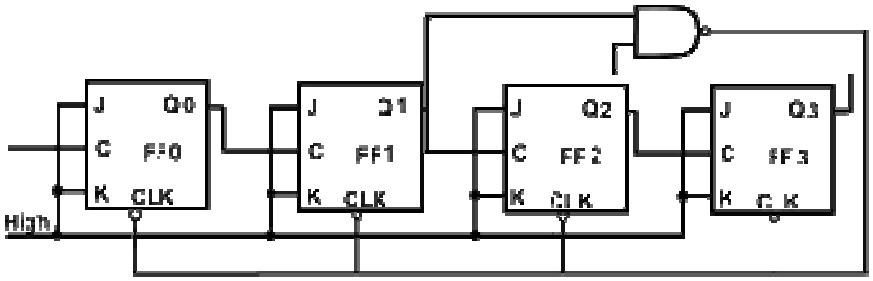
An up down counter is a bi-directional counter and it can be made to count upwards as well as downwards. In other words an up down counter is one which can provide both count up and down counts operations in a single unit. In the previous section it was seen that if triggering pulses are obtained from $\overline{Q}$ output the counter is a count up and if the triggering pulses are obtained from $\overline{Q}$ outputs, the counter is a countdown. Figure below gives an up down counter. When the count up signal is high the AND gate connecting Q output and count up signal gives and output 1 which passes through the OR gate to trigger the next flip flop. This results in the count up operation. Similarly a signal from countdown line will result the circuit to act as a down counter.




**(Figure:** Up Down Counter)

# BCD Counter

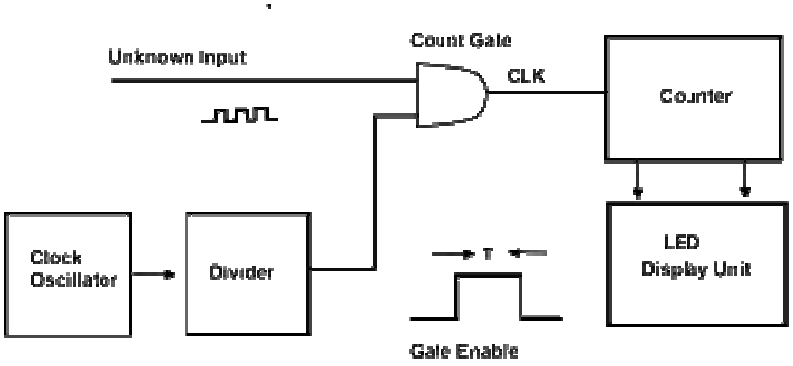
It is a special case of a decade counter in which the counter counts 0000 to 1001 and then resets. The output weights of the flip flops in these counters are in accordance with 8421 code. For instance, at the end of seventh clock pulse, the output sequence will be 0111 (Decimal equivalent of 0111 as per 8421 code is 7). These counters will thus be different from other decade counters that provide the same count by using some kind of forced feedback to skip some of the natural binary counts Figure below shows a counter of the BCD type.




**(Figure:** BCD Counter)

# Frequency Counter

Frequency counter is a digital device which can be used to measure the frequency of the periodic waveforms. The block diagram of frequency counter is shown in Figure below.




**(Figure:** Frequency Counter)

A signal having time period t applied at one of the input terminal of AND gate. While a unknown signal is also applied at the other input terminal of the AND gate. So, it is used as a clock for counter indicates the frequency of the unknown signal in respect to this time period. The time interval of the counter may be called contents. Let us suppose that time period of gate signal is one second and unknown signal is a square wave of 250 Hertz. In this condition counter counts 250 at the end of one second. This will be frequency of unknown signal.

# MODULE-4

# Memory and programmable logic

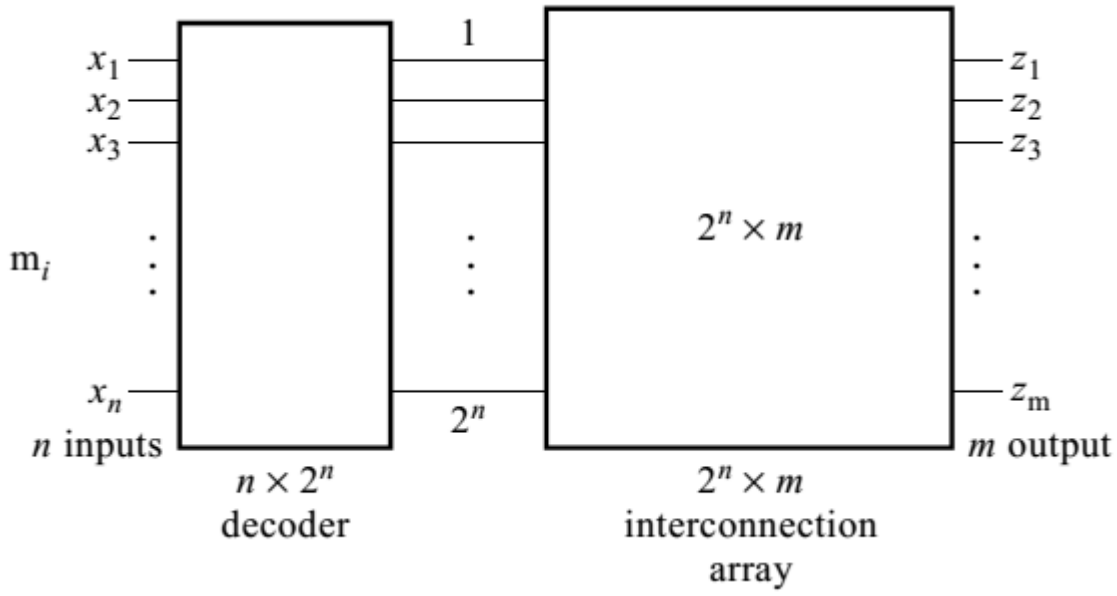
## READ-ONLY MEMORY (ROM)

A read-only memory (ROM) is a device that includes both the decoder and the OR gates within a single IC package. The connections between the outputs of the decoder and the inputs of the OR gates can be specified for each particular configuration.

A ROM is essentially a memory (or storage) device in which permanent binary information is stored. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern. ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established, it stays within the unit even when power is turned off and on again.

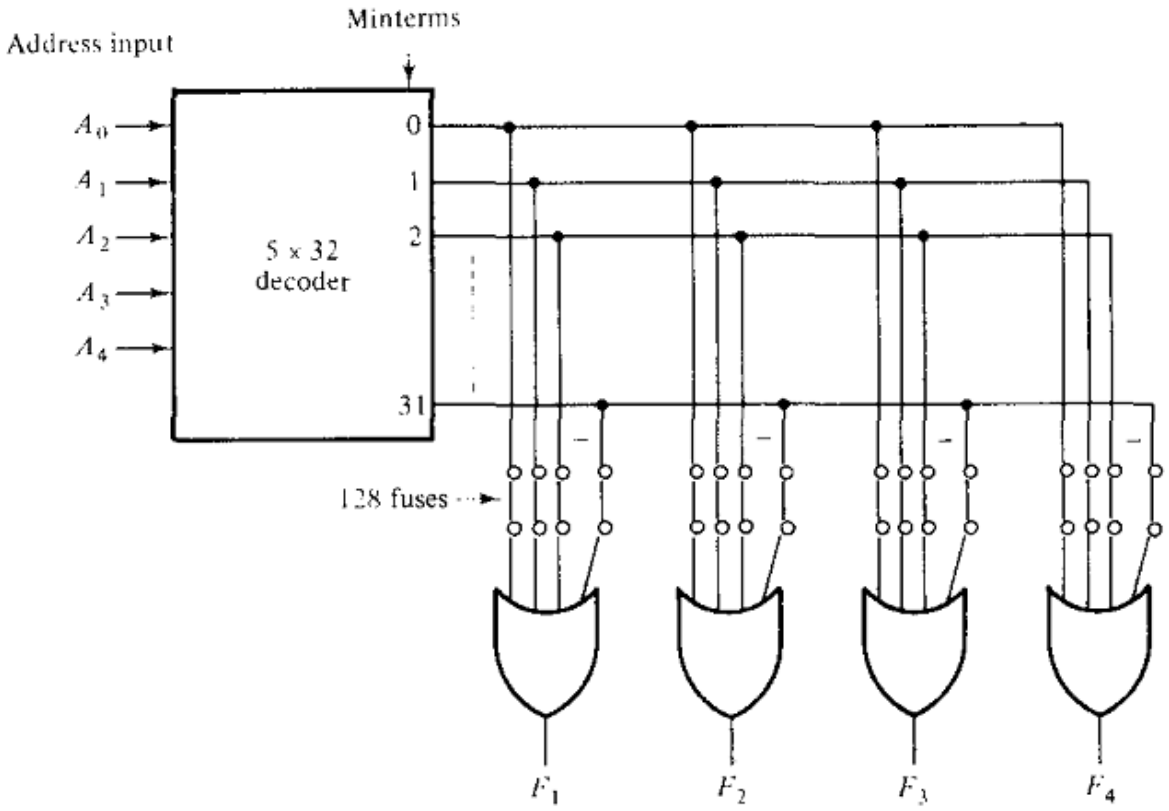
It consists of n input lines and m output lines. Each bit combination of the input variables is called an address. Each bit combination that comes out of the output lines is called a word. The number of bits per word is equal to the number of output lines, m. An address is essentially a binary number that denotes one of the min terms of n variables. The number of distinct addresses possible with n input variables is $2^n$. An output word can be selected by a unique address, and since there are $2^n$ distinct addresses in a ROM, there are $2^n$ distinct words that are said to be stored in the unit.

The word available on the output lines at any given time depends on the address value applied to the input lines. A ROM is characterized by the number of words $2^n$ and the number of bits per word m.




Basic structure of ROM

Consider a 32 x 8 ROM. The unit consists of 32 words of 8 bits each. This means that there are eight output lines and that there are 32, distinct words stored in the unit, each of which may be applied to the output lines. The particular word selected that is presently available on the output lines is determined from the five input lines. There are only five inputs in a 32 x 8 ROM because $2^5 = 32$, and with five variables, we can specify 32 addresses or min terms. For each address input, there is a unique selected word. Thus, if the input address is 00000, word number 0 is selected and it appears on the output lines. If the input address is 11111, word number 31 is selected and applied to the output lines. In between, there are 30 other addresses that can select the other 30 words.




# ROM Application:-

Preprogrammed toy circuit,

- Preprogrammed robot circuit,
- Standard look up table,
- Arithmetic function table generator,
- User defined code generator,
- Character generator,
- Printable or displayable fonts table

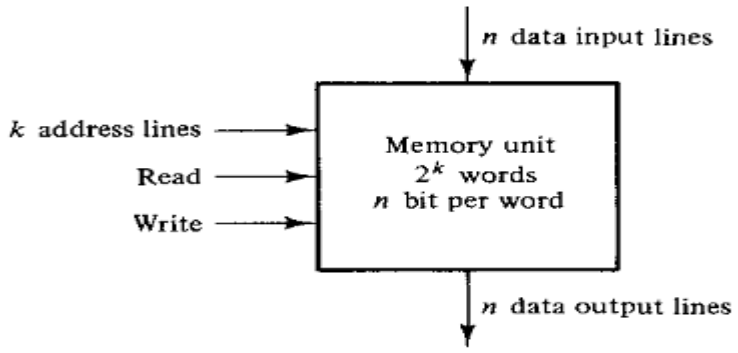
# RANDOM-ACCESS MEMORY (RAM):-

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the device. Memory cells can be accessed for information transfer to or from any desired random location and hence the name random access memory, abbreviated RAM.

A memory unit stores binary information in groups of bits called words. A word in memory is an entity of bits that move in and out of storage as a unit. A memory word is a group of 1's and 0's and may represent a number, an instruction, one or more alphanumeric characters, or any other binary-coded information. A group of eight bits is called a byte. Most computer memories use words that are multiples of 8 bits in length. Thus, a 16-bit word contains two bytes, and a 32-bit word is made up of four bytes. The capacity of a memory unit is usually stated as the total number of bytes that it can store.

The communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer .The n data input lines provide the information to be stored in memory and the n data output lines supply the information coming out of memory. The k address lines specify the particular word chosen among the many available. The two control inputs specify the direction of transfer desired: The write input causes binary data to be transferred into the memory, and the read input causes binary data to be transferred out of memory.

The memory unit is specified by the number of words it contains and the number of bits in each word. The address lines select one particular word. Each word in memory is assigned an identification number, called an address, starting from 0 and continuing with 1, 2, 3,upto $2^k - 1$, where k is the number of address lines. The selection of a specific word inside the memory is done by applying the k-bit binary address to the address lines.




Basic structure of RAM

| Memory address | | |
|---|---|---|
| Binary | decimal | Memory content |
| 0000000000 | 0 | 1011010101011101 |
| 0000000001 | 1 | 1010101110001001 |
| 0000000010 | 2 | 0000110101000110 |
| | ⋮ | ⋮ |
| 1111111101 | 1021 | 1001110100010100 |
| 1111111110 | 1022 | 0000110100011110 |
| 1111111111 | 1023 | 1101111000100101 |

Content of a 1024 x 16 memory

## Write and Read Operations

The two operations that a random-access memory can perform are the write and read operations. The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function. The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

1. Transfer the binary address of the desired word to the address lines.

2. Transfer the data bits that must be stored in memory to the data input lines.

3. Activate the write input.

**Control Inputs to Memory Chip**

| Memory Enable | Read/Write | Memory Operation |
|---|---|---|
| 0 | *X* | None |
| 1 | 0 | Write to selected word |
| 1 | 1 | Read from selected word |

The memory unit will then take the bits from the input data lines and store them in the word specified by the address lines. The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Transfer the binary address of the desired word to the address lines.

2. Activate the read input.

The memory unit will then take the bits from the word that has been selected by the address and apply them to the output data lines. The content of the selected word does not change after reading.

## Types of Memories

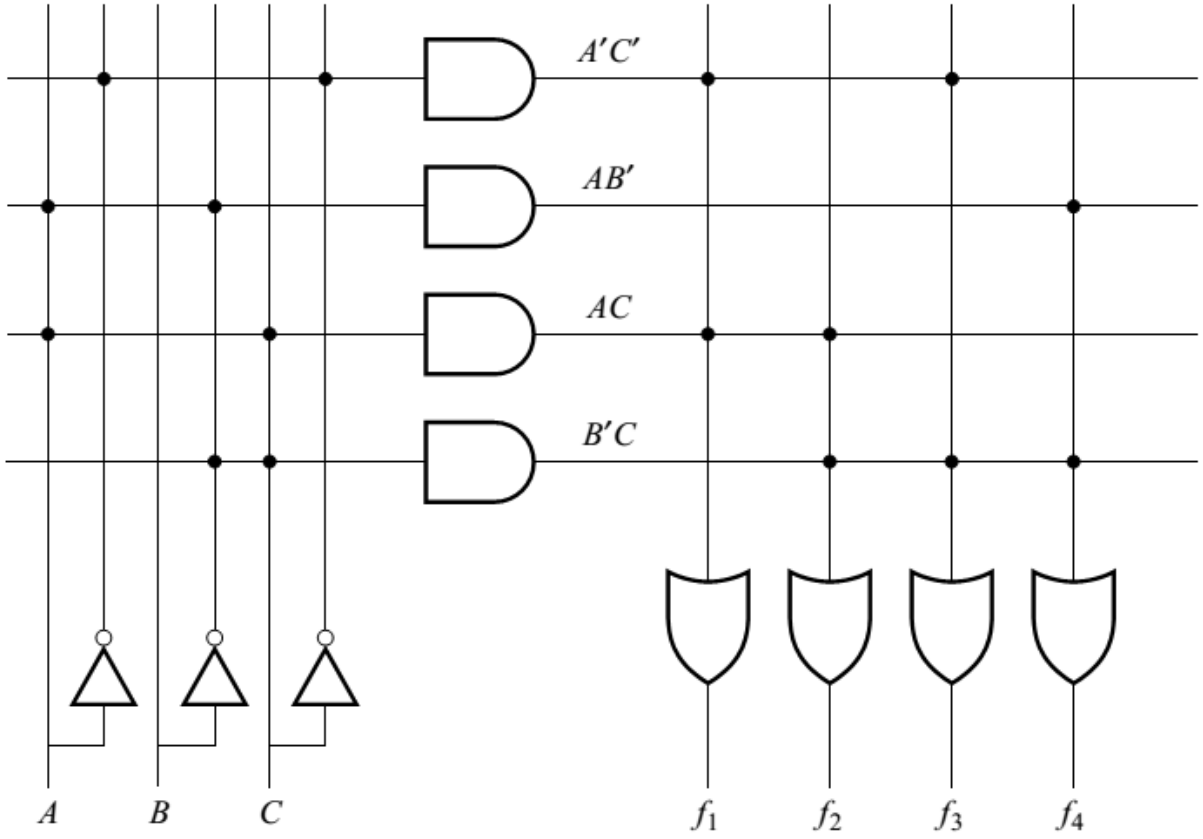
Integrated-circuit RAM units are available in two possible operating modes, static and dynamic. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tends to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. Dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip, but static RAM is easier to use and has shorter read and write cycles.

# Programmed Logic Array (PLA) :-

The canonic sum-of-products implementation of a logic function is wasteful in two ways: in the number of AND gates used (as many as there are min terms, $2^n$) and in the number of inputs to each AND gate n .Suppose we contemplate a reduced (possibly minimal) sum-of-products implementation. Given a logic function of n variables, the largest number of terms in a minimal sum-of-products expression representing this function is $2^{n-1}$-just half the number of min terms.

That means a savings of 50 percent in AND gates for the worst single-output case. Since there will be a reduced set of inputs to the AND gates, this saving in gates is paid for by the need to program not only the outputs of the AND gates but their inputs as well. The structure of the circuit that results is called a programmable (or programmed) logic array (PLA).

The diagram in given Figure is not a circuit diagram but a schematic diagram. A single line is shown to represent all inputs to each AND and OR gate. The number of input lines to each AND gate should be 2n, twice the number of inputs, to accommodate the possibility of connecting each variable or its complement to each AND gate. The number of input lines to each OR gate should equal the number of AND gates, say p. (For simplicity and without fear of confusion, even the gate symbols can be omitted.) The programmed connections between the inputs and the AND gates, and between the AND-gate outputs and the OR gates for a specific set of output functions are shown by the heavy dots at the intersections.




Structure of a PLA.

Maps of the four output functions and minimal sum-of-products expressions are shown in below Figure. In this example, a total of only four product terms covers all functions, so only four AND gates are needed in the implementation. Two sets of lines must be programmed: the input lines and the output lines. To do this, we construct a programming table as follows:

• The implicants (product terms) are listed as row headings.

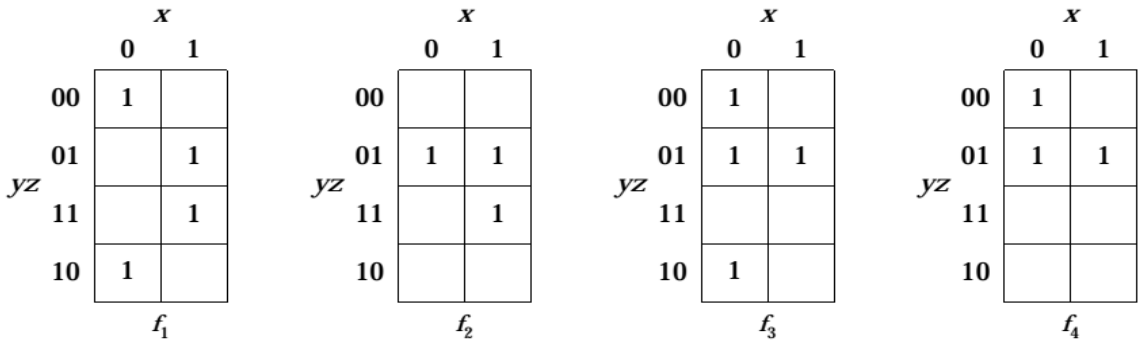
• In one set of columns, the headings are the input variables; this part of the table must provide the information that tells which variables (or their complements) are factors in each implicant.

• In a second set of columns, the headings are the output functions; this part of the table must provide the information that indicates the output gate to which each implicant (AND-gate output) is directed.

In the first set of columns, if a variable ( uncomplemented) is present in a particular row, the corresponding entry is 1; if its complement is present, the entry is 0. If neither is present, the entry can be left blank, but it is preferable to show some symbol instead; a dash is often used.

In the second set of columns, corresponding to the output functions, if a particular function covers a particular implicant, then the corresponding entry is 1; otherwise it could be left blank, but it is customary to enter a dot. To illustrate, consider row 4. Since the implicant is y'z, the entry in column z is 1, that in column y is 0, and that in x is a dash. In the output columns, only $f_1$ does not cover implicant y'z; hence, the entry will be 1 in every column in row 4 except the $f_1$ column. Confirm the remaining rows.

Once the programming is done, fabricating the links (connection points) in a PLA is carried out in a similar manner as for the ROM. The PLA is either mask programmable or field programmable (FPLA). In the case of the FPLA, with p= the number of AND gates, there will be 2np links at the inputs and mp links at the outputs.




| Product Term | Inputs | | | Outputs | | | | |
|---|---|---|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | |
| 1: $x'z'$ | 0 | – | 0 | 1 | • | 1 | • | $f_1 = x'z' + xz$ |
| 2: $xy'$ | 1 | 0 | – | • | • | • | 1 | $f_2 = xz + y'z$ |
| 3: $xz$ | 1 | – | 1 | 1 | 1 | • | • | $f_3 = x'z' + y'z$ |
| 4: $y'z$ | – | 0 | 1 | • | 1 | 1 | 1 | $f_4 = xy' + y'z$ |

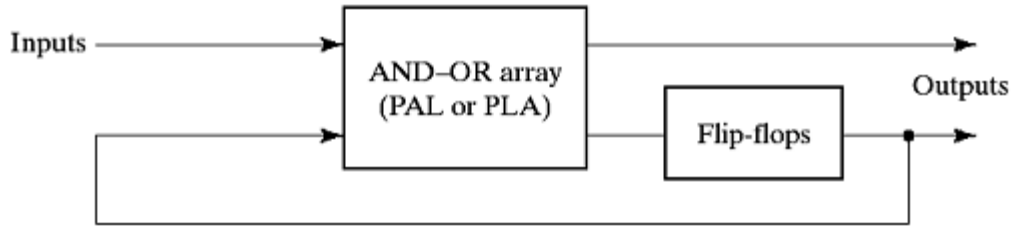
## Programming the PLA

In the example, the number of links is 4(6 + 4) = 40. Only 16 of these are to be kept, meaning that, during field programming, 24 links are to be blown out. Typical PLAs have many more inputs, outputs, and AND gates than are shown in the example.

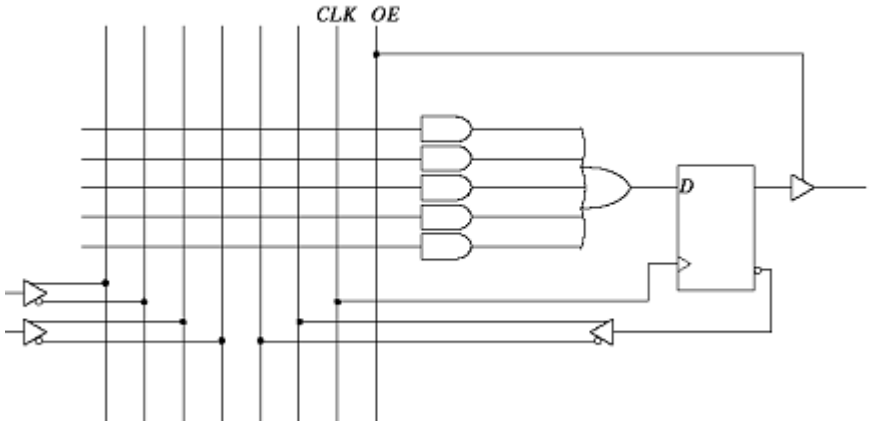
When a set of switching functions is presented for implementation with a PLA, a design goal would be reduction in p(the number of AND gates). The economy achieved is not derived from a reduction in the production cost of gates. (The production cost of an IC is practically the same for one with 40 gates as it is for one with 50 gates.) Rather, the removal of one AND gate eliminates 2n+m links; the main source of savings is the elimination of a substantial number of links due to the elimination of each AND gate. On the other hand, reduction of the number of AND gates to a minimum does not mean that each function should be minimized or that all implicants should be prime implicants. The implicants should be chosen so that as many as possible of them are common to many of the output functions.

# Sequential Programmable Devices:-

The most simple sequential PLD = PLA (PAL) + Flip-Flops




The mostly used configuration for SPLD is constructed with 8 to 10 macro cells as shown below.

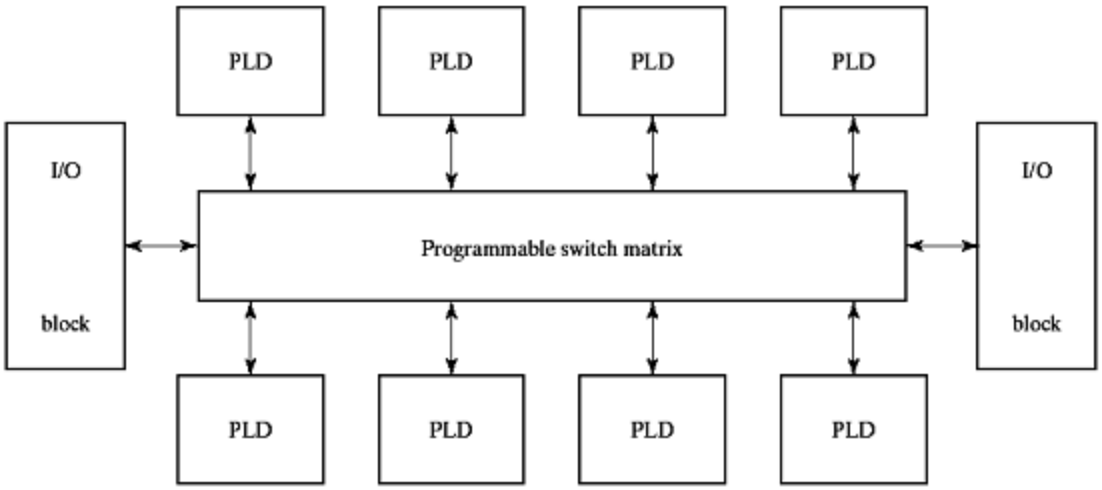


# Complex PLD:-

Complex digital systems often require the connection of several devices to produce the complex specification

More economical to use a complex PLD (CPLD)

CPLD is a collection of individual PLDs on a single IC with programmable interconnection structure




# Field Programmable Gate Array (FPGA):-

-Gate array: a VLSI circuit with some pre-fabricated gates repeated thousands of times

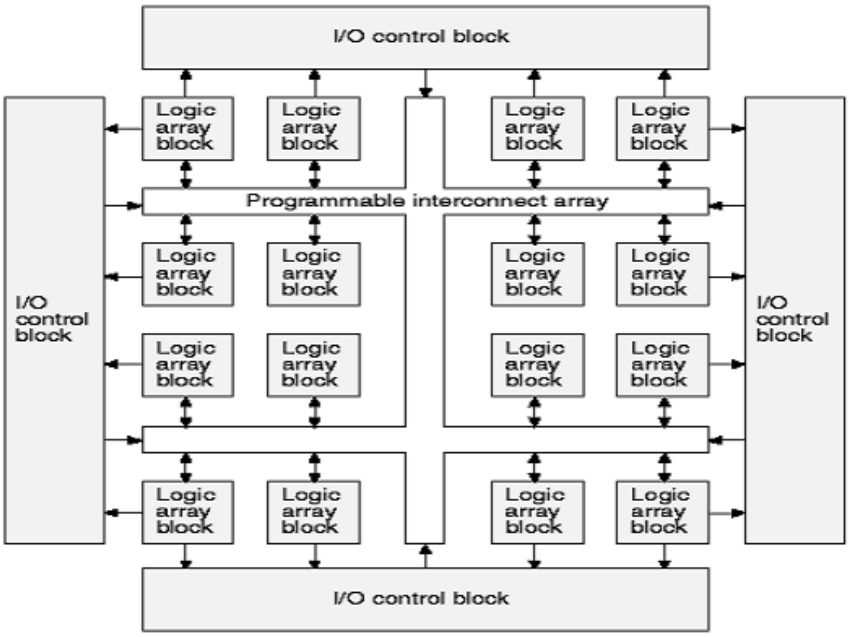
-Designers have to provide the desired interconnection patterns to the manufacturer (factory)

-A field programmable gate array (FPGA) is a VLSI circuit that can be programmed in the user's location

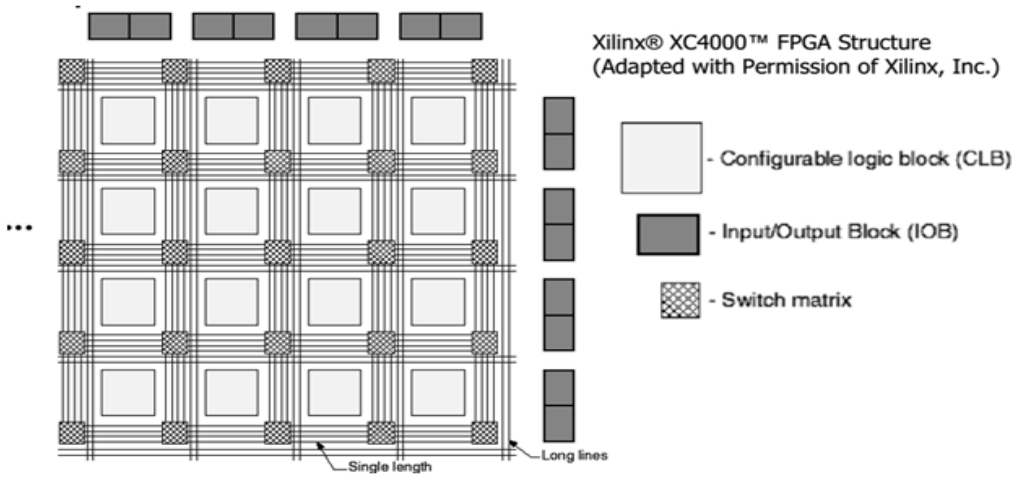
-Easier to use and modify

-Getting popular for fast and reusable prototyping

-There are various implementations for FPGA




FPGA structure (Altera)



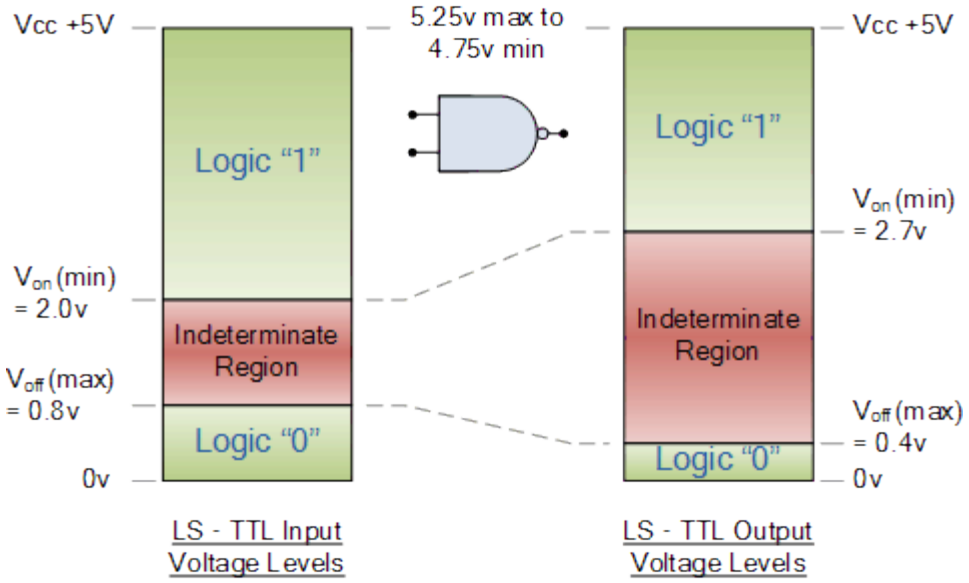

FPGA (Xilinx)

# Digital Logic Circuits :-

## TTL Circuit:-

**TTL** logic IC's use NPN and PNP type Bipolar Junction Transistors while **CMOS** logic IC's use complementary MOSFET or JFET type Field Effect Transistors for both their input and output circuitry. As well as TTL and CMOS technology, simple Digital Logic Gates can also be made by connecting together diodes, transistors and resistors to produce **RTL,** Resistor-Transistor logic gates, **DTL,** Diode-Transistor logic gates or **ECL,** Emitter-Coupled logic gates but these are less common now compared to the popular **CMOS** family.

### TTL Input & Output Voltage Levels:-




When using a standard +5 volt supply any TTL voltage input between 2.0 V and 5 V is considered to be a logic "1" or "HIGH" while any voltage input below 0.8 V is recognized as a logic "0" or "LOW". The voltage region in between these two voltage levels either as an input or as an output is called the *Indeterminate Region* and operating within this region may cause the logic gate to produce a false output.

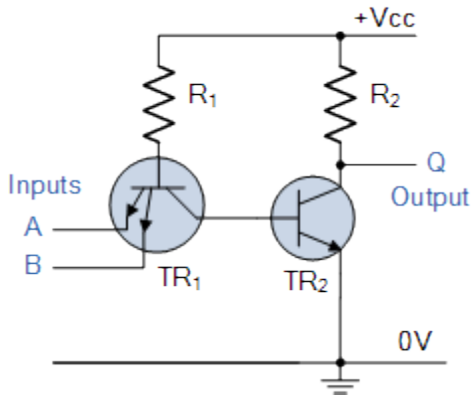
The CMOS 4000 logic family uses different levels of voltages compared to the TTL types as they are designed using field effect transistors, or FET's. In CMOS technology a logic "1" level operates between 3.0 volts and 18 volts and a logic "0" level is below 1.5 volts.

## Basic TTL Logic Gates:-

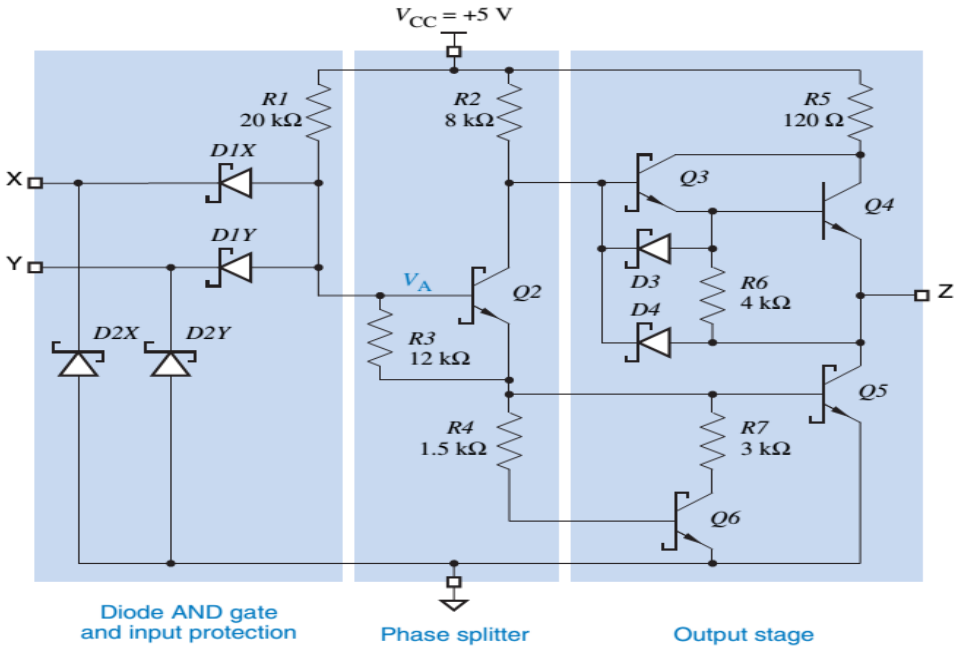
The simple Diode-Resistor AND gate above uses separate diodes for its inputs, one for each input. As a transistor is made up off two diode circuits connected together representing an NPN or a PNP device, the input diodes of the DTL circuit can be replaced by one single NPN transistor with multiple emitter inputs as shown.

As the NAND gate contains a single stage inverting NPN transistor circuit ($TR_2$) an output logic level "1" at Q is only present when both the emitters of $TR_1$ are connected to logic level "0" or ground allowing base current to pass through the PN junctions of the emitter and not the collector. The multiple emitters of $TR_1$ are connected as inputs thus producing a NAND gate function.

In standard TTL logic gates, the transistors operate either completely in the "cut off" region, or else completely in the saturated region, Transistor as a Switch type operation.




2 input NAND Gate




Circuit diagram of 2-input LS-TTL NAND gate.

(a)

| X | Y | $V_A$ | Q2 | Q3 | Q4 | Q5 | Q6 | $V_Z$ | Z |
|---|---|---|---|---|---|---|---|---|---|
| L | L | 1.05 | off | on | on | off | off | 2.7 | H |
| L | H | ≤1.05 | off | on | on | off | off | 2.7 | H |
| H | L | ≤1.05 | off | on | on | off | off | 2.7 | H |
| H | H | 1.2 | on | off | off | on | on | ≤0.35 | L |

(b)

| X | Y | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(c)


(a)Function Table (b) Truth Table (c) Logic symbol

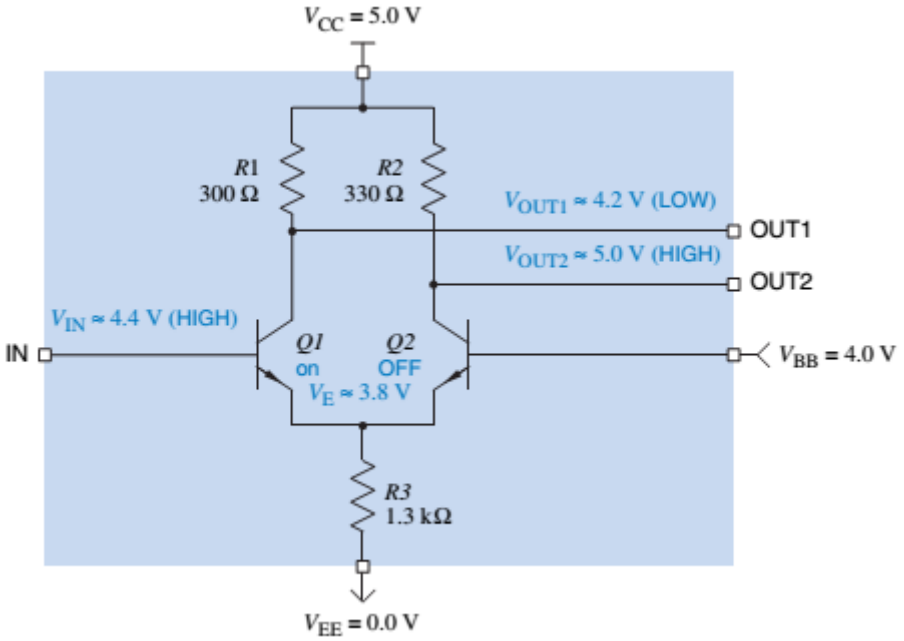
# ECL: Emitter-Coupled Logic :-

The key to reducing propagation delay in a bipolar logic family is to prevent a gate's transistors from saturation. However, it is also possible to prevent saturation by using a radically different circuit structure, called current-mode logic (CML) or emitter-coupled logic (ECL).

Unlike the other logic families in this chapter, ECL does not produce a large voltage swing between the LOW and HIGH levels. Instead, it has a small voltage swing, less than a volt, and it internally switches current between two possible paths, depending on the output state.

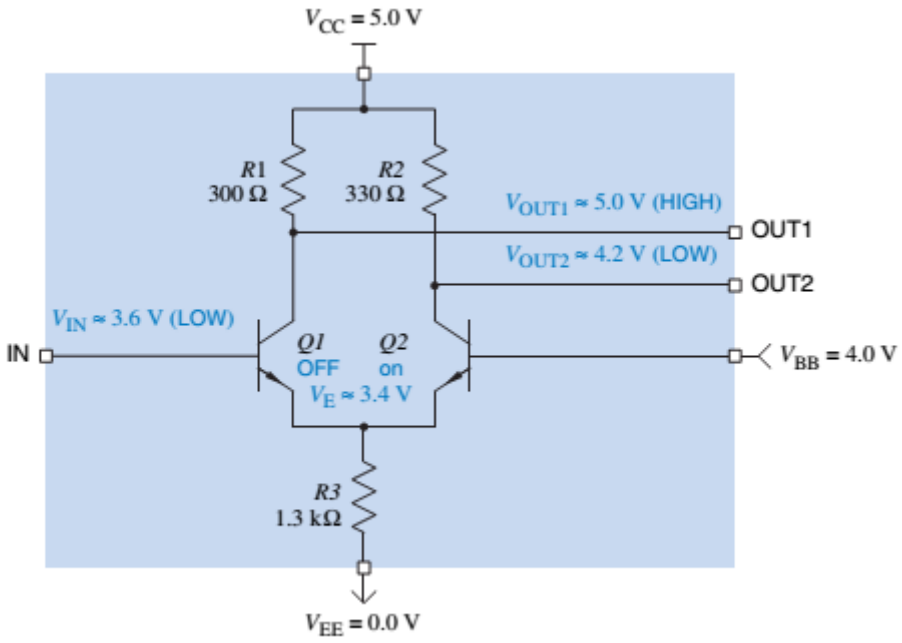
## Basic ECL Circuit

The basic idea of current-mode logic is illustrated by the inverter/buffer circuit in given Figure. This circuit has both an inverting output (OUT1) and a non inverting output (OUT2). Two transistors are connected as a differential amplifier with a common emitter resistor. The supply voltages for this example are $V_{CC}$ =5.0, $V_{BB}$ =4.0, and $V_{EE}$=0 V, and the input LOW and HIGH levels are defined to be 3.6 and 4.4 V. This circuit actually produces output LOW and HIGH levels that are 0.6 V higher (4.2 and 5.0 V), but this is corrected in real ECL circuits.

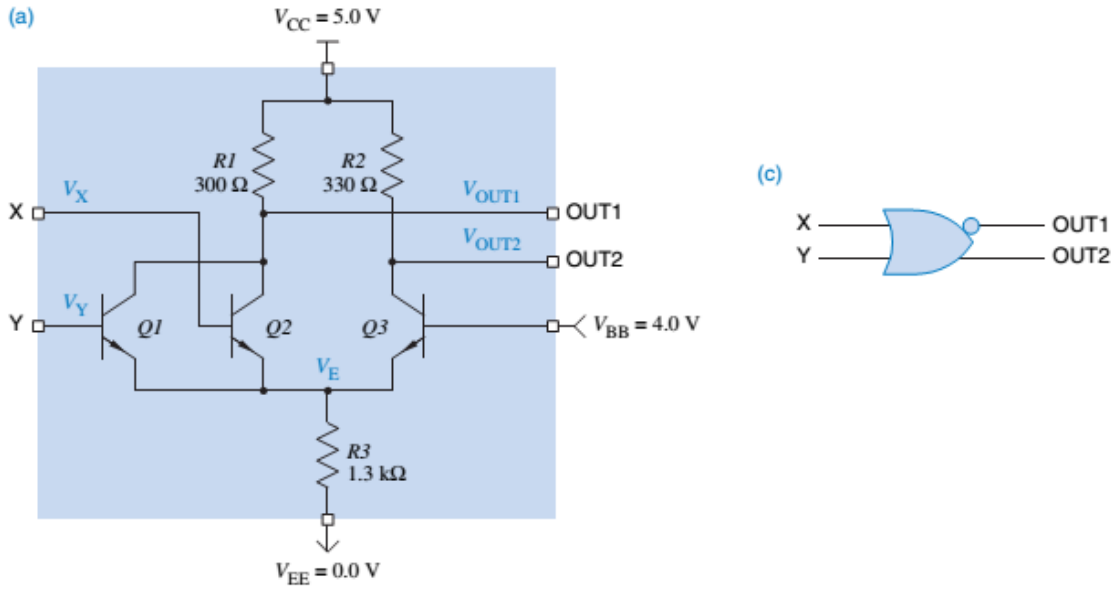



Basic ECL inverter/buffer circuit with input HIGH.

When $V_{IN}$ is HIGH, as shown in the figure, transistor $Q_1$ is on, but not saturated, and transistor $Q_2$ is OFF. This is true because of a careful choice of resistor values and voltage levels. Thus, $V_{OUT2}$ is pulled to 5.0 V (HIGH) through $R_2$, and it can be shown that the voltage drop across $R_1$ is about 0.8 V, so that $V_{OUT1}$ is about 4.2 V (LOW).




Basic ECL inverter/ buffer circuit with input LOW.




(b)

| X | Y | $V_X$ | $V_Y$ | Q1 | Q2 | Q3 | $V_E$ | $V_{OUT1}$ | $V_{OUT2}$ | OUT1 | OUT2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| L | L | 3.6 | 3.6 | OFF | OFF | on | 3.4 | 5.0 | 4.2 | H | L |
| L | H | 3.6 | 4.4 | OFF | on | OFF | 3.8 | 4.2 | 5.0 | L | H |
| H | L | 4.4 | 3.6 | on | OFF | OFF | 3.8 | 4.2 | 5.0 | L | H |
| H | H | 4.4 | 4.4 | on | on | OFF | 3.8 | 4.2 | 5.0 | L | H |

(d)

| X | Y | OUT1 | OUT2 |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |

ECL 2-input OR/NORgate: (a) circuit diagram; (b) function table; (c) logic symbol; (d) truth table

# DIODE TRANSISTOR LOGIC CIRCUITS :-

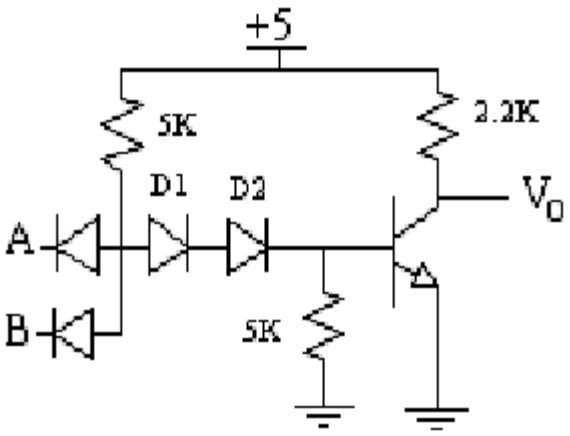
A typical DTL NAND gate is shown in the given Figure. Observe the diode AND function on the front end and the transistor NOT at the output end. The extra resistors and diodes are used to maintain appropriate currents, to maintain proper functioning, and to guarantee certain noise margins.

## Analysis of the DTL gate :-

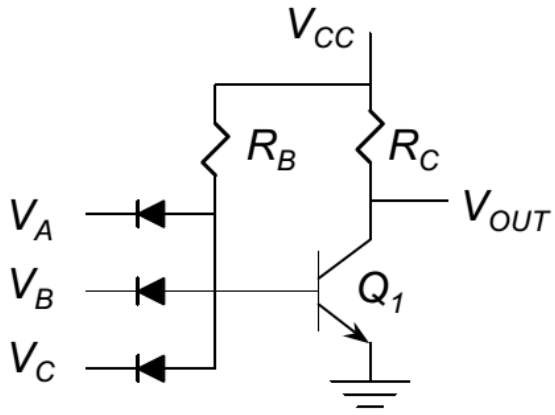
Analysis of the DTL gate is dependent on complete understanding of the currents within the gate under all logic conditions. First let us develop a generic understanding of the operation. Of particular importance will be the direction of currents at the terminals of the gate. As in most logic systems, the transistor will either be cutoff or saturated.

If all inputs are high, (+5 V), no current will come out of the input diodes at the input and current will flow down through the first 5K resistor and through the diodes $D_1$ and $D_2$ toward the base of the transistor. Some current will split off and go down through the lower 5K resistor to ground. However, most of the current will go into the base of the transistor causing it to saturate, pulling the output low, $V_O$=0.2 Volts. We will show this condition quantitatively shortly.

If one or more of the inputs to the gate are held low (0.2 V), then the current down through the 5K resistor will go out the input diode, away from the transistor base. Under this condition, the transistor will be cutoff and the ouput will be high with $V_O$=5 Volts.




DTL circuit




Improved gate with reversed diodes.

-If all inputs are high, the transistor saturates and $V_{OUT}$ goes low.

-If any input goes low, the base current is diverted out through the input diode. The transistor cuts off and $V_{OUT}$ goes high.

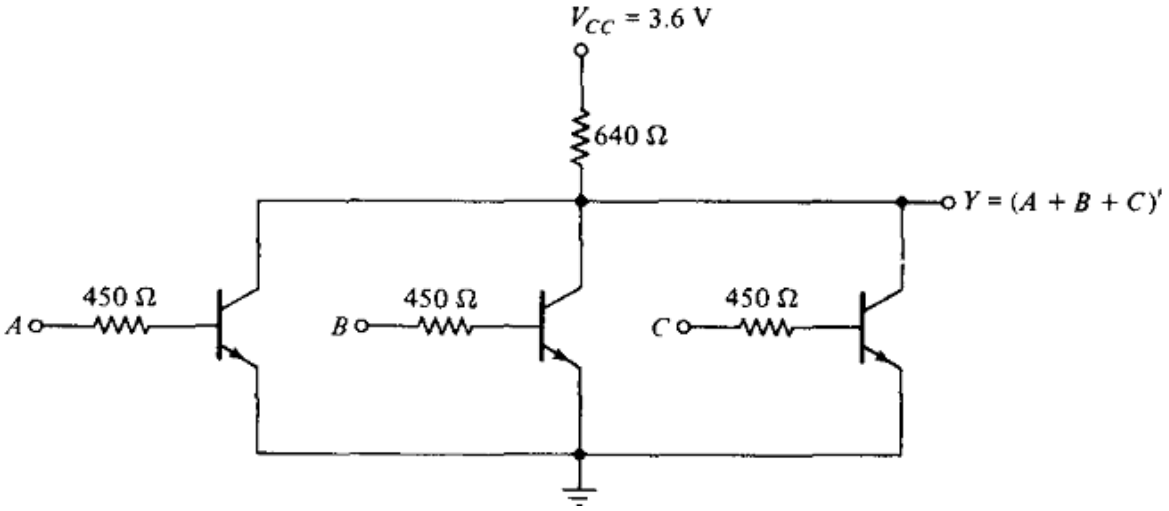
-This is a NAND gate.

# RTL Logic:-

The basic circuit of the RTL digital logic family is the NOR gate shown in given Figure. Each input is associated with one resistor and one transistor. The collectors of the transistors are tied together at the output. The voltage levels for the circuit are 0.2 V for the low level and from 1 to 3.6 V for the high level.

The analysis of the RTL gate is very simple and as follows. If any input of the RTL gate is high, the corresponding transistor is driven into saturation. This causes the output to be low, regardless of the states of the other transistors. If all inputs are low at 0.2 V, all transistors are cut off because $V_{BE} < 0.6$ V. This causes the output of the circuit to be high, approaching the value of supply voltage $V_{cc}$. This confirms the conditions stated in given Fig for the NOR gate. Note that the noise margin for low signal input is 0.6 - 0.2 = 0.4 V.

The fan-out of the RTL gate is limited by the value of the output voltage when high. As the output is loaded with inputs of other gates, more current is consumed by the load. This current must flow through the 640 Ω resistor.

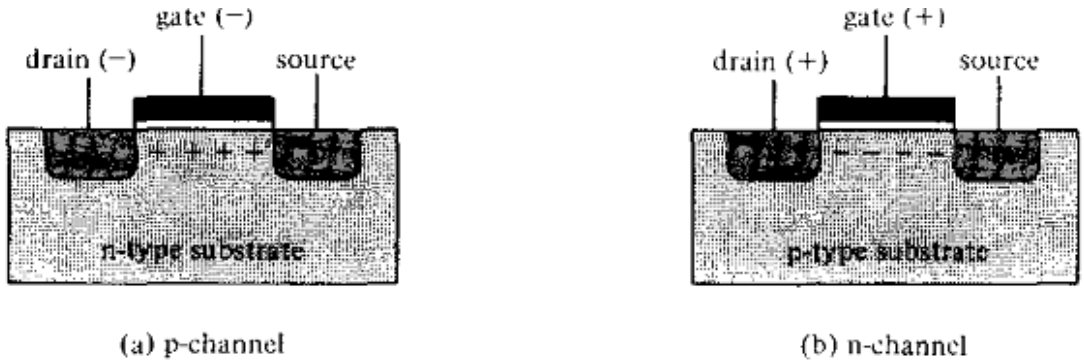



RTL NOR Gate

## METAL OXIDE SEMICONDUCTOR (MOS) :-

The field-effect transistor (FET) is a unipolar transistor, since its operation depends on the flow of only one type of carrier. There are two types of field-effect transistors: the junction field-effect transistor (JFET) and the metal-oxide semi-conductor (MOS). The former is used in linear circuits and the latter in digital circuits. MOS transistors can be fabricated in less area than bipolar transistors.

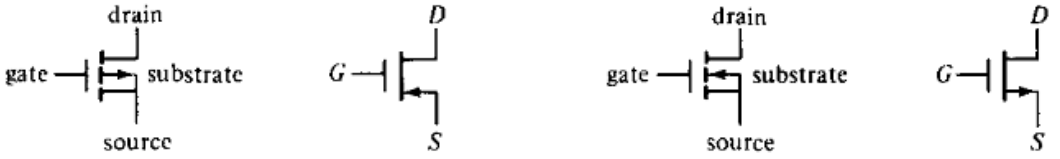
The basic structure of the MOS transistor is shown in given Fig. The p-channel MOS consists of a lightly doped substrate of n-type silicon material. Two regions are heavily doped by diffusion with p-type impurities to form the source and drain. The region between the two p type sections serves as the channel. The gate is a metal plate separated from the channel by an insulated dielectric of silicon dioxide. A negative voltage (with respect to the substrate) at the gate terminal causes an induced electric field in the channel that attracts p-type carriers from the substrate. As the magnitude of the negative voltage on the gate increases, the region below the gate accumulates more positive carriers, the conductivity increases, and current can flow from source to drain provided a voltage difference is maintained between these two terminals.




(a) p-channel (b) n-channel

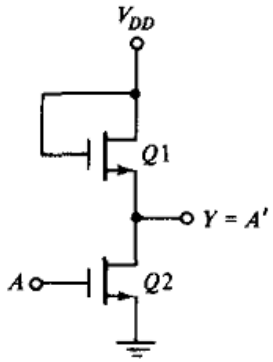
Basic structure of MOS transistor

There are four basic types of MOS structures. The channel can be a p-or n-type, depending on whether the majority carriers are holes or electrons. The mode of operation can be enhancement or depletion, depending on the state of the channel region at zero gate voltage. If the channel is initially doped lightly with p-type impurity (diffused channel), a conducting channel exists at zero gate voltage and the device is said to operate in the depletion mode. In this mode, current flows unless the channel is depleted by an applied gate field. If the region beneath the gate is left initially uncharged, a channel must be induced by the gate field before current can flow. Thus, the channel current is enhanced by the gate voltage and such a device is said to operate in the enhancement mode.

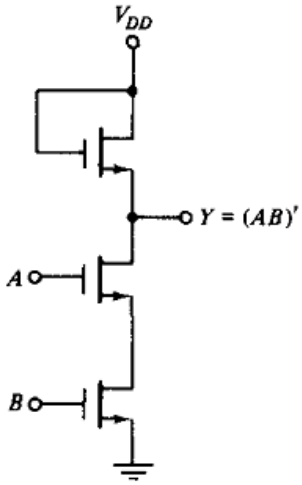



(a) p-channel (b) n-channel

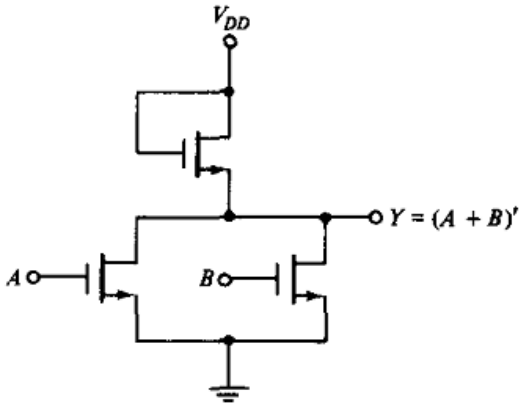
Symbol of MOS Transistor




(a) Inverter




(b) NAND gate

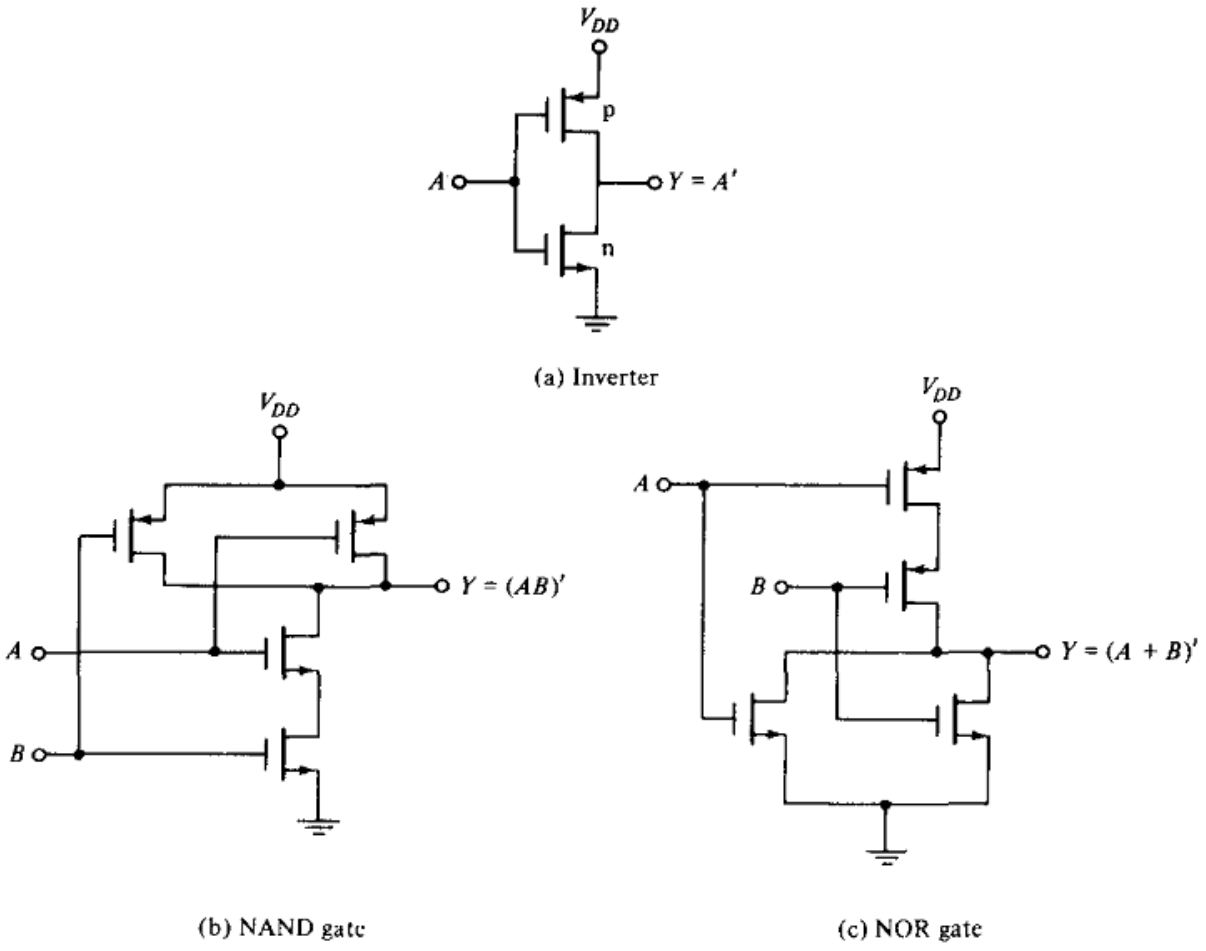



(c) NOR gate

n-channel MOS logic circuits

## COMPLEMENTARY MOS (CMOS) :-

Complementary MOS circuits take advantage of the fact that both n-channel and p-channel devices can be fabricated on the same substrate. CMOS circuits consist of both types of MOS devices interconnected to form logic functions. The basic circuit is the inverter, which consists of one p-channel transistor and one n -channel transistor.



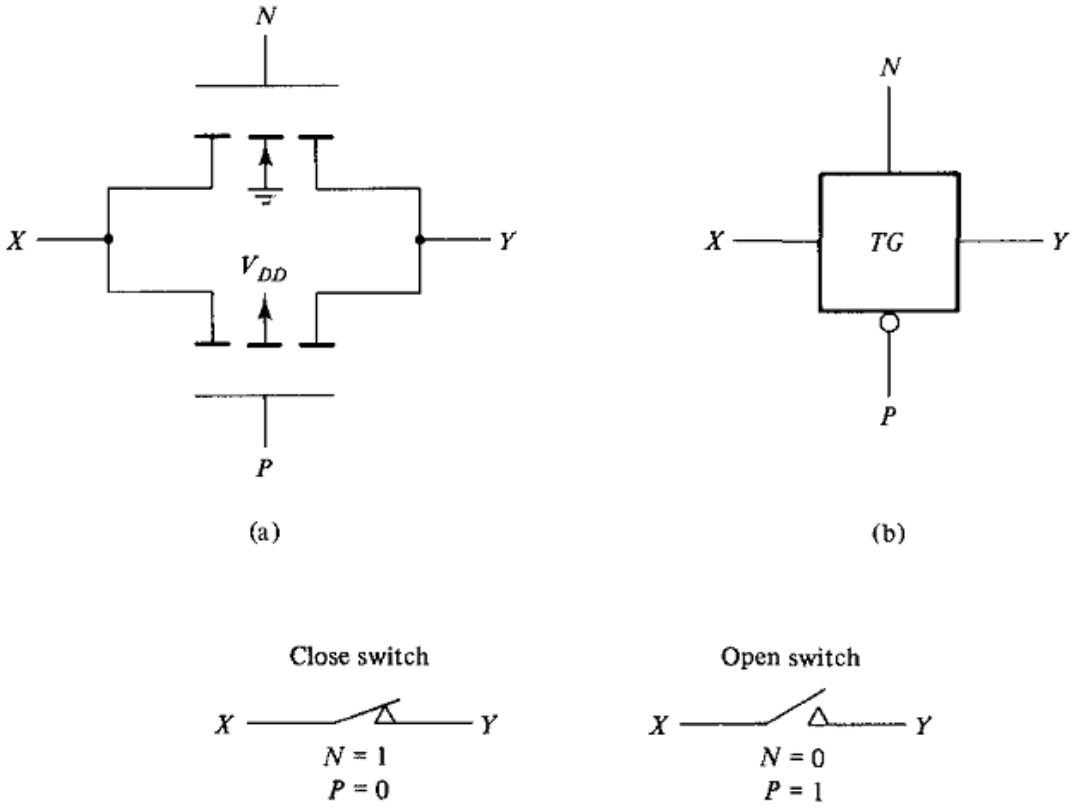

CMOS logic circuits

Now consider the operation of the inverter. When the input is low, both gates arc at zero potential. The input is at $-V_{DD}$ relative to the source of the p-channel device and at 0 V relative to the source of the n-channel device. The result is that the p-channel device is turned on and the n-channel device is turned off. Under these conditions, there is a low-impedance path from $V_{DD}$ to the output and a very high-impedance path from output to ground. Therefore, the output voltage approaches the high level $V_{DD}$ under normal loading conditions. When the input is high, both gates are at $V_{DD}$ and the situation is reversed: The p-channel device is off and the n-channel device is on. The result is that the output approaches the low level of 0 V.

## CMOS TRANSMISSION GATE CIRCUITS:-

The transmission gate is essentially an electronic switch that is controlled by an input logic level. It is used for simplifying the construction of various digital components when fabricated with CMOS technology.

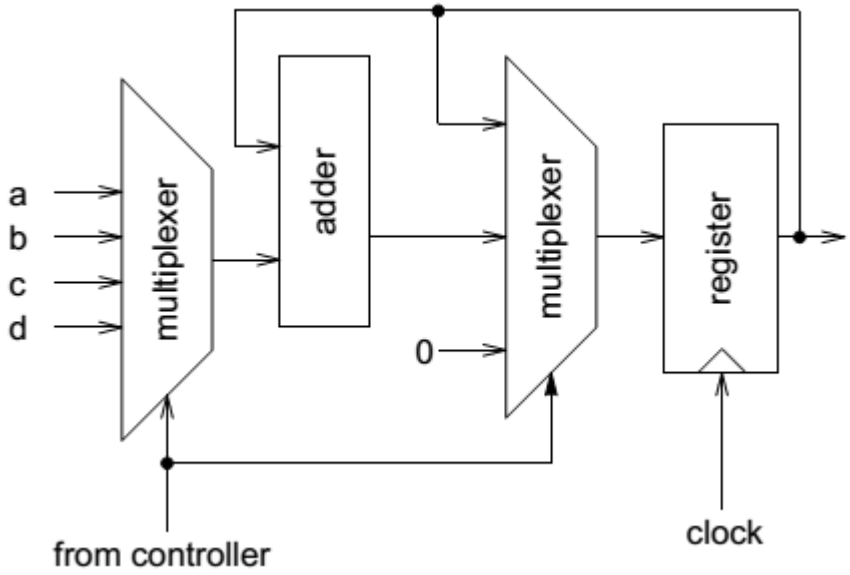
It consists of one n-channel and one p-channel MOS transistor connected in parallel. The n-channel substrate is connected to ground and the p-channel substrate is connected to $V_{DD}$. When the N gate is at $V_{DD}$ and the P gate is at ground, both transistors conduct and there is a closed path between input X and output Y. When the N gate is at ground and the P gate at $V_{DD}$ both transistors are off and there is an open circuit between X and Y.




Transmission Gate (TG)

# RTL Design Example:-

To show how an RTL design is described in VHDL and to clarify the concepts involved, we will design a four-input adder. This design will also demonstrate how to create packages of components that can be re-used. The data path shown below can load the register at the start of each clock cycle with one of: zero, the current value of the register, or the sum of the register and one of the four inputs. It includes one 8-bit register, an 8-bit adder and a multiplexer that selects one of the four possible inputs as the value to be added to the current value of the register.




The first design unit is a package that defines a new type, num, for eight-bit unsigned numbers and an enumerated type, states, with six possible values. nums are defined as a subtype of the unsigned type.

```
-- RTL design of 4-input summer

-- subtype used in design

library ieee ;

use ieee.std_logic_1164.all;

use ieee.std_logic_arith.all;
```

```
package averager_types is
sub type num is unsigned (7 downto 0);
type states is (clr, add_a, add_b, add_c,
add_d, hold);
end averager_types;
```

The first entity defines the data path. In this case the four numbers to be added are available as inputs to the entity and there is one output for the current sum. The inputs to the data path from the controller are a 2-bit selector for the multiplexer and two control signals to load or clear (set to 0) the register.

```
-- data path
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.al;
use work.averager_types.all;
entity datapath is
port (
a, b, c, d: in num;
sum : out num ;
sel : in std_logic_vector (1 downto 0) ;
load, clear, clk : in std_logic
) ;
end datapath ;
architecture rtl of datapath is
signal mux_out, sum_reg, next_sum_reg : num ;
```

```
constant sum_zero : num :=
conv_unsigned(0,next_sum_reg'length) ;
begin
-- mux to select input to add
with sel select mux_out <=
a when "00",
b when "01",
c when "10",
d when others ;
-- mux to select register input
next_sum_reg <=
sum_reg + mux_out when load = '1' else
sum_zero when clear = '1' else
sum_reg ;
-- register sum
process(clk)
begin
if clk'event and clk = '1' then
sum_reg <= next_sum_reg ;
end if ;
end process ;
-- entity output is register output
sum <= sum_reg ;
end rtl;
```

The RTL design's controller is a state machine whose outputs control the multiplexers in the datapath. The controller's inputs are signals that control the controller's state transitions. In this case the only input is an update signal that tells our device to recompute the sum (presumably because one or more of the inputs has changed).

This particular state machine sits at the "hold" state until the update signal is true. It then sequences through the other five states and then stops at the hold state again. The other five states are used to clear the register and to add the four inputs to the current value of the register.

```
-- controller

library ieee ;

use ieee.std_logic_1164.all ;

use work.averager_types.all ;

entity controller is

port (

update : in std_logic ;

sel : out std_logic_vector (1 downto 0) ;

load, clear : out std_logic ;

clk : in std_logic

) ;

end controller ;

architecture rtl of controller is

signal s, holdns, ns : states ;

signal tmp : std_logic_vector (3 downto 0) ;

begin

-- select next state

with s select ns <=
```

```
add_a when clr,
add_b when add_a,
add_c when add_b,
add_d when add_c,
hold when add_d,
holdns when others ; -- hold
-- next state if in hold state
holdns <=
clr when update = '1' else
hold ;
-- state register
process(clk)
begin
if clk'event and clk = '1' then
s <= ns ;
end if ;
end process ;
-- controller outputs
with s select sel <=
"00" when add_a,
"01" when add_b,
"10" when add_c,
"11" when others ;
load <= '0' when s = clr or s = hold else '1' ;
```

```
clear <= '1' when s = clr else '0' ;

end rtl ;
```

# State machine :-

In general, a state machine is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change. A computer is basically a state machine and each machine instruction is input that changes one or more states and may cause other actions to take place. Each computer's data register stores a state. The read-only memory from which a boot program is loaded stores a state (the boot program itself is an initial state). The operating system is itself a state and each application that runs begins with some initial state that may change as it begins to handle input. Thus, at any moment in time, a computer system can be seen as a very complex set of states and each program in it as a state machine. In practice, however, state machines are used to develop and describe specific device or program interactions.

To summarize it, a state machine can be described as:

- An initial state or record of something stored someplace
- A set of possible input events
- A set of new states that may result from the input
- A set of possible actions or output events that result from a new state

A finite state machine is one that has a limited or finite number of possible states. (An infinite state machine can be conceived but is not practical.) A finite state machine can be used both as a development tool for approaching and solving problems and as a formal way of describing the solution for later developers and system maintainers. There are a number of ways to show state machines, from simple tables through graphically animated illustrations.

The finite state machine is also a useful approach to many problems in software architecture; only in this case you don't build one you simulate it.

Essentially a finite state machine consists of a number of states – finite naturally! When a symbol, a character from some alphabet say, is input to the machine it changes state in such a way that the next state depends only on the current state and the input symbol.

Notice that this is more sophisticated than you might think because inputting the same symbol doesn't always produce the same behaviour or result because of the change of state.

The new state depends on the old state and the input.

What this means that the entire history of the machine is summarized in its current state. All that matters is the state that it is in and not how it reached this state. Before you write off the finite state machine as so feeble as to be not worth considering as a model of computation it is worth pointing out that as you can have as many states as you care to invent the machine can record arbitrarily long histories. All you need is a state for each of the possible past histories and then the state that you find the machine in is an indication of not only its current state but how it arrived in that state.

Because a finite state machine can represent any history and a reaction, by regarding the change of state as a response to the history, it has been argued that it is a sufficient model of human behaviour i.e. humans are finite state machines.

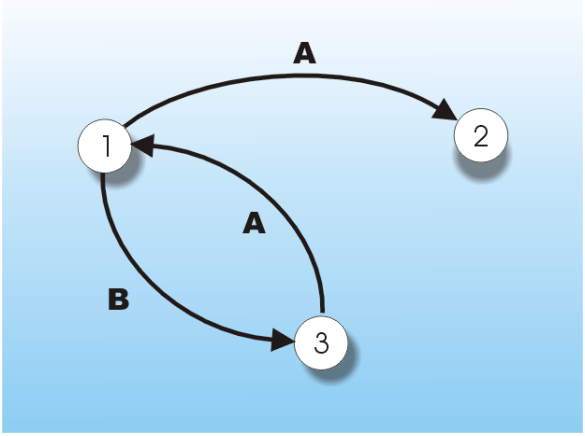
If you know some probability theory you will recognize a connection between finite state machines and Markov chains. A Markov chain sums up the past history in terms of the current state and the probability of transition to the next state only depends on the current state. The Markov chain is a sort of probabilistic version of the finite state machine.

## Representing Finite State Machines

You can represent a finite state machine in a form that makes it easier to understand and think about.

All you have to do is draw a circle for every state and arrows that show which state follows for each input symbol.

For example, the finite state machine in the diagram below has three states. If the machine is in state 1 then an A moves it to state 2 and a B moves it to state 3.




A three-state finite state machine

This really does make the finite state machine look very simple and you can imagine how as symbols are applied to it how it jumps around between states.

What is the point of such a simple machine?

There are two good reasons for being interested in finite state machines. The first is practical. As mentioned earlier, there are some practical applications which are best modelled as a finite state machine.

For example, many communications protocols, such as USB can be defined by a finite state machine's diagram showing what happens as different pieces of information are input. You can even write or obtain a compiler that will take a finite state machine's specification and produce code that behaves correctly.

Many programming problems are most easily solved by actually implementing a finite state machine. You set up an array or other data structure which stores the possible states and you implement a pointer to the location that is the current state. Each state contains a lookup table that shows what the next state is given an input symbol. When a symbol is read in your program simply has to look it up in the lookup table and move the pointer to the new state.