Unit – I

Objectives:

- To familiarize with the concepts of different number systems and codes.

Syllabus:

Number systems - binary numbers, octal, hexadecimal, other binary codes; complements, signed binary numbers, digital logic operations and gates, basic theorems and properties of Boolean algebra, Boolean functions, canonical and standard forms, complements of Boolean functions, two-level NAND and NOR Implementation of Boolean functions.

Outcomes:

Students will be able to

- understand various number systems.
- perform the arithmetic operations using complementary methods.
- understand basic theorems and properties of Boolean algebra.
- understand basic logic operations and gates.
- perform the Two level NAND – NAND and NOR-NOR realizations of Boolean expressions.

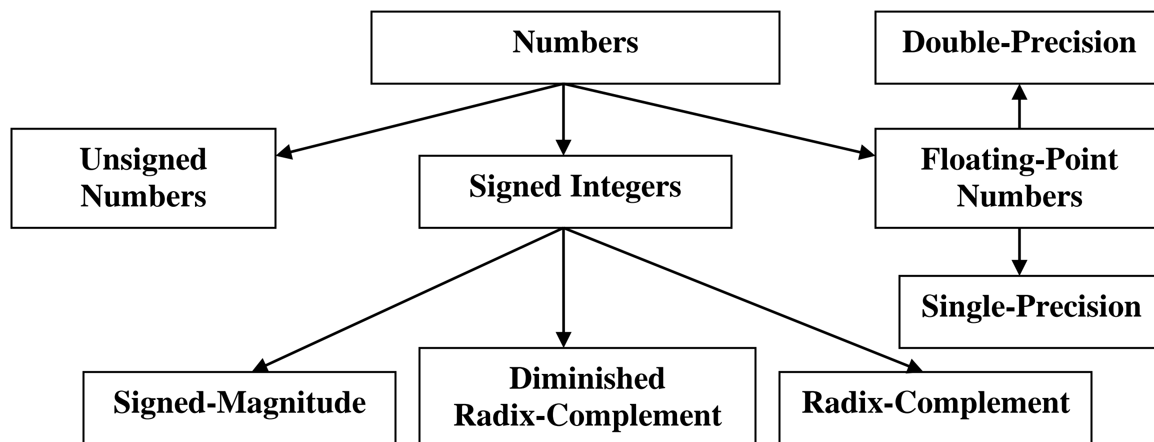
Learning Material

Number Systems

Purposes:

1. ***To understand how does a digital computer work***. Binary digital computers only work with 1's and 0's, or high and low voltage, or true and false.
2. ***To convert among different number systems***. We use **decimal** numbers everyday. Computers understand only **binary** numbers, which are lengthy and inconvenient to human beings. **Octal** and **Hexadecimal** numbers are introduced to make both happy: they are easier to be converted to binary numbers and also easier for us to handle.

Classification:

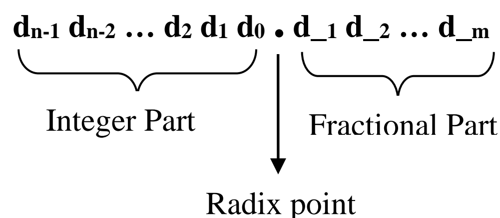


Unsigned Numbers

Radices and Characters:

- **Binary:** 0, 1
- **Decimal:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Octal:** 0, 1, 2, 3, 4, 5, 6, 7
- **Hexadecimal:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Structure of a number:



Note: If no fractional part, the radix point can be omitted!

Positional Notation or representation of numbers:

$$N = d_{n-1}r^{n-1} + d_{n-2}r^{n-2} + \cdots + d_1 r^1 + d_0 r^0 + d_{-1}r^{-1} + d_{-2}r^{-2} + \cdots + d_{-m}r^{-m},$$

where $d_i \in \{0,1,2, r-1\}, i \in \{n-1, n-2, \cdots 2,1,0,-1,-2,\cdots -m\}$, and r is the radix.

The number of numerical values the system uses is called the **Base or Radix** of the system

System	Radix	Allowable Digits
Binary	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A, B, C, D, E, F

Conversion of numbers from one radix to another radix

- Conversion from given base to Decimal:

  write the number using the positional notation and then perform decimal arithmetic to compute the result, which is the decimal number.

Example: Given the positional notations of the following numbers: $(1101.1)_2$, $(724)_8$, and $(BCD)_{16}$.

- $(4021.2)_5$ = $4 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 1 \times 5^0 + 2 \times 5^{-1} = (511.4)_{10}$
  $4 \times 125 + 0 + 10 + 1 + 2 \times (1/5)$
  $500 + 11 + .4$

- $(B65F)_{16}$ = $11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = (46687)_{10}$
  $11 \times 4096 + 6 \times 256 + 5 \times 16 + 15$
  $45056 + 1536 + 80 + 15$

- $(1010.011)_2$ = $2^3 + 2^1 + 2^{-2} + 2^{-3} = (10.375)_{10}$

- $(630.4)_8$ = $6 \times 8^2 + 3 \times 8^1 + 0 \times 8^0 + 4 \times 8^{-1} = (408.5)_{10}$

- Conversion from Decimal to given base:

  Integer part: Divide the decimal number by the base to which we want to convert and cast out the reminders.
  Fractional part: Multiply the decimal number by the base to which we want to convert and cast out the integer part.

Rationale: based on the **positional notation.**

The conversion of decimal numbers with both integers and fraction parts is done by converting the integer and fraction separately and then combining the two answers.

Example: Convert $(210)_{10}$ to binary and to hexadecimal (Radix 16).

- $(210)_{10} = 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

  $= 128 + 64 + 0 + 16 + 0 + 0 + 1 + 0$

  $= (11010010)_2$

- $(210)_{10} = 13 \times 16^1 + 2 \times 16^0$

  $= 208 + 2 = 210 = (D2)_{16}$

- Conversion from Decimal 41 to Binary:

		Integer quotient		Remainder	Coefficient
41/2	=	20	+	½	$a_0 = 1$
20/2	=	10	+	0	$a_1 = 0$
10/2	=	5	+	0	$a_2 = 0$
5/2	=	2	+	½	$a_3 = 1$
2/2	=	1	+	0	$a_4 = 0$
1/2	=	0	+	½	$a_5 = 1$

- The conversion from decimal integers to any base-*r* system is similar to the example, except that division is done by *r* instead of 2.

- Conversion from Decimal 153 to Octal:

```
153 |
 19 |      1
  2 |         3
    |  ↑__
  0   2      = (231) 8
```

- Conversion from Decimal fraction $(0.6875)_{10}$ to Binary:

	Integer		Fraction	Coefficient
0.6875 x 2 =	1	+	0.3750	$a_{-1} = 1$
0.3750 x 2 =	0	+	0.7500	$a_{-2} = 0$
0.7500 x 2 =	1	+	0.5000	$a_{-3} = 1$
0.5000 x 2 =	1	+	0.0000	$a_{-4} = 1$

- The conversion from decimal fraction to any base-*r* system is similar to the example. Multiplication is by *r* instead of **2**, and the coefficients found from the integers may range in value from 0 to *r*-1 instead of **0** and **1**.

- Conversion from Decimal fraction $(0.513)_{10}$ to Octal:

$$
\begin{aligned}
0.513 \times 8 &= 4.104 \\
0.104 \times 8 &= 0.832 \\
0.832 \times 8 &= 6.656 \\
0.656 \times 8 &= 5.248 \\
0.248 \times 8 &= 1.984 \\
0.984 \times 8 &= 7.872
\end{aligned}
$$

$$(0.513)_{10} = (0.406517\ldots)_8$$

Binary to/from Octal and Hexadecimal: Starting at the binary point, cast off three (four) bits at a time and convert each group to its octal (hexadecimal) equivalent. Padding 0's to the left for the integer part and to the right for the fractional part when necessary.

The conversion from and to binary, octal and hexadecimal plays an important part in digital computers. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits.

- Conversion from binary to Octal:

  $(10\ 110\ 001\ 101\ 011.\ 111\ 100\ 000\ 110)_2 = (26153.7406)_8$

- Conversion from binary to Hexadecimal:

  $(10\ 1100\ 0110\ 1011.\ 1111\ 0000\ 0110)_2 = (2C6B.F06)_{16}$

- Conversion from Octal to binary:

  $(673.124)_8 = (110\ 111\ 011.\ 001\ 010\ 100)_2$

- Conversion from Hexadecimal to binary:

  $(306.D)_{16} = (0011\ 0000\ 0110.\ 1101)_2$

- Conversion from Hexadecimal to Decimal:

  $(37B)_{16}$
  $3 \times 16^2 + 7 \times 16^1 + 11 \times 16^0$
  $= 3 \times 256 + 7 \times 16 + 11 \times 1$
  $= 768 + 112 + 11$
  $= (891)_{10}$

r-1's complement and r's complement of unsigned numbers subtraction:

9's & 10's Complements for decimal numbers:

- The Subtraction of decimal numbers can be accomplished by the 9's & 10's compliment methods similar to the 1's & 2's compliment methods of binary numbers.
- The 9's compliment (*diminished radix complement*) of a decimal number is obtained by subtracting each digit of that decimal number from 9.
- The 10's compliment (*radix complement*) of a decimal number is obtained by adding a 1 to its 9's compliment.

Example:
9's compliment of 3465 and 782.54 is

```
 9999                    999.99
-3465                   -782.54
----------              -----------
6534                    217.45
-----------------       -------------------
```

10's complement of 4069 is

```
  9999 -
4069
----------
  5930
    +1
----------
 5931
-----------
```

9's compliment method of subtraction:

To perform this, obtain the 9's compliment of the subtrahend and to it, add the minuend, now call this number as intermediate result. If there is a carry to the LSD of this result to get the answer called **end around carry**. If there is no carry, it indicates that the answer is negative & the intermediate result is its 9's compliment.

Example: Subtract using 9's complement

```
(1) 745.81- 436.62                                  (2) 436.62 - 745.82

  745.81     (normal subtraction)                      436.62
-436.62                                               -745.81
----------                                            ----------
309.19                                                -309.19
-----------                                           ---------
  745.81                                              436.62
+563.37     9's compliment of 436.62                 +254.18
```

```
----------                                        ------------
1309.18   (end around carry)                        690.80 (no carry)
     +1                                            ------------
-----------                                 9‘s complement of 690.80
 +309.19                                            = - 309.19
-------------
```

- If there is no carry indicating that answer is negative. so take 9‘s complement of intermediate result & put minus sign (-) then the result should be -309.19.
- If there is a carry indicates that the answer is positive +309.19. Then there is no need of taking 9‘s complement.

10’s compliment method of subtraction:

- To perform this, obtain the 10‘s compliment of the subtrahend & add it to the minuend. If there is a carry ignore it.
- The presence of the carry indicates that the answer is positive, the result is the answer.
- If there is no carry, it indicates that the answer is negative & the result is its 10‘s compliment.
- Obtain the 10‘s compliment of the result & place negative sign infront to get the answer.

Example:

(a) 2928.54 - 416.73 (b) 416.73 - 2928.54

```
 2928.54   (normal subtraction)                  0416.73
-0416.73                                        -2928.54
----------                                      ----------
2511.81                                         -2511.81
-----------                                     ---------
2928.54                                          0416.73
+9583.27     10‘s compliment of 416.73          +7071.46
----------                                     ------------
12511.81       ignore the carry                  7488.19
                                             (10’s complement)
 +2511.81                                        ---------
                                                 -2511.81
```

1‘s & 2’s compliment form for binary numbers:

- The **1‘s complement** of a binary number is defined as the value obtained by inverting all the bits in the binary representation of the number (swapping 0s for 1s and vice versa).

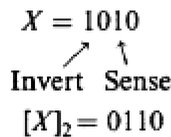
Example:

For X = 1010, the 1's complement is given by 0101.

- The **2's complement** of a binary number X is obtained by following three methods
    1. The expression $2^n - X$, where n is the number of bits of X.
    2. All the bits are inverted (1’s complement) and a 1 is added in the least significant place.
    3. The lowest order 1 in X is sensed, and all succeeding higher digits are inverted.

Example:
For X = 1010, the 2's complement is given by:

1. $2^4 - 1010 = 10000 - 1010 = 0110$.
2. 1’s complement of 1010 is 0101 and 0101 + 1 = 0110.
3. The low order 1 in 1010 is at 1st bit position and after that the higher digits are inverted and the result is 1010.



Signed binary numbers:

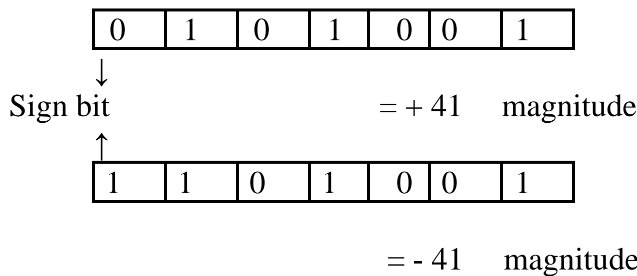
Two ways of representation of signed numbers

1. Sign Magnitude form
2. Complemented form

Sign Magnitude form:

- In sign magnitude form, an additional bit called the sign bit is placed in front of the number.
- If the sign bit is 0, the number is positive, and if it is a 1, then the number is negative.

Example:



Representation of signed numbers using 2’s or 1’s complement method:

- If the number is positive, the magnitude is represented in its true binary form & a sign bit 0 is placed in front of the MSB.
- If the no is negative, the magnitude is represented in its 2‘s or 1‘s compliment form & a sign bit 1 is placed in front of the MSB.

Example:

Sign bit magnitude
↓

0	1	1	0	0	1	1

In any form

= +51

1	1	1	0	0	1	1

In sign magnitude form

= -51

1	0	0	1	1	0	1

In sign 2‘s compliment form

= -51

1	0	0	1	1	0	0

In sign 1‘s compliment form

= -51

Given no.	Sign magnitude form	2‘s complement form	1‘s complement form
01101	+13	+13	+13
010111	+23	+23	+23
10111	-7	-9	-8
1101010	-42	-22	-21

Special case in 2’s complement representation:

Whenever a signed no. has a 1 in the sign bit & all 0‘s for the magnitude bits, the decimal equivalent is -2^n, where n is the no of bits in the magnitude.

Example:

1000 = -8 & 10000 = -16

2’s compliment Arithmetic:

- The 2‘s complement system is used to represent positive numbers using modulus arithmetic.
- The word length of a computer is fixed. i.e., if a 4-bit number is added to another 4-bit number, the result will be only of 4 bits.
- Carry if any, from the fourth bit will overflow called the Modulus arithmetic.
  Example: 1100+1111=1011

- In the 2‘s complement subtraction, add the 2‘s complement of the subtrahend to the minuend.
- If there is a carry out, ignore it and look at the sign bit i.e., MSB of the sum term.
- If the MSB is a 0, the result is positive and it is in true binary form.
- If the MSB is a 1 (carry in or no carry at all) the result is negative and is in its 2‘s complement form. Take its 2‘s complement to find its magnitude in binary.

Example:

Subtract 14 from 46 using 8-bit 2‘s complement arithmetic:

```
    +14    =  00001110
    -14    =  11110010        2‘s complement of 14

    +46    =   00101110
    -14    =  +11110010       2‘s complement form of 14
    ___     ____________
    -32      (1)00100000      ignore carry
```

Ignore carry and the MSB is 0. So, the result is positive and is in normal binary form. So the result is +00100000 = +32.

Example: Add -75 to +26 using 8-bit 2‘s complement arithmetic

```
    +75    =  01001011
    -75    = 10110101         2‘s complement of 75

    +26    =   00011010
    -75    = +10110101
    ___     ____________
    -49        11001111        No carry
```

No carry and MSB is 1. So the result is negative and is in 2‘s complement form.
The magnitude is 2‘s complement of 11001111. i.e., 00110001 = 49. So result is -49

1’s compliment arithmetic:

- In 1‘s complement subtraction, add the 1‘s complement of the subtrahend to the minuend.
- If there is a carryout, bring the carry around & add it to the LSB called the **end around carry.**
- Look at the sign bit (MSB). If this is a 0, the result is positive and a true binary number.
- If the MSB is a 1 (carry or no carry), the result is negative and in complement form. Take its 1‘s complement to get the magnitude in binary.

Example: Using 8-bit 1‘s complement

```
Subtract 14 from 25                      ADD -25 to +14
     25    =   00011001                    +14    =   00001110
    -14    =   11110001                    -25    = +11100110
    ___      ___________                   ___     ____________
    +11      (1)00001010                   -11        11110100
                      +1
              ____________                 No carry and MSB = 1
                00001011               Result is negative  and in 1’s complement form
MSB is a 0 so result is positive (true binary)
```

Compliment Arithmetic Advantage:

Subtraction is also performed by addition. Instead of subtracting one number from other the compliment of the subtrahend is added to minuend.

Codes

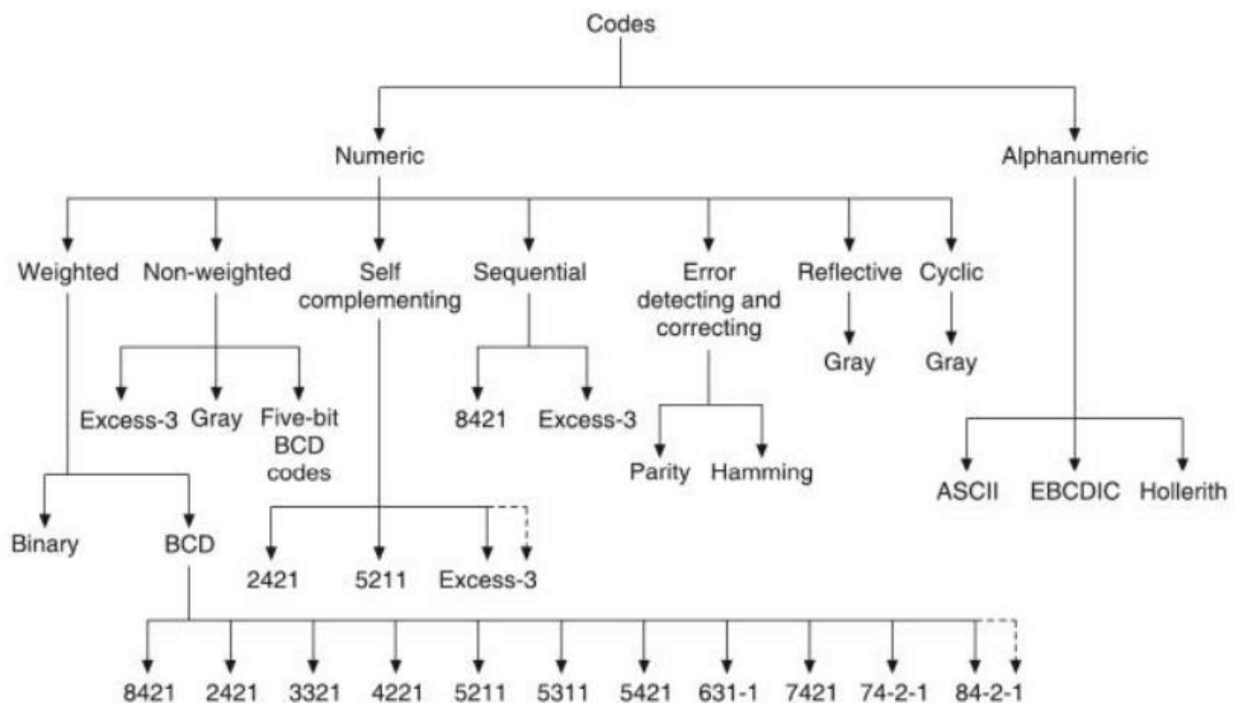
The digital data is represented, stored and transmitted as group of binary bits. This group is also called as **binary code**. The binary code is represented by the number as well as alphanumeric letter.

Advantages of Binary Code

Following is the list of advantages that binary code offers.

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Binary codes make the analysis and designing of digital circuits if we use the binary codes.
- Since only 0 & 1 are being used, implementation becomes easy.

Classification of codes

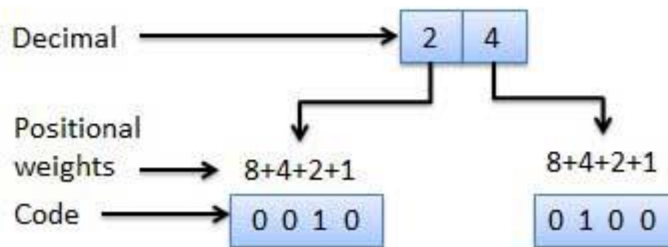


The codes are broadly categorized into following four categories.

- Weighted Codes
- Non-Weighted Codes
- Binary Coded Decimal Code
- Alphanumeric Codes
- Error Detecting Codes
- Error Correcting Codes

Weighted Codes

Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits.

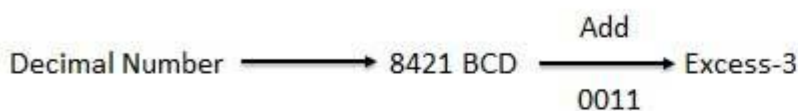


Non-Weighted Codes

In this type of binary codes, the positional weights are not assigned. The examples of non-weighted codes are Excess-3 code and Gray code.

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 in 8421. The excess-3 codes are obtained as follows −



Example:

Decimal	BCD 8 4 2 1	Excess-3 BCD + 0011
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

Gray Code

- 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, only one bit will change each time the decimal number is incremented as shown in fig. As only one bit changes at a time, the gray code is called as a unit distance code. The gray code is a cyclic code. Gray code cannot be used for arithmetic operation.

Decimal	BCD	Gray
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1

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.

Binary Coded Decimal (BCD) code

In this code each decimal digit 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.

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

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.

Alphanumeric codes

A binary digit or bit can represent only two symbols as it has only two states '0' or '1'. But this is not enough for communication between two computers because there we need many more symbols for communication. These symbols are required to represent 26 alphabets with capital and small letters, numbers from 0 to 9, punctuation marks and other symbols.

The alphanumeric codes are the codes that represent numbers and alphabetic characters. Mostly such codes also represent other characters such as symbol and various instructions necessary for conveying information. An alphanumeric code should at least represent 10 digits and 26 letters of alphabet i.e. total 36 items. The following three alphanumeric codes are very commonly used for the data representation.

- American Standard Code for Information Interchange (ASCII).
- Extended Binary Coded Decimal Interchange Code (EBCDIC).
- Five bit BCD Code.

ASCII code: ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers.

Sequential Code: These are those codes in which each succeeding code is 1 binary number greater than the preceding code. This property is used for mathematical manipulation of data. For ex:- BCD And Excess-3 Code.

Self-complementary Code: A code is said to be self-complementary if the code for 9's complement of N i.e. 9-N can be obtained by interchanging all 0s and 1s.

- Decimal 9 is the complement of code for 0, 8 for 1, 7 for 2 and so on.
- For a code to be self complementing, the sum of all its weights must be 9. Digit. 8421 and 5421 codes are not self complementing codes whereas 5211, 2421, 3321, 4321 are self complementing.
- In general, a code is self-complementary if we produce a code by taking the first complement of the digit which is same as 9's complement of the number.

Cyclic codes:

- Cyclic codes are those in which each successive code word differs from the preceding one in only one bit position.
- They are also called unit distance codes
- Example: gray code Reflective Code: Example : Gray code

Binary–Gray Code Conversion A given binary number can be converted into its Gray code equivalent by going through the following steps:

- Begin with the most significant bit (MSB) of the binary number. The MSB of the Gray code equivalent is the same as the MSB of the given binary number.
- The second most significant bit, adjacent to the MSB, in the Gray code number is obtained by adding the MSB and the second MSB of the binary number and ignoring the carry, if any. That is, if the MSB and the bit adjacent to it are both '1', then the corresponding Gray code bit would be a '0'.
- The third most significant bit, adjacent to the second MSB, in the Gray code number is obtained by adding the second MSB and the third MSB in the binary number and ignoring the carry, if any.
- The process continues until we obtain the LSB of the Gray code number by the addition of the LSB and the next higher adjacent bit of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of binary code 1011 into its Gray code equivalent:

Gray code 1- - - Binary 1011

Gray code 11- - Binary 1011

Gray code 111- Binary 1011

Gray code 1110

Basic logic operations NOT, OR, AND:

Binary logic consists of binary variables and logic operations. Each binary variable consists of two states called logic ‘0’ and logic ‘1’. There are 3 basic logical operations: AND,OR,NOTand derived operations are NAND, NOR,X-OR, X-NOR.

AXIOMS:

Axioms or Postulates are a set of logical expressions without proof. Each axiom can be interpreted as the outcome of an operation performed by a logic gate.

AND(A.B=C)	OR(A+B=C)	NOT(A’=B)
0.0=0	0+0=0	1’= 0
0.1=0	0+1=1	0’ = 1
1.0=0	1+0=1	
1.1=1	1+1=1	

LOGIC GATES:

Logic gates are fundamental building blocks of digital systems. Logic gateproduces one output level when some combinations of input levels are present and a different output level when other combination of input levels is present. Based on the axioms there 3 basic types of logic gates were available which are indicated by AND, OR, NOT.

The interconnection of gates to perform a variety of logical operation is called *LogicDesign*. Inputs & outputs of logic gates can occur only in two levels i.e., 1,0 or High, Low or True ,False or On , Off.

A table which lists all the possible combinations of input variables & the corresponding outputs is called a Truth Table. It shows how the logic circuits output responds to various combinations of logic levels at the inputs.

Level Logic, a logic in which the voltage levels represent logic 1 & logic 0.Level logic may be Positive Logic or Negative Logic.

In *PositiveLogic* the higher of two voltage levels represent logic 1 & Lower of two voltage levels representlogic 0.In *Negative Logic* the lower of two voltage levels represent logic 1 & higher of two voltage levels represent logic 0.

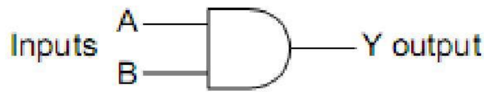
Ex:

In TTL (Transistor-Transistor Logic) Logic family voltage levels are +5V and 0V.Logic 1 represent +5Vand Logic 0 represent 0V.

AND Gate:

It is represented by “.”(dot) It has two or more inputs but only one output. The output assume the logic 1 state only when each one of its inputs is at logic 1 state. The output assumes the logic 0 state even if one of its inputs is at logic 0 state. The AND gate is also called an All or Nothing gate.

Boolean Expression: A AND B, Y=A.B



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

Logic Symbol Truth Table

OR Gate:

It is represented by "+"(plus). It has two or more inputs but only one output. The output assumes the logic 1 state only when one of its inputs is at logic 1 state. The output assumes the logic 0 state even if each one of its inputs is at logic 0 state. The OR gate is also called an any or All gate. Also called an inclusive OR gate because it includes the condition both the inputs can be present.

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

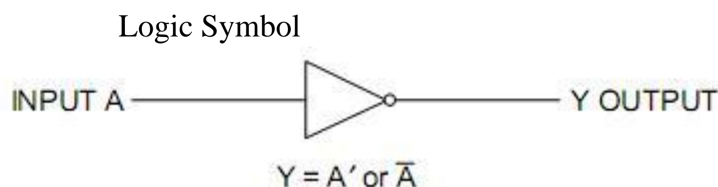
Logic Symbol Truth Table

Boolean Expression:A OR B, A+B=Y

NOT Gate:

It is represented by "-"(bar).It is also called an *Inverter or Buffer*. It has only one input and one output. Whose output always the compliment of its input. Theoutput assumes logic 1 when input is logic 0 & output assume logic 0 when input is logic 1.

Logic Symbol



Truth table:

A	X
1	0
0	1

- Logic circuits of any complexity can be realized using only AND, OR , NOT gates. Using these 3 called AND-OR-INVERT i.e, AOI Logic circuits.

The Universal Gates:

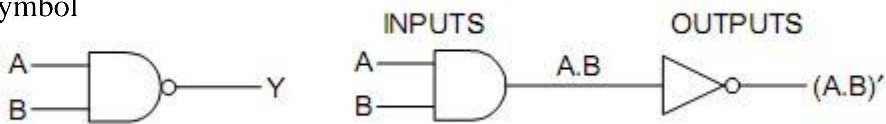
The universal gates are NAND, NOR. These gates are called universal gates because any Boolean logic function including basic operations(AND, OR, INVERT) can be implemented using NAND and NOR gates. More over AOI logic can be easily converted to NAND logic or NOR logic.

NAND Gate: It is combination of AND gate followed by NOT gate

Boolean Expression: $Y = \overline{(A.B)}$

NAND assumes Logic 0 when each of inputs assumes logic 1.

Logic Symbol



Truth table

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

Bubbled OR gate: The output of this is same as NAND gate.

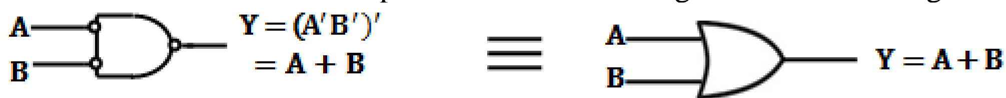
Bubbled OR gate is OR gate with inverted inputs.

$$Y = \bar{A} + \bar{B} = (AB)$$

NAND gate as an Inverter:

All its input terminals together & applying the signal to be inverted to the common terminal by connecting all input terminals except one to logic 1 & applying the signal to be inverted to the remaining terminal. It is also called Controlled Inverter.

Bubbled NAND Gate: The output of bubbled NAND gate is same as OR gate

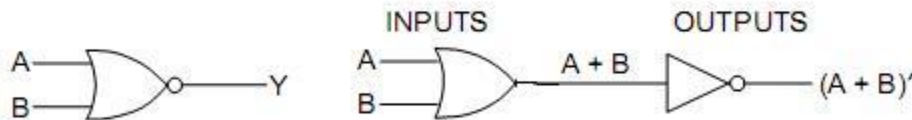


NOR Gate:

NOR gate is NOT gate with OR gate. i.e, OR gate is NOTed.

Boolean expression:$Y = \overline{(A + B)}$

Logic SymbolLogic symbol with OR and NOT



Truth Table:

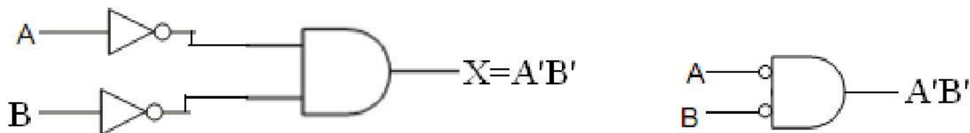
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Bubbled AND gate:

It is AND gate with inverted inputs. The AND gate with inverted inputs is called a bubbled AND gate. So a NOR gate is equivalent to a bubbled and gate. A bubbled AND gate is also called a negative AND gate. Since its output assumes the HIGH state only when all its inputs are in LOW state, a NOR gate is also called active-LOW AND gate.

Output Y is 1 only when both A & B are equal to 0.i.e, only when both A' and B' are equal to 1.NOR can also realized by first inverting the inputs and performing AND operation those inverted inputs.

Logic Symbol

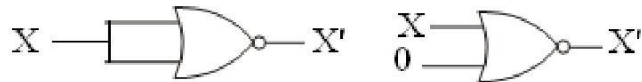


Truth table:

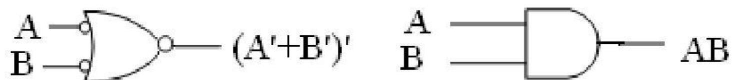
Inputs A B	Inverted Inputs A' B'	Output Y
0 0	1 1	1
0 1	1 0	0
1 0	0 1	0
1 1	0 0	0

NOR gate as an inverter:

is tying all input terminals together & applying the signal to be inverted to the common terminals or all inputs set as logic 0 except one & applying signal to be inverted to the remaining terminal.



Neither bubbled NOR Gate: is AND gate.



The Exclusive OR (X-OR) gate:

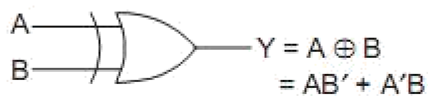
It has 2 inputs& only 1 output. It assumes output as 1 when input is not equal called anti-coincidence gate or inequality detector.

Logic Symbol

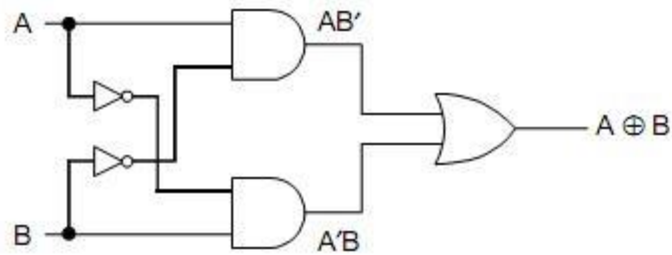
Truth table:

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

The high outputs are generated only when odd number of high inputs is present. This is why x-or function also known as odd *function.*

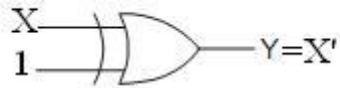


The X-OR gate using AND-OR-NOT gates:



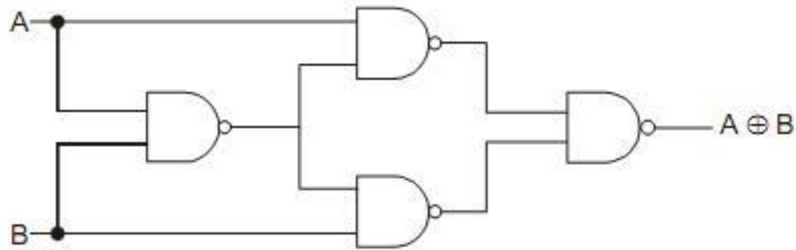
X-OR gate as an Inverter:

By connecting one of two input terminals to logic 1 & feeding the sequence to be inverted to other terminal

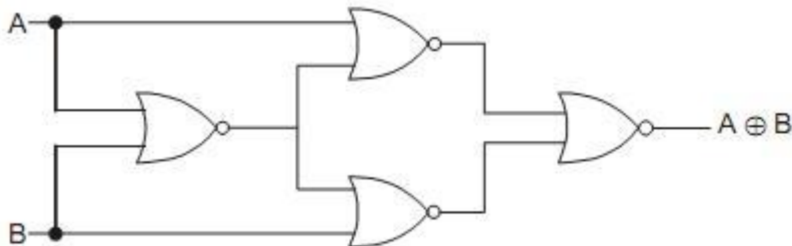


Logic Symbol

X-OR gate using NAND gates only:



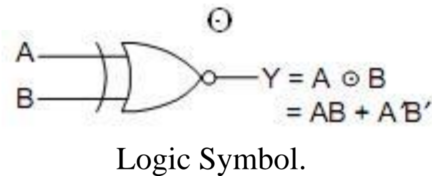
X-OR gate using NOR gates only:



The EX-NOR Gate:

It is X-OR gate with a NOT gate.It has two inputs & one output logic circuit. It assumes output as 0 when one if inputs are 0 and other 1.It can be used as an equality detector because it outputs a 1 only when its inputs are equal.

Proof: $A \odot B = (A \oplus B)'$
$= (AB' + A'B)'$
$= (A' + B).(A + B')$
$= AA' + A'B' + AB + BB'$
$= AB + A'B'$



Logic Symbol.

Truth table:		
Inputs		Output
A	B	X= A B
0	0	1
0	1	0
1	0	0
1	0	1

Boolean theorems:

Boolean algebra:

Switching circuits called Logic circuits, gate circuits & digital circuits. Switching algebra called Boolean Algebra. Boolean algebra is a system of mathematical logic. It is an algebraic system consisting of the set of element (0,1) two binary operators called OR & AND & One unary operator NOT.

A+A=A , A.A=A because variable has only a logic value.

Complementation Laws:

Complement means invert(0' as 1 & 1' as 0)

Law1:0'=1

Law2:1'=0

Law3:If A=0 then A' =1

Law4:If A=1 then A' =0

Law5: (A')' =A(double complementation law)

AND laws:

Law 1: A.0=0(Null law)

Law 2:A.1=A(Identity law)

Law 3:A.A=A

Law 4:A.A' =0

OR laws:

Law 1: A+0=A(Null law)

Law 2:A+1=1

Law 3:A+A=A

Law 4:A+ =0

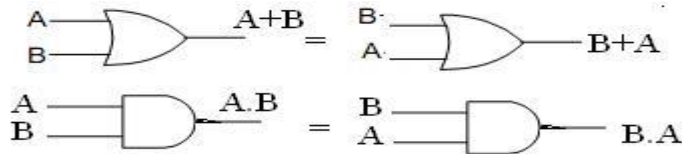
Commutative laws: allow change in position of AND or OR variables.2 commutative laws

Law 1: A+B=B+A

Law 2: A.B=B.A

A B	A+B	=	B A	B+A
0 0	0		0 0	0
0 1	1		0 1	1
1 0	1		1 0	1
1 1	1		1 1	1

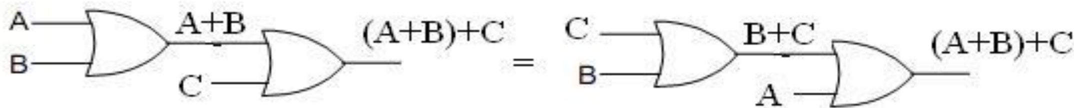
A.B	B.A
0	0
0	0
0	0
1	1



Associative laws: This allows grouping of variables. It has 2 laws.

Law 1: (A+B)+C=A+(B+C) =A OR B ORed with C

This law can be extended to any no. of variables
(A+B+C)+D=(A+B+C)+D=(A+B)+(C+D)



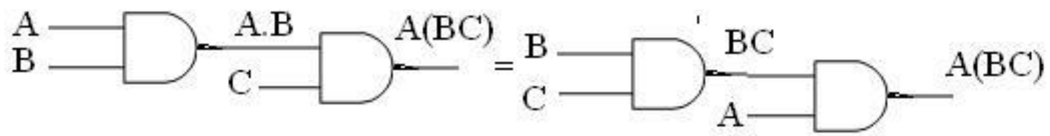
A B C	A+B	(A+B)+C
0 0 0	0	0
0 0 1	0	1
0 1 0	1	1
0 1 1	1	1
1 0 0	1	1
1 0 1	1	1
1 1 0	1	1
1 1 1	1	1

=

A B C	B+C	A+(B+C)
0 0 0	0	0
0 0 1	1	1
0 1 0	1	1
0 1 1	1	1
1 0 0	0	1
1 0 1	1	1
1 1 0	1	1
1 1 1	1	1

Law 2: (A.B).C =A(B.C)

This law can be extended to any no. of variables
(A.B.C).D=(A.B.C).D



A B C	AB	(AB)C
0 0 0	0	0
0 0 1	0	0
0 1 0	0	0
0 1 1	0	0
1 0 0	0	0
1 0 1	0	0
1 1 0	1	0
1 1 1	1	1

=

A B C	BC	A(BC)
0 0 0	0	0
0 0 1	0	0
0 1 0	0	0
0 1 1	1	0
1 0 0	0	0
1 0 1	0	0
1 1 0	0	0
1 1 1	1	1

Distributive Laws:

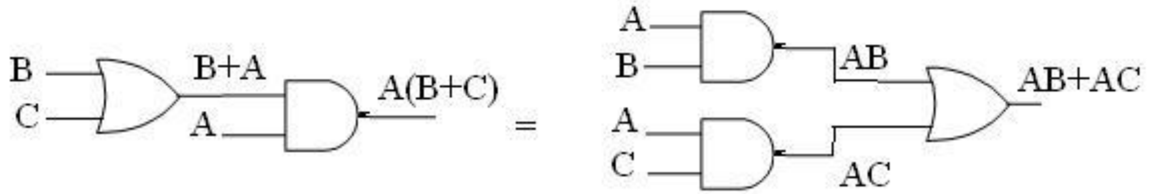
This has 2 laws

Law 1.A(B+C)=AB+AC

This law applies to single variables.

Ex:ABC(D+E)=ABCD+ABCE

AB(CD+EF)=ABCD+ABEF



A B C	B+C	A(B+C)
0 0 0	0	0
0 0 1	1	0
0 1 0	1	0
0 1 1	1	0
1 0 0	0	0
1 0 1	1	1
1 1 0	1	1
1 1 1	1	1

=

A B C	AB	AC	AB+AC
0 0 0	0	0	0
0 0 1	0	0	0
0 1 0	0	0	0
0 1 1	0	0	0
1 0 0	0	0	0
1 0 1	0	1	1
1 1 0	1	0	1
1 1 1	1	1	1

Law 2.A+BC=(A+B)(A+C) RHF=(A+B)(A+C)

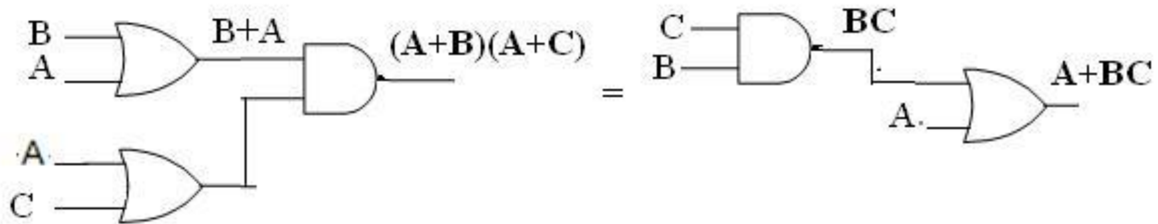
=AA+AC+BA+BC

=A+AC+AB+BC

=A(1+C+B)+BC

=A.1+BC

=A+BC LHF



A B C	BC	A+BC
0 0 0	0	0
0 0 1	0	0
0 1 0	0	0
0 1 1	1	1
1 0 0	0	1
1 0 1	0	1
1 1 0	0	1
1 1 1	1	1

A B C	A+B	A+C	(A+B)(A+C)
0 0 0	0	0	0
0 0 1	1	1	0
0 1 0	1	0	0
0 1 1	1	1	0
1 0 0	0	1	0
1 0 1	1	1	1
1 1 0	1	1	1
1 1 1	1	1	1

Redundant Literal Rule(RLR):

Law 1: A+ A'B=A+B

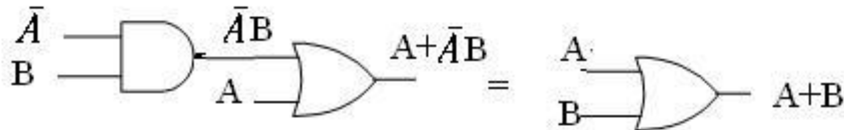
LHF =

(A+A')(A+B)

=1.(A+B)

=A+BRHF

Performing OR operation of a variable with the AND of the compliment of that variable with another variable, is equal to the Performing OR operation of the two variables.



A B	B	A+ B
0 0	0	0
0 1	1	1
1 0	0	1
1 1	0	1

=

A	B	A+B
0	0	0
0	1	1
1	1	1
1	1	1

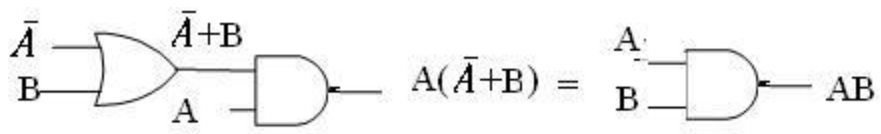
Law 2: A (A'+B) = AB

LHF = A.A' + AB

= 0+AB

=AB RHF

Performing AND operation of a variable with the OR of the complement of that variable with another variable, is equal to the performing AND operation of the two variables.



A B	A’+B	A(A’+B)
0 0	1	0
0 1	1	0
1 0	0	0
1 1	1	1

=

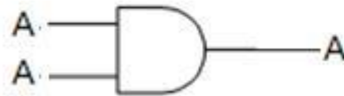
A	B	A+B
0	0	0
0	1	0
1	1	0
1	1	1

Idempotent Laws:

Idempotent means same value. It has 2 laws.

Law 1=A.A=A

This law statesperforming AND operation of a variable with itself is equal to that variable only

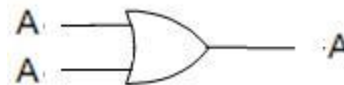


If A=0, then A.A=0.0=0=A

If A=1, then A.A=1.1=1=A

Law 2: A+A=A

This law states that performing OR operation of a variable with itself is equal to thatvariable only.



If A=0, then A+A=0+0=0=A

If A=1, then A+A=1+1=1=A

Absorption Laws:

Law 1=A+A.B=A

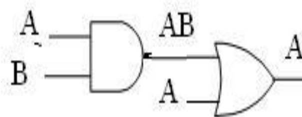
= A(1+B)

=A.1

=A

i.e., A+A. any term=A

A B	A.B	A+(A.B)
0 0	0	0
0 1	0	0
1 0	0	1
1 1	1	1



Law 2=A(A+B)=A

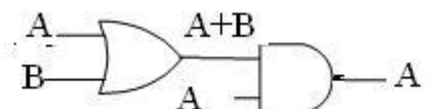
A(A+B)=A.A+A.B

= A+AB

=A(1+B)

= A.1 =A

A B	+	A(A+B)
0 0	0	0
0 1	1	0
1 0	1	1
1 1	1	1



Transposition Theorem:

AB+ A‘C=
(A+C)(A‘+B)
RHS = (A+C)(A‘ +B)

=AA‘ +CA‘ +AB+CB
=0+ A‘C+AB+BC
=A‘C+AB+BC(A+A‘)
=AB+ABC+ C+ BC =AB+ CLHS

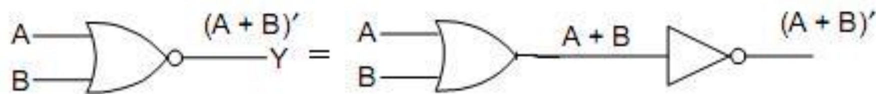
DeMorgan’s Theorem:

It represents two of the most powerful laws in Boolean algebra

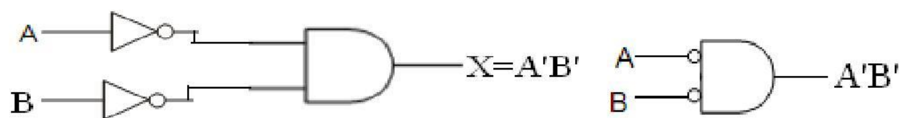
Law 1: (A+B)‘ = A‘.B‘

This law states that the compliment of a sum of variables is equal to the product of their individual complements.

LHS



RHS



NOR gate= Bubbled AND gate

This can be extended to any variables. (A+B+C+D+-----) ‘=A‘B‘C‘D‘----

Law 2: (AB)‘=A‘+B‘

Complement of the product of variables is equal to the sum of their individual components.

Duality:

In a positive Logic system the more positive of the two voltage levels is represented by a 1 & the more negative by a 0. In a negative logic system the more positive of the two voltage levels is represented by a 0 & more negative by a 1. This distinction between positive &negative logic systems is important because an OR gate in the positive logic system becomes an AND gate in the negative logic system &vice versa. Positive & Negative logics give a basic duality in Boolean identities. Procedure dual identity by changing all + (OR) to. (AND) & complementing all 0‘s &1‘s. Once a theorem or statement is proved, the dual also thus stands proved called Principle of duality.

Relations between complement

(A+B+C+….) ‘= A’. B’.C’ ….

(A.B.C.….) ‘= A’ + B’ + C’ + ….

Duals:

Expression	Dual
0=1	1=0
0.1=0	1+0=1
0.0=0	1+1=1
1.1=1	0+0=0
A.0=0	A+1=1
A.1=A	A+0=A
A.A=A	A+A=A
A.A' =0	A+A' =1
A.B=B.A	A+B=B+A
A.(B.C)=(A.B).C	A+(B+C)=(A+B)+C
A.(B+C)=(AB+AC)	A+BC=(A+B)(A+C)
A(A+B)=A	A+AB=A
A.(A.B)=A.B	A+A+B=A+B
(A+B)(A'+C)(B+C)=(A+B)(A'+C)	AB+ A'C+BC=AB+ A'C

Standard SOP and POS forms

Reducing Boolean Expressions:

Procedure:

1. Multiply all variables necessary to remove parenthesis
2. Look for identical terms. Only one of those terms to be retained & other dropped.

Ex: AB+AB+AB+AB=AB

3. Look for a variable & its negation in the same term. This term can be dropped 1

Ex: AB +AB = AB (+1)=AB .1=AB

4.Look for pairs of terms which have the same variables, with one or more variables complemented. If a variable in one term of such a pair is complemented while in the second term it is not then such terms can be combined into a single term with variable dropped.

Ex: AB +AB D= AB (+D)=AB .1=AB

Boolean functions & their representation:

A function of n Boolean variables denoted by $f(x_1,x_2,x_3\text{------}x_n)$ is another variable denoted by & takes one of the two possible values 0 & 1.
The various ways of representingthe given function is

1. Sum of Product(SOP) form:It is called the Disjunctive NormalForm(DNF)
Ex: f(A,B,C)= A.B'+ C'

2. Product of Sums (POS) form:It is called the Conjunctive Normal Form(CNF).This is implemented using Consensus theorem.

Ex:f(A,B,C)=(A+B)(B+C)

3. Truth Table form:The function is specified by listing all possible combinations of values assumed by the variables & the corresponding values of the function.

Ex: Truth table for f(A,B,C) = (B+ C)

Decimal Code	A	B	C	F(A,B,C)
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	0

4. Standard Sum of Products form called Disjunctive Canonical form (DCF) & also called Expanded SOP form or Canonical SOP form.

Ex: f(A,B,C)= A.B.C' +A.B.C

A product term contains all the variables of the function either in complemented or uncomplemented form is called a minterm. A minterm assumes the value 1 only for one combination of the variables. An n variable function can have in all 2^n minterms to 1 is the standard sum of products form of the function. Minterms are denoted as m_0, m_1,m_2….. Here suffixes are denoted by the decimal codes.

Ex: F(A,B,C)=m_1+m_2+m_3 then m_1=A'B'C , m_2=AB'C, m_3=A'BC

The function in DCF is listing the decimal codes of the minterms for which F=1

F(A,B,C)=$\sum$m(1,2,3,).

5. Standard Product of Sums form: It is called as Conjunctive Canonical form (CCF). It is also called Expanded POS or Canonical POS.

Ex: If A=0,B=0, C=0 and the term=0
Thus function f (A, B, C) = (A'+B'+C').(A+B'+C').(A+B+C')

A sum term which contains each of the n variables in either complemented form is called a *Maxterm.* A maxterm assumes the value '0'only for one combination of the variables. The most there are 2^n maxterms. It is represented as M_0,M_1,M_2…... Here the suffixes are decimal codes.

The CCF of f(A,B,C)=M_0.M_4.M_6

f(A,B,C)=πM(0,4,6,7) where π or ^ represents the product of all maxterms.

Expansion of a Boolean expression in SOP form to the standard SOP form:

1. Write down all the terms.
2. If one or more variables are missing in any term.Expand that term by multiplying it with the sum of each one of the missing variable and its complement.
3. Drop out redundant terms.

Expansion of a Boolean expression in POS form to standard POS form:

1. Write down all the terms.
2. If one or more variables are missing in any sum term. Expand that term by adding the product of each of the missing variable and its complement.
3. Drop out redundant terms.

Conversion between Canonical forms:

The complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function is expressed by those minterms that make the function equal to 1 for those minterms that make the function equal to 0.

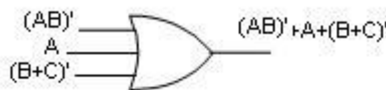
Ex: f(A,B,C)=∑m(0,2,4,6,7)
Complement isf'(A,B ,C) =∑m(1,3,5).=$m_1+m_3+m_5$
Complement of deMorgan's theorem: f= (m1 + m3 + m5) then f'= M1.M3.M5

1=M_j, the maxterm with subscript j is a complement of the minterm with the same subscript j and vice versa. To convert one canonical form to another, interchange the symbol ∑ and π, and list those numbers missing from the original form.

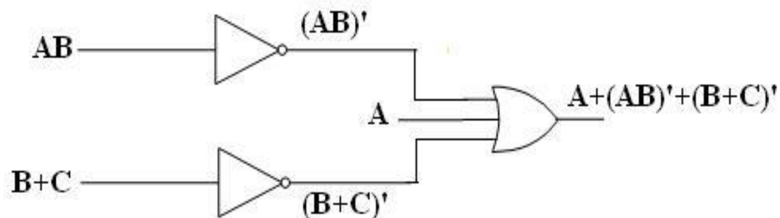
Two level NAND – NAND and NOR-NOR realizations:

Boolean expressions can be realized as hardware using logic gates. Conversely, hardware can be translated into Boolean expressions for the analysis of existing circuits.

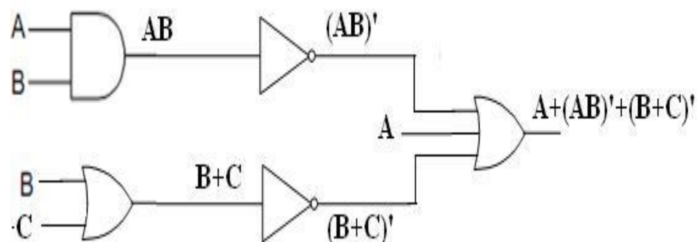
1. *Converting Boolean Expressions to Logic:*To convert, start with the output & work towards the input.Assume the expression (AB)'+A+(B+C)' is to be realized using AOI logic. Start with this expression. Since it is three terms, it must be the output of a three-input OR gates. So, draw an OR gate with three inputs as



(AB)' is the output of an inverter whose inputs is AB and (B+C)' must be the output of an inverter whose input is B+C. so, those two inverters are as



Now AB must be output of a two-input AND gate whose inputs are A and B. And B+C must be the output of a two-input OR gate whose inputs are B and C. so, an AND gate and an OR gate are as



2. *Converting Logic to Boolean Expressions:*

To convert logic to algebra, start with the input signals and develop the terms of the Boolean expression until the output is reached.

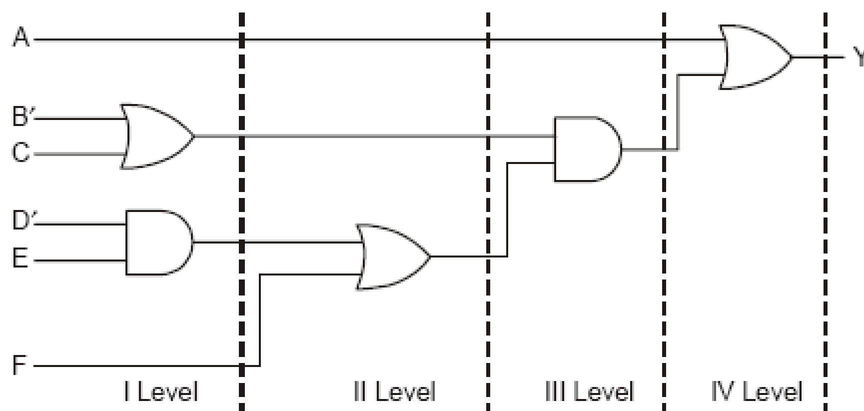
Since NAND logic and NOR logic are universal logic circuits which are first computed and converted to AOI logic may then be converted to either NAND logic or NOR logic depending on the choice. The procedure is

1. Draw the circuit in AOI logic
2. If NAND hardware is chosen, add a circle at the output of each AND gate and at the inputs to all the AND gates.
3. If NOR hardware is chosen, add a circle at the output of each OR gate and at the inputs to all the AND gates
4. Add or subtract an inverter on each line that received a circle in steps 2 or 3 so that the polarity of signals on those lines remains unchanged from that of the original diagram
5. Replace bubbled OR by NAND and bubbled AND by NOR Eliminate double inversions.

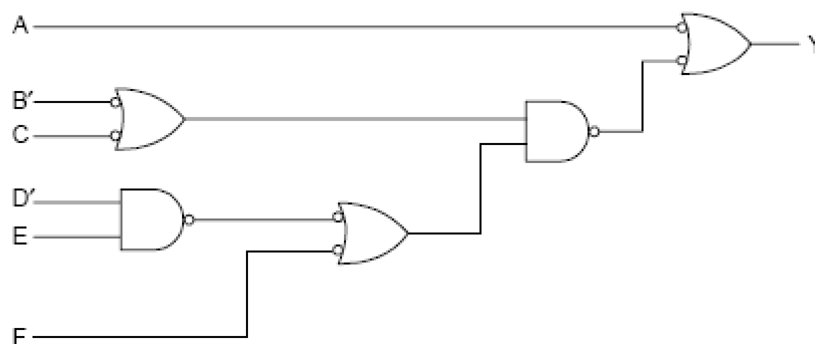
Ex: Now consider a Boolean function to demonstrate the procedure for converting into NAND gates:

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

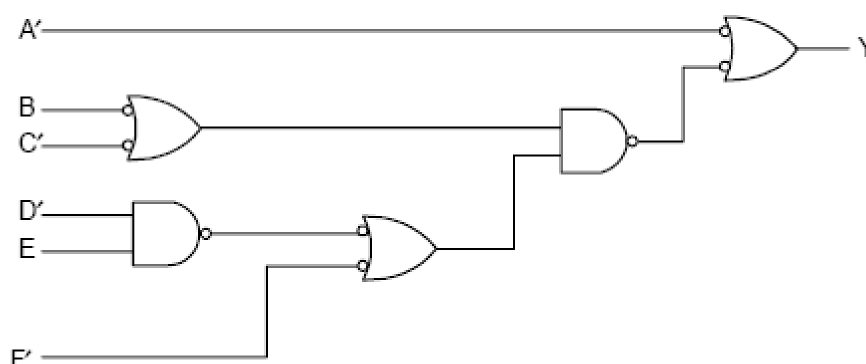
Step 1:We first draw the logic diagram using basic gates as shown in figure before



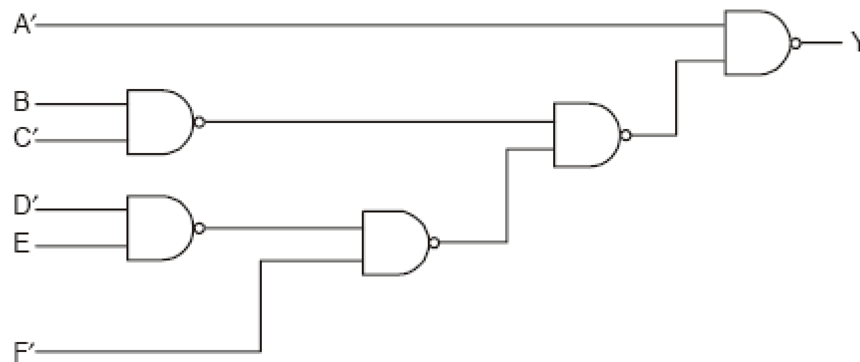
Step 2 and 3: Convert all AND gates to NAND using AND-invert symbol and all OR gates to NAND using Invert-OR symbol.



Step 4: It is very clear that only two inputs D′ and E are emerging in the original forms at the output. Rest inputs A, B′, C and F are emerging as the complement of their original form.

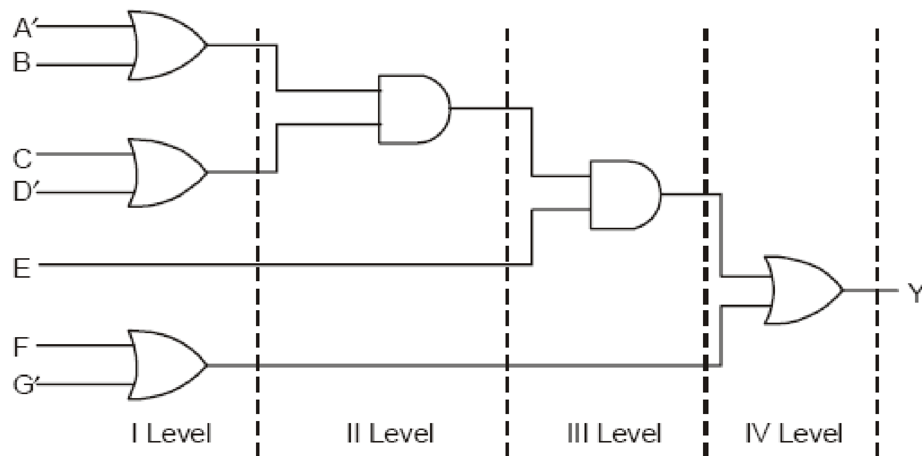


Step 5: Now because both the symbols AND-invert and invert-OR represent a NAND gate.



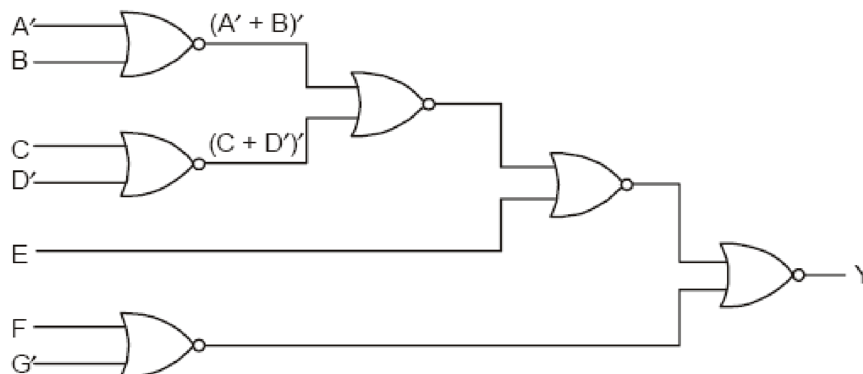
Ex: Now consider a Boolean function to for converting into NOR gates:

$$Y = ((A+B).(C+D))E+(F+G')$$



Convert all OR gates to NOR using OR-invert and all AND gates to NOR using invert AND symbol. Convert both symbols OR-invert and invent-AND represent a NOR gate

1 MINIMIZATION OF LOGIC FUNCTIONS USING BOOLEAN THEOREMS



2 THE KEYS TO BOOLEAN FUNCTION MINIMIZATION LIE IN THE THEOREMS INTRODUCED FOR BOOLEAN ALGEBRA. PARTICULARLY THE THEOREMS SHOWN BELOW ARE USEFUL.

(*a*) $A + AB = A$ (*b*) $A (A + B) = A$

(*c*) $A + A'B = A + B$ (*d*) $A (A' + B) = AB$

(*e*) $AB + AB' = A$ (*f*) $(A + B) (A + B') = A$

Ex: Minimize $F = CD + AB'C + ABC' + BCD$

3

4 A+AB=A

$F = CD + AB'C + ABC'$

Unit – II

Objectives:

- To familiarize with K-map method
- To understand various combinational logic circuits.

Syllabus:

Combinational Logic Circuits: The Map Method(upto 4 Variables), Don't care conditions, design procedure, adders, subtractors, 4-bit adder-subtractor circuit, BCD adder, carry look ahead adder, decoders and encoders, multiplexers, demultiplexers.

Outcomes:

Students will be able to

- Determine the minimized Boolean function using K-maps
- Design adders and subtractors.
- Understand 4-bit adders like BCD adder and look-a-head carry adder.
- Understand other combinational circuits like decoder, encoder, multiplexer, demultiplexer.

Learning Material

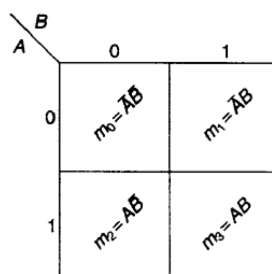
Minimization of switching functions using K (Karnaugh) -Maps

- The K-map is a diagram made up of squares.
- Each square represents one minterm. Since any function can be expressed as a sum of minterms, it follows that a Boolean function can be recognized from a map by the area enclosed by those squares, whose minterms are included in the operation.
- By various patterns, we can derive alternative algebraic expression for the same operation, from which we can select the simplest one. (One that has minimum number of literals).
- Construct the K-map as discussed above. Enter 1 in those squares corresponding to the minterms for which function value is 1. Leave empty the remaining squares. Now in following steps the square means the square with a value 1.
- Examine the map for squares that cannot be combined with any other squares and form group of such single squares.
- Now, look for squares which are adjacent to only one other square and form groups containing only two squares and which are not part of any group of 4 or 8 squares. A group of two squares is called a pair.
- Next, group the squares which result in groups of 4 squares but are not part of an 8-squares group. A group of 4 squares is called a quad.
- Group the squares which result in groups of 8 squares. A group of 8 squares is called octet.
- Form more pairs, quads and outlets to include those squares that have not yet been grouped, and use only a minimum no. of groups. There can be overlapping of groups if they include common squares.
- Omit any redundant group.
- Form the logical sum of all the terms generated by each group.
- Using Logic Adjacency Theorem we can conclude that,
    - ✓ a group of two squares eliminates one variable,
    - ✓ a group of four squares eliminates two variable and a group of eight squares eliminates three variables.

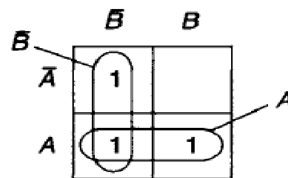
There are two, three and four variable K maps.

Two Variable K-Map:

- For two variables there are four minterms and these can be conveniently placed on a 'map' as shown in figure below



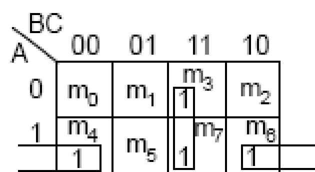
- The map consists of a square divided into four cells, one for each of the minterms.
- The possible values of the variable A are written down the left hand side of the map, labeling the corresponding rows of the map, while the possible values of the variable B are written along the top of the map, labeling the corresponding columns of the map.
- Hence, the top left-hand cell represents the minterm where A=0 and B=0, i.e. the minterm AB.
- The bottom right-hand cell represents the minterm AB where A=1and B=1.
- The process of simplifying a Boolean function with the aid of a K-map is simplya process of finding adjacencies on the function plot.
- This is best explained with the aidof a very simple example.
- Suppose that it is required to simplify the Boolean function f = A'B'+ AB' + AB.
- Using Boolean algebra alone, it can be readily found that F=B(A' + A) + AB = AB + B'
- However, suppose that F is plotted on a 2-variable K-map, as in Figure below.



- The next stage of the simplification process is to group together adjacent cells containing 1's. (In this context, note carefully that 'adjacent' means 'horizontally or vertically', *not* 'diagonally'.)
- Therefore, the bottom two cells, corresponding to A alone, may be grouped together.
- Similarly, the two left-hand cells, corresponding to B alone, may also be grouped together, as indicated in the figure above.
- The final stage is to write down the final simplified expression for the function obtained from the groupings thus identified. In this case, therefore, f = A + B'.

Three Variable K-Map:

- If the following Boolean function F (A, B, C) = Σ (3, 4, 6, 7).Then it is represented in k-map as shown in figure below:



Step 1. *m*3 is adjacent to *m*7. It forms a group of two squares and is not a part of any group of 4 or 8 squares. Similarly *m*6 is adjacent to *m*7. So this is second group (pair) that is not a part of any group of 4 or 8 squares. Now according to new definition of adjacency *m*4 and *m*6 are also adjacent and form a pair. Moreover, this pair (group) is not a part of any group of 4 or 8 squares.
Step 2. All the 1's have already been grouped.
Step 3. The pair formed by *m*6 *m*7 is redundant because *m*6 is already covered in pair*m*4 *m*6 and *m*7 in pair *m*3 *m*7. Therefore, the pair *m*6 *m*7 is discarded.

Step 4. The terms generated by the remaining two groups are 'OR' operated togetherto obtain the expression for F as follows:

$$F = AC' + BC$$

AC' ↓ From group $m_4\ m_6$. The row is corresponding to the value of A = 1 and in the two columns (00 → B′C′ and the 10 → BC′), the value C = 0 ⇒ C′ is common = AC′

BC ↓ From group $m_3\ m_7$. Correspond to both rows (A = 0 and A = 1) so A is omitted, and single column (B = 1 and C = 1), *i.e.*, BC.

Four Variable K-Map:

- If the following Boolean functionF(*w, x y, z*) = Σ (0, 2, 3, 6, 7, 8, 10, 11, 12, 15), then the K-map is given in the figure below

WX \ YZ	00	01	11	10
00	1		1	1
01			1	1
11	1		1	
10	1		1	1

  - Minterm 8 and 12. From a pair.
  - Minterms 0, 2, 8 and 10 form I quad.
  - Minterms 3, 7, 11, 15 form II quad.
  - Minterms 2, 3, 6, 7 form III quad.
  - Therefore the final minimized expression is given by

$$F = x'z' + yz + w'y + wy'z'$$

$x'z'$ ↓ Due to I quad. yz ↓ II quad. $w'y$ ↓ III quad. $wy'z'$ ↓ Due to pair.

Don't care map entries:

- The occurrence of particularinput combinations will have no effect onthe output, then those inputs are known as don't cares.
- That is a d or a × (cross) is entered into each square to signify "don'tcare" MIN/MAX terms.
- Simplify the following Boolean function. F (A, B, C, D) = Σ (0, 1, 2, 10, 11, 14)&d (5, 8, 9)

AB \ CD	00	01	11	10
00	1	1		1
01		X		
11				1
10	X	X	1	1

- As shown in K-map in Figure above, by combining 1's and d's(Xs), three quads can be obtained.
- The X in square 5 is left free since it does not contribute in increasing the size of any group. Therefore, the
    - I Quad covers minterms 0, 2, 10 and d8
    - II Quad covers minterms 10, 11 and d8, d9.
    - III Quad covers minterms 0, 1 and d8, d9.
    - A pair covers minterms 10 and 14.
- Therefore the final expression is

$$F = B'D' + AB' + B'C' + ACD'$$

	$B'D'$		AB'		$B'C'$		ACD'
$F =$	Due to I quad.	+	II quad	+	III Quad	+	Due to Pair.

Combinational circuit consists of logic gates whose outputs at anytime are determined directly from the present combination of inputs without regard to previous inputs.

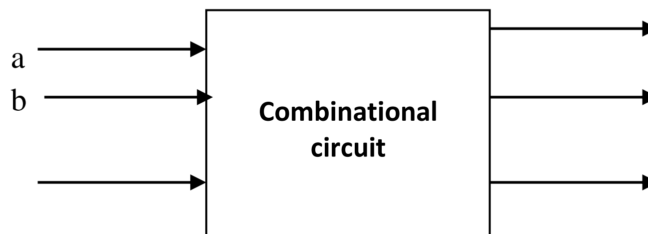
Combinational circuit is a combination of different gates.

For example: encoder, decoder, multiplexer and de-multiplexer etc. are some combinational circuits.

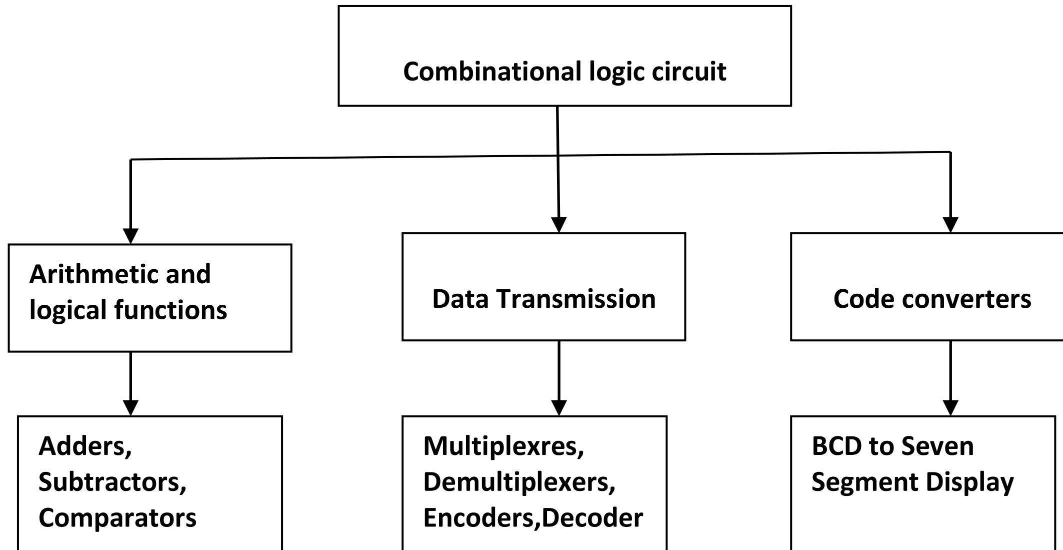
Some of the characteristics of combinational circuits are following:

- The output of combinational circuit at any instant of time depends only on the levels present at input terminals.
- The combinational circuit does not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- A combinational circuit can have a n number of inputs and m number of outputs.

Block diagram



Classification of combinational Logic Circuits:



Design procedure of combinational Logic circuit:

1. The problem is stated
2. The number of available input variables and required output variables is determined
3. The input and output variables are assigned letter symbols
4. The truth table that defines the required relationship between inputs and outputs is derived
5. The simplified Boolean function for each output is obtained
6. The logic diagram is drawn.

Adders:

Digital computers perform various arithmetic operations. The most basic arithmetic operation is the addition of two binary digits.

Different types of adders are discussed below:

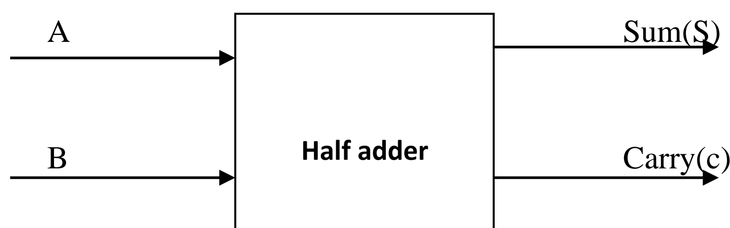
Half Adder

Half adder is a combinational logic circuit that performs the addition of two bits.

Half adder circuit needs two binary inputs and two binary outputs.

The input variables designate the augend and addend bits, the output variables produce the sum and carry.

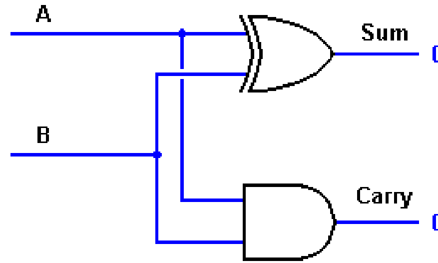
Block diagram



Truth Table

Inputs		Outputs	
A	**B**	**Sum(s)**	**Carry(c)**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Circuit Diagram:

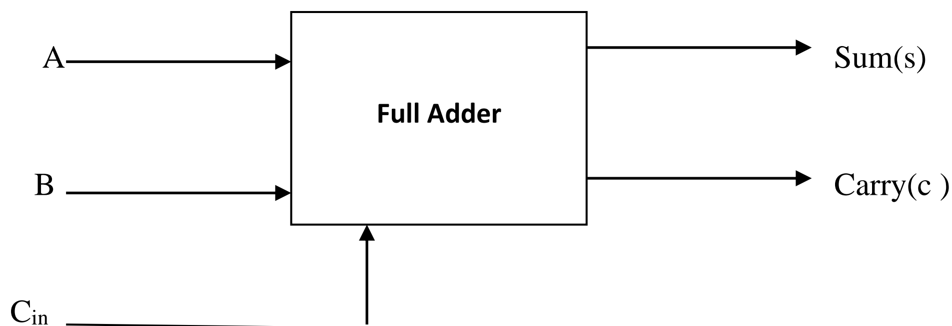


Logic Equations:
Sum(s)= A⊕B;   Carry(c)=AB;

Full Adder

The combinational circuit that performs the addition of three bits (two significant bits and previous carry) is called full adder. It consists of three inputs and two outputs. Two significant bits represented as A and B and the third input C_{in} represents the carry from the previous lower significant position. The two outputs are Sum (s) and Carry(c).

Block diagram:

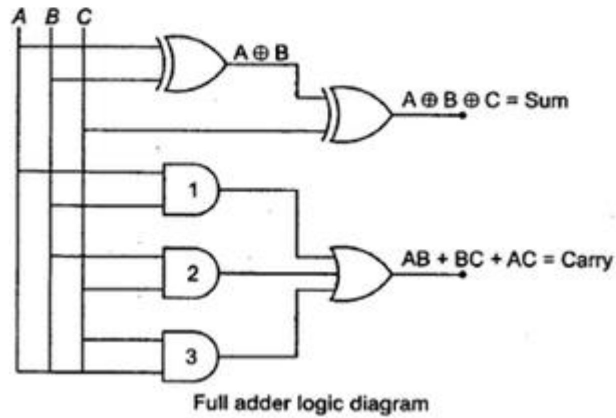


Truth Table

Inputs			outputs	
A	**B**	**C_{in}**	**Sum**	**Carry**
0	0	0	0	0
0	0	1	1	0

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

Circuit Diagram:



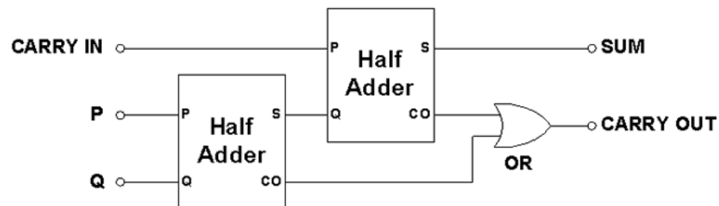
Full adder logic diagram

Logic Equations:

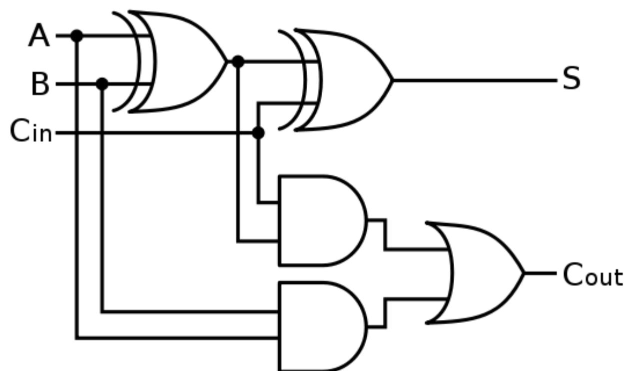
Sum(s)=$A \oplus B \oplus C_{in}$

Carry(c)=$AB+BC_{in}+AC_{in}$

Full adder implementation using two halfadders and orgate:



Logic Diagram:



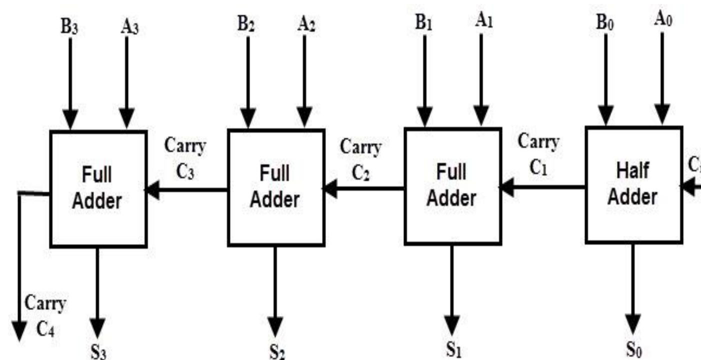
N-Bit Parallel Adder

The Full Adder is capable of adding only two single digit binary number along with a carry input. But in practice we need to add binary numbers which are much longer than just one bit. To add two n-bit binary numbers we need to use the n-bit parallel adder. It uses a number of full adders in cascade. The carry output of the previous full adder is connected to carry input of the next full adder.

4 Bit Parallel Adder

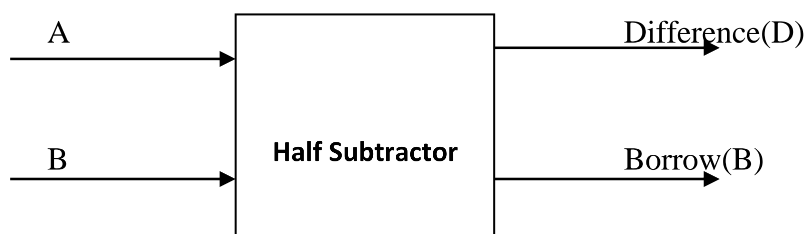
In the block diagram, A0 and B0 represent the LSB of the four bit words A and B. Hence Full Adder-0 is the lowest stage. Hence its Cin has been permanently made 0 .The rest of the connections are exactly same as those of n-bit parallel adder is shown in fig. The four bit parallel adder is a very common logic circuit.

Block diagram:



Half Subtractor

A Half subtractor is a combinational circuit that subtracts two bits and produces their difference. It produces the difference between the two binary bits at the input and also produces an output Borrow to indicate if a1 has been borrowed. In the subtraction A−B, A is called as Minuend bit and B is called as Subtrahend bit.

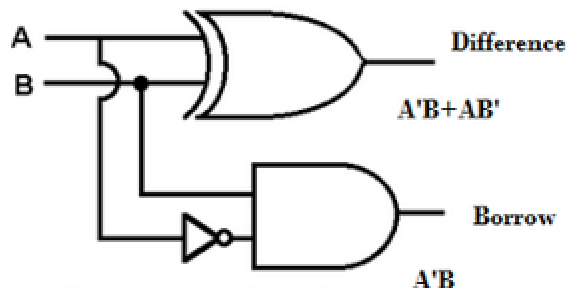


Truth Table

Inputs		Outputs	
A	**B**	**Difference(D)**	**Borrow(B)**
0	0	0	0

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

Logic Diagram:

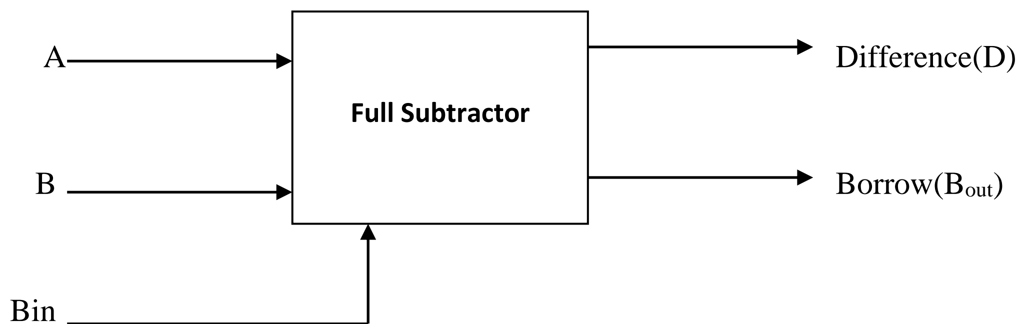


Logic Equations:

difference(D)= A⊕B; Borrow(B)=A^1B;

Full Subtractor:

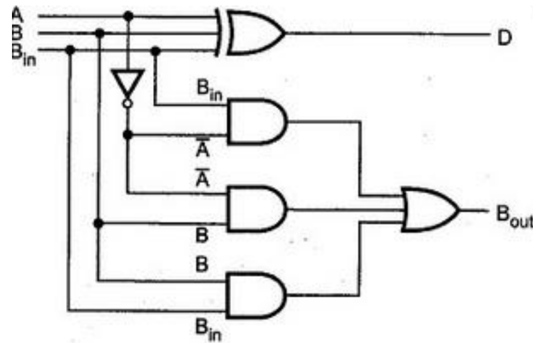
The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A, B, B_{in} and two output D and B_{out}. A is the 'minuend', B is 'subtrahend', C is the 'borrow' produced by the previous stage, D is the difference output and B is the borrow output.



Truth Table

Inputs			outputs	
A	**B**	**B_{in}**	**D**	**B_{out}**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Circuit Diagram



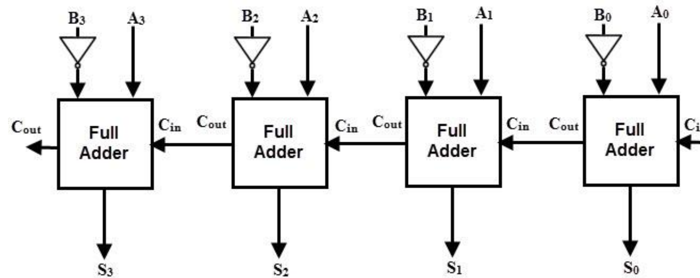
Logic Equations:

difference(D)= $A \oplus B \oplus B_{in}$

4 Bit Parallel Subtractor

The number to be subtracted B is first passed through inverters to obtain its 1's complement. The 4-bit adder then adds A and 2's complement of B to produce the subtraction. $S_3S_2S_1S_0$ represents the result of binary subtraction A−B and carry output Cout represents the polarity of the result. If A>B then Cout=0 and the result of binary form A−B then Cout=1 and the result is in the 2's complement form.

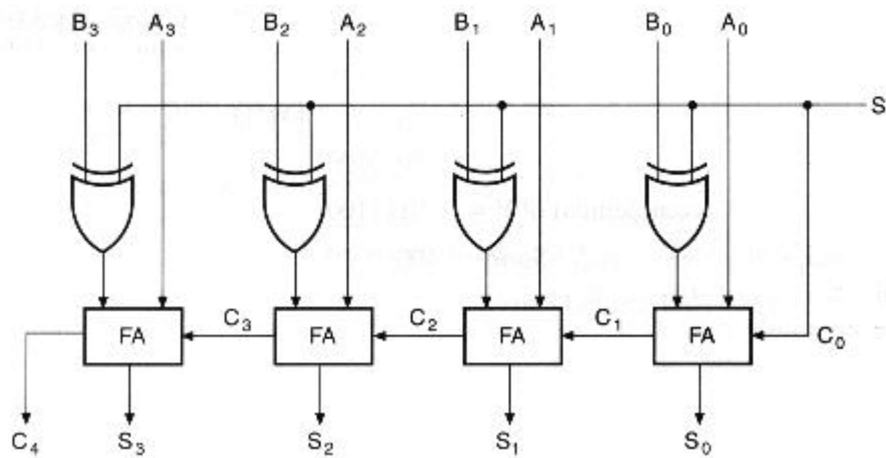
Block diagram



N-Bit Parallel Subtractor

The subtraction can be carried out by taking the 1's or 2's complement of the number to be subtracted. For example we can perform the subtraction A−B by adding either 1's or 2's complement of B to A. That means we can use a binary adder to perform the binary subtraction.

4-Bit Adder-Subtractor Circuit



This figure represents a 4-bit adder-subtractor circuit. Here the addition and subtraction operations are combined in to one circuit with one common binary adder. The mode input S controls the operation.

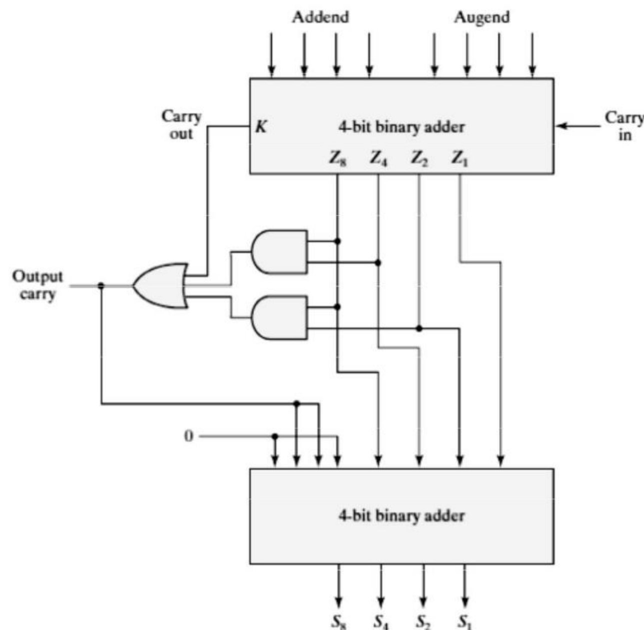
When S=0, the circuit is an adder

When S=1, the circuit is a subtractor

Each XOR gate receives input S and one of the inputs of B. when S=0, we have $B \oplus 0 = B$. the full adder receives the value of B, the input carry is 0 and the circuit performs A+B.

When S=1, we have $B \oplus 1 = B'$ and C1=1. The B inputs are complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B.

BCD Adder



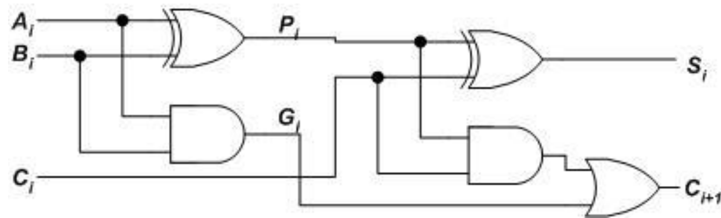
The logic circuit that detects the necessary correction can be derived from the entries in the table. It is obvious that a correction is needed when the binary sum has an output carry K = 1. The other six combinations from 1010 through 1111 that need a correction have a 1 in position Z8. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z8, we specify further that either Z 4 or Z2 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

When C = 1, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage.

Look Ahead Carry Adder

The Figure shows the full adder circuit used to add the operand bits in the ith column; namely Ai & Bi and the carry bit coming from the previous column (Ci).



In this circuit, the 2 internal signals Pi and Gi are given by:

$$Pi = Ai \quad Bi \text{ ……………………..(1)}$$

$$Gi = Ai\, Bi \text{ …………………..……(2)}$$

The output sum and carry can be defined as :

$$Si = Pi \quad Ci \text{ ……………………(3)}$$

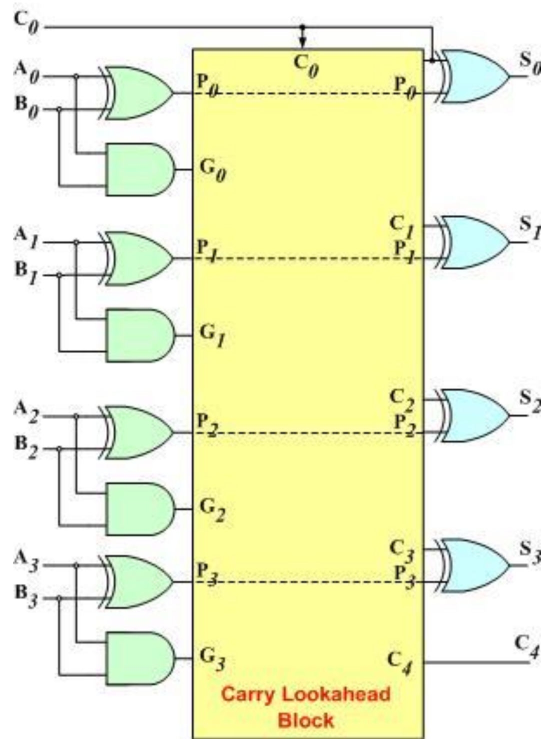
$$C\, i+1 = Gi \quad Pi\, C\, i \text{ …………(4)}$$

$\boldsymbol{G_i}$ is known as the ***carry Generate*** signal since a carry (C_{i+1}) is generated whenever G_i =1, regardless of the input carry (C_i). $\boldsymbol{P_i}$ is known as the ***carry propagate*** signal since whenever P_i =1, the input carry is propagated to the output carry, i.e., C_{i+1}. = C_i (note that whenever P_i =1, G_i =0). Computing the values of P_i and Gi only depend on the input operand bits (Ai & Bi) as clear from the Figure and equations. Thus, these signals settle to their ***steady-state value*** after the propagation through their respective gates.

$C_1 = G_0 + P_0C_0$
$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$
$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$ C_4
$= G_3 + P_3C_3= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

In general, the ith carry output is expressed in the form $\mathbf{C}_i = F_i$ (P's, G's , $\mathbf{C_0}$).
In other words, each carry signal is expressed as a direct SOP function of C_0 rather than its preceding carry signal.
Since the Boolean expression for each output carry is expressed in SOP form, it can be implemented in two-level circuits.
The 2-level implementation of the carry signals has a propagation delay of 2 gates,

The 4-bit carry look-ahead (CLA) adder consists of 3 levels of logic:



Multiplexers:

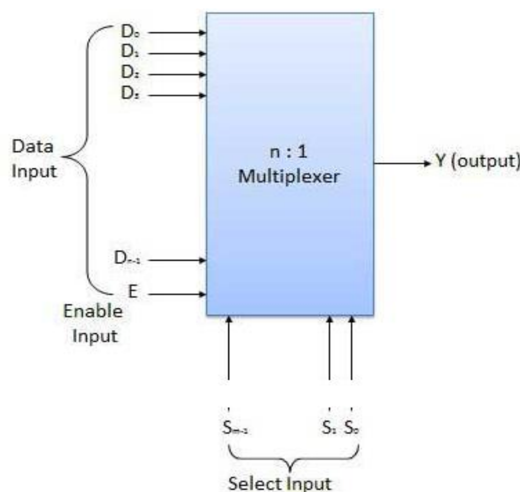
A digital multiplexer is a combinational circuit that selects binary information fron one of many input lines and directs it to a single output line.

The selection of a particular input line is controlled by a set of selection lines.

Normaliy there are 2^n input lines and an selection lines whose bit combinations determine which input is selected.

E is called the strobe or enable input which is useful for the cascading. It is generally an active low terminal that means it will perform the required operation when it is low.

Block diagram



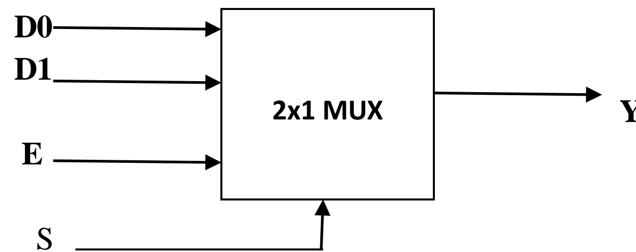
Multiplexers come in multiple variations

- 2:1 multiplexer
- 4:1 multiplexer
- 16:1 multiplexer
- 32:1 multiplexer

2X1 Multiplexer:

A 2 to 1 line multiplexer consists of 2 input lines and one select line and single output line.

Block Diagram:



Truth Table:

Enable	Select	Output
E	**S**	**Y**
0	X	0
1	0	D_0
1	1	D_1

X=Don't Care

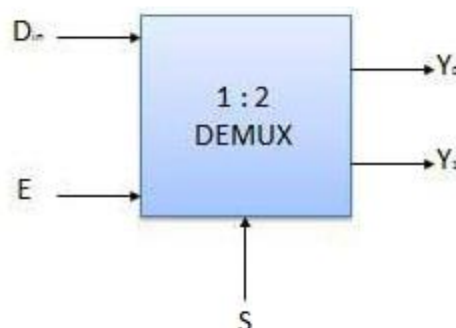
Demultiplexers

A demultiplexer performs the reverse operation of a multiplexer i.e. it receives one input and distributes it over several outputs. It has only one input, n outputs, m select input. At a time only one output line is selected by the select lines and the input is transmitted to the selected output line.

Demultiplexer comes in multiple variations.

- 1:2 demultiplexer
- 1:4 demultiplexer
- 1:16 demultiplexer
- 1:32 demultiplexer

Block diagram



Truth Table

Enable	Select	Output	
E	S	Y0	Y1
0	x	0	0
1	0	0	D_{in}
1	1	D_{in}	0

x = Don't care

Decoder

A decoder is a combinational circuit. It has n input and to a maximum= 2^n 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

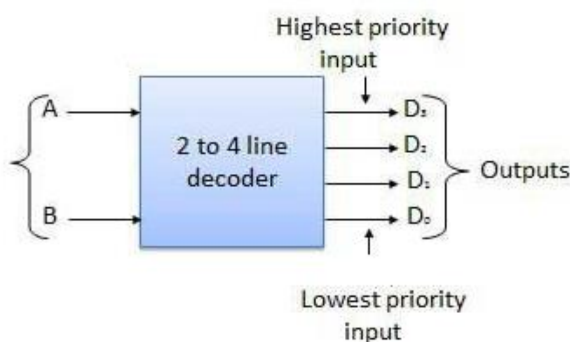
Examples of Decoders are following.

- 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_3 through D_0 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.

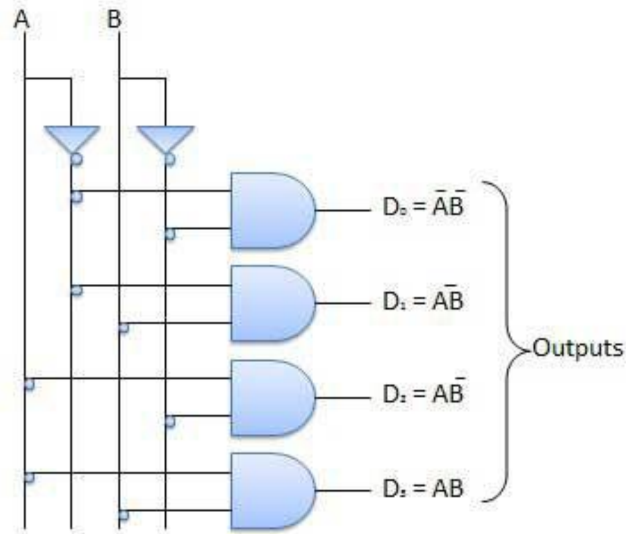
Block diagram



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

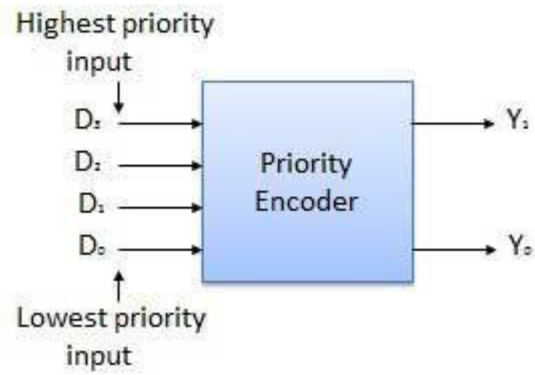
Examples of Encoders are following.

- 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 D0, D1, D2, D3 and two output Y0, Y1. Out of the four input D3 has the highest priority and D0 has the lowest priority. That means if D3=1 then Y1Y1=11 irrespective of the other inputs. Similarly if D3= 0 and D2=1 then Y1Y0=10 irrespective of the other inputs.

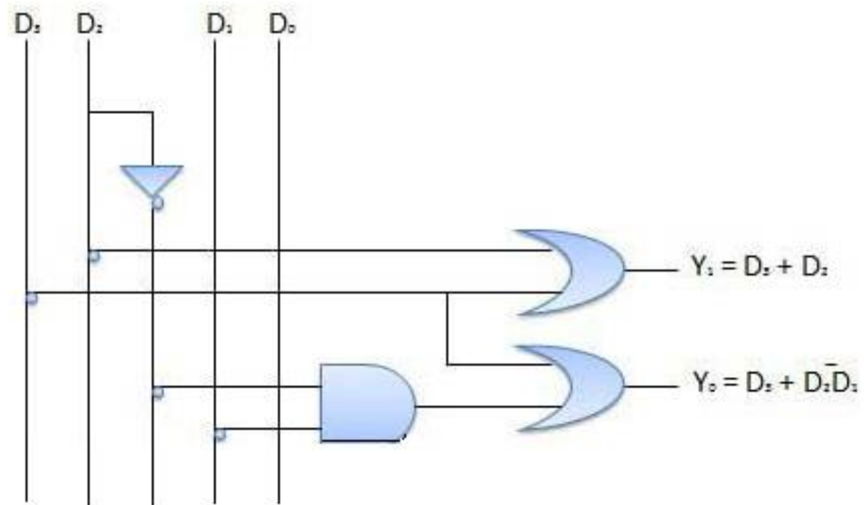
Block diagram



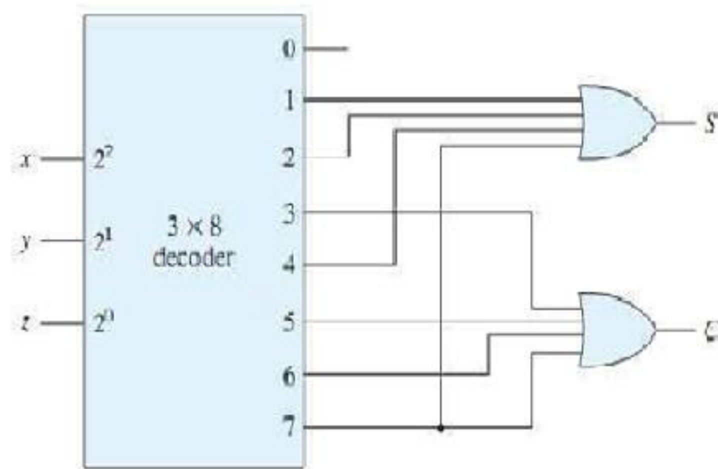
Truth Table

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



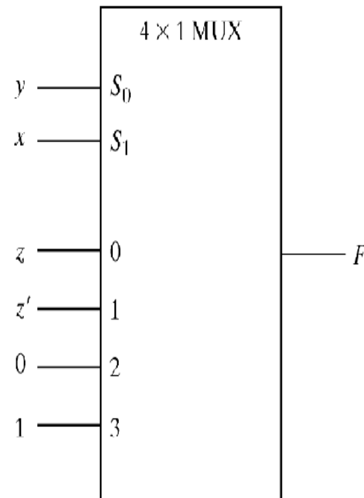
- **Implement full adder circuit whose outputs are given as : S (x,y,z) = Σ (1,2,4,7) C(x,y,z) = Σ(3,5,6,7) with a suitable decoder and external gates**



- **Implement the function F(x,y,z) = m(1,2,6,7) x, and y should be connected with the same order to S1and S0respectively**

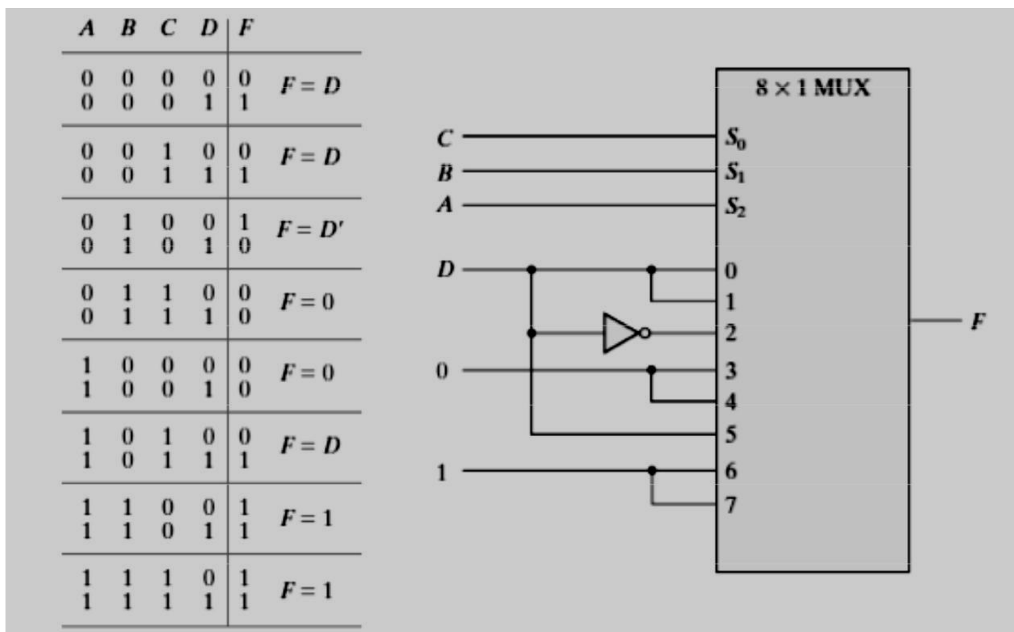
x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

- **Implementation of the Boolean function F (A, B, C, D) = ∑(1, 3, 4, 11, 12, 13, 14, 15) using 8 X 1 MUX**



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

Assignment-Cum-Tutorial Questions

Section-A

1. A full subtractor circuit requires ________.
   a) Two inputs and two outputs b) Two inputs and three outputs
   c) Three inputs and one output d)Three inputs and two outputs
2. A de multiplexer has ________.
   a) One data input and a number of selection inputs, and they have several outputs
   b) One input and one output
   c) Several inputs and several outputs
   d) Several inputs and one output
3. How many outputs are on a BCD decoder?
   a) 4 b) 16 c) 8 d) 10
4. A decoder converts ________.
   a) Non coded information into coded form b) Coded information into non coded form
   c) HIGHs to LOWs d) LOWs to HIGHs
5. Parallel Adders are
   a) Combinational logic circuits b) Sequential logic circuits
   c) Both of the above d) None of the above
6. A Full Adder can be realized using
   a) One half adder, two OR gates b) Two half adders, one OR gate
   c) Two half adders, two OR gates d) Two half adders, one AND gate
7. In which of the following adder circuits is the carry ripple delay is eliminated?
   a) Half adder b) Full adder c) Parallel adder d) Carry-look-ahead-adder
8. A multiplexer is also known as
   a) data accumulator b) data restorer c) data selector d) data distributor
9. Which logic device is called a distributor?
   a) Multiplexer b) Demultiplexer c) Encoder d) Decoder
10. How many inputs are on a BCD decoder?
    a) 4 b) 16 c) 8 d) 10
11. Which digital system translates coded characters into a more useful form?
    a) Encoder b) Display c) Counter d) Decoder
12. If f(A,B,C,D)=1 then the K-map contains _________ number of logic 1's is
    a) 4 b) 8 c) 16 d) 32
13. In a hexadecimal to binary priority encoder, ______ has ______ priority.
    a) 0, high b) 7, low c) F, low d) F, high
14. What control signals may be necessary to operate a 1-line-to-16 line decoder?

a) Flasher circuit control signal   b) A LOW on all gate enable inputs

c) Input from a hexadecimal counter   d) A HIGH on all gate enable circuits

15. The logic function implemented by the circuit below is (ground implies a logic "0")

a) F= AND (P, Q)   b) F= OR (P, Q)   c) F= XNOR (P, Q)   d) F= XOR (P, Q)

16. A digital system is required to amplify a binary-encoded audio signal. The user should be able to control the gain of the amplifier from minimum to a maximum in 100 increments. The minimum number of bits required to encode, in straight binary, is

a) 8   b) 6   c) 5   d) 7

17. The minimum number of 2-input NAND/NOR gates required to realize a half adder is

a) 3   b) 4   c) 5   d) 6

18. The minimum number of 2-input NAND gates required to realize a full adder / full subtractor is

a) 8   b) 9   c) 10   d) 12

19. The K – map for a Boolean function is shown in the figure. The number of essential prime implicants for this function is

ab \ cd	00	01	11	10
00	1	1	0	1
01	0	0	0	1
11	1	0	0	0
10	1	0	0	1

a) 4   b) 5   c) 6   d) 7

20. A function F(A,B,C) contains minterms 1,2,3,5,6,7, its complement contains

a) Σm (0,4)   b) Σm(1,2,3,5,6,7)   c) ΠM (1,2,3,5,6,7)   d) ΠM (0,4)

Section-B

1. Simplify the following three variable Boolean expression using karnaugh map method.

   Y= ABC' +A'B'C'+ABC+AB'C'.

2. Using K-Map simplify the following Boolean function

   F=A'BC+ABC'+ABC+AB'C'

3. Define combinational logic? Write the design procedure for combinational circuits.
4. Explain the operation of half adder? Realize full adder using logic gates.

5. Explain the operation of half subtractor? Realize full subtractor using logic gates.
6. Discuss the functional principle of 4-bit ripple carry adder. what is its major disadvantage?
7. What is decoder? Draw the logic diagram of 3 to 8 line decoder and explain its operation.
8. What is the difference between encoder and priority encoder? Give the implementation of a 4-bit priority encoder?
9. Discuss how four bit BCD adder circuit is designed. Explain its operation.
10. Briefly describe the concept of look-ahead carry generation with respect to its use in adder circuits.
11. Draw the circuit diagram of a 4-bit adder/subtractor and briefly describe its functional principle.
12. Implement the following function with 8 to 1 multiplexer:
13. Implement the three-variable Boolean function using an 8-to-1multiplexer
14. Realize the logic expression given below using a (i) 8:1 MUX (ii) 16:1 MUX

    $f=\sum m(0,1,3,5,8,11,12,14,15)$
15. Design a 32:1 multiplexer using two 16:1 and 2:1 multiplexers. Implement the following multiple output combinational logic circuit using a 4 to 16 decoder:

    $F1=\sum m(0,1,4,7,12,14,15)$, $F2=\sum m(1,3,6,9,12)$, $F3=\sum m(2,3,7,8,10)$ and $F4=\sum m(1,3,5)$
16. Implement the full adder sum and carry functions with decoder and multiplexers.
17. Develop a 3-to-8 line decoder using NOR gates only, and draw its logic diagram.
18. A combinational circuit is defined by the following equations:

    $f1 = AB + A'B'C'$, $f2 = A + B + C'$, $f3 = A'B + AB'$. Design a circuit which will implement these three equations using a decoder and NAND gates external to the decoder.
19. Design a combinational circuit that detects an error in the representation of a decimal digit in BCD. The output of the circuit must be equal to logic 1 when the inputs contain any one of the six unused bit combinations in the BCD code.
20. A combinational circuit is defined by the following three functions $F_1 = x'y'+xyz'$, $F_2 = x'+y$, $F_3=xy+x'y'$. Design the circuit with a decoder and external gates.
21. A logic function has four inputs A, B, C and D that will produce output 1 whenever two adjacent input variables are 1's. Treat A and D are also adjacent. Implement this logic function using 8 x 1 and 4 x 1 multiplexers.
22. Obtain logical functions to design decimal to octal using priority encoder.
23. Obtain the minimal expression for $\Sigma m(2,3,5,7,9,11,12,13,14,15)$ and implement it in NOR logic

24. Obtain the minimal expression for ΠM(2,3,5,7,9,11,12,13,14,15) and implement it in NAND logic

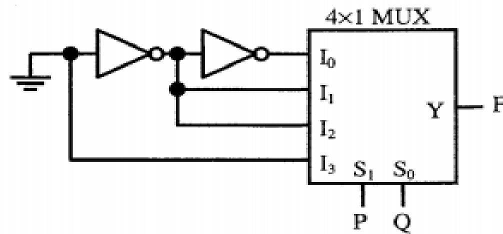
25. With the use of maps, find the simplest sum-of-products form of the function F = fg where f = abc' + c'd + a'cd' + b'cz' and g = (a + b + c' + d')(b' + c' + d)(a' + c + d')

Section-C

1. The output Y of a 2 bit comparator is logic 1whenever 2-bit input A is greater than 2-bit input B. The no. of combinations for which the output is logic 1 is **GATE-2012**

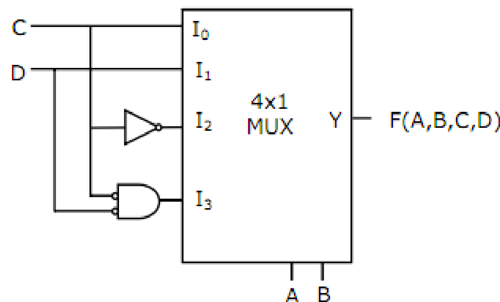
   a) 4 b) 6 c) 8 d) 10

2. The logic function implemented by the circuit below is (ground implies a logic '0') **GATE-2011**



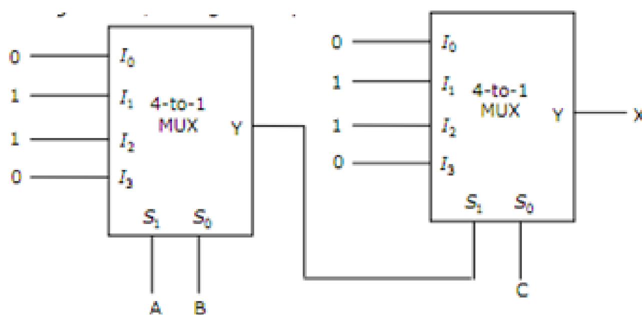
   a) F= AND (P,Q) b) F= OR (P,Q) c) F= XNOR (P,Q) d) F= XOR (P,Q)

3. The Boolean function realized by the logic circuit shown is **GATE-2010**



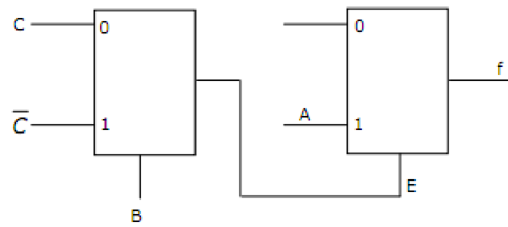
   a) F= Σm(0,1,3,5,9,10,14)
   b) F= Σm(2,3,5,7,8,12,13)
   c) F= Σm(1,2,4,5,11,14,15)
   d) F= Σm(2,3,5,7,8,9,12)

4. In the following circuit X is given by **GATE-2007**



a) X=AB'C'+A'BC'+A'B'C+ABC. b) X= AB'C'+A'BC'+A'B'C+ABC
c) X=AB +BC+AC d) X=A'B'+B'C'+A'C'

5. The Boolean function f implemented in figure using two input multiplexers is **GATE-2005**

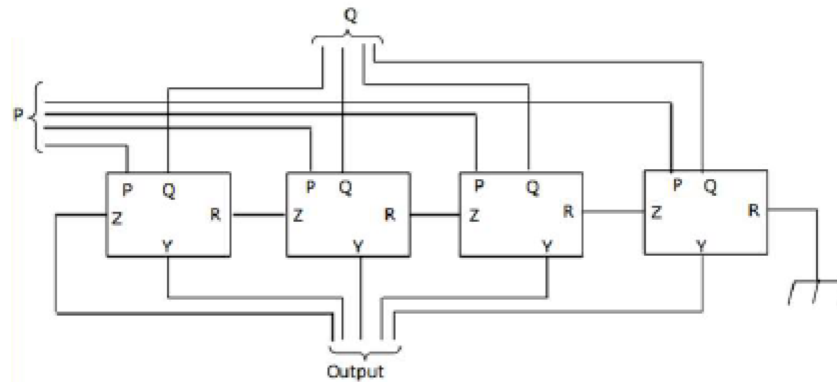


a) AB'C + ABC' b) ABC + AB'C' c) A'BC + A'B'C' d) A'B'C + A'BC'

6. The minimum no. of 2:1 multiplexers required to realize a 4:1multiplexer is **GATE-2004**

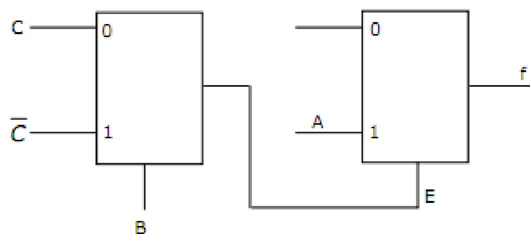
a) 1 b) 2 c) 3 d) 4

7. The circuit shown in figure below has 4 boxes each described by inputs P, Q, R and outputs Y, Z with Y= P Å Q Å R; Z=RQ+ P'R+Q P'. The circuit acts as a **GATE-2003**



a) 4 bit adder giving P+Q
b) 4 bit subtractor giving P-Q
c) 4 bit subtractor giving Q-P
d) 4 bit adder giving P+Q+R

8. The Boolean function f implemented in figure using 2 input multiplexers is **GATE 2005**

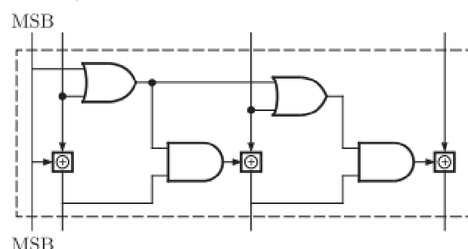


a) AB'C+ABC' b) ABC+AB'C' c) A'BC+A'B'C' d) A'B'C+A'BC'

9. The minimum number of 2 to 1MUX requires to realize a 4 to 1 MUX are **GATE 2004**

a) 1 b) 2 c) 3 d) 4

10. The circuit shown in figure converts **GATE 2003**



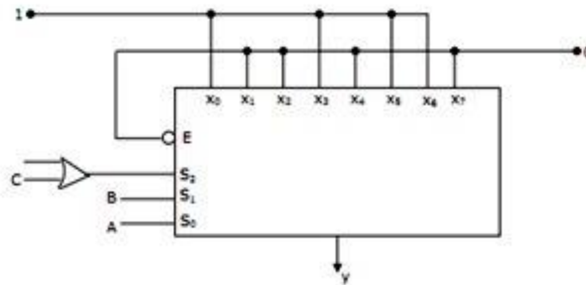
a) BCD to binary code b) Binary to Excess-3 c) Excess-3 to gray d) Gray to binary

11. Without any additional circuitry, an 8:1 MUX can be used to obtain **GATE 2003**

a) Some but not all Boolean functions of 3 Variables
b) All functions of 3 variables but not of 4 variables
c) All functions of 3 variables and some but not all functions of 4 variables
d) All functions of 4 variables

12. In the TTL circuit in figure below s_2 to s_0 are select lines and x_7 and x_0 are input lines . s_0 and x_0 are LSB's. The output Y is **GATE-2001**

a) indeterminate b) A⊕B c) (A⊕B)' d) C'(A⊕B) + C(A⊕B)'



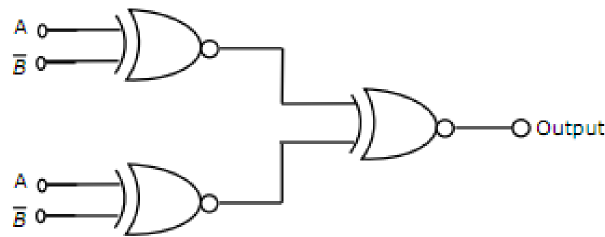
13. For a binary halfsubtractor having inputs A and B ,The correct set of logical expressions for the outputs D and X are

a) D=AB+A'B, X=A'B c) D=A'B+AB', X=A'B **GATE-1999**
b) D=A'B+AB', X=AB' d) D=AB+A'B', X=AB'

14. A 2 bit binary multipler can be implemented using **GATE-1997**

a) 2 inputs AND only b) 2 input XORS and 4 input AND gates only
c) Two 2 inputs NORS and one XOR gate d) XOR gates and shift registers

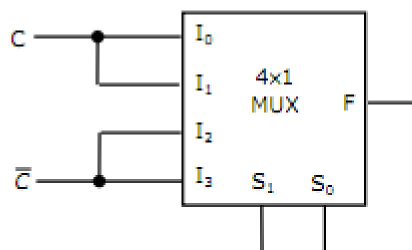
15. The output of the circuit shown in figure is equal to **GATE-1995**



a) 0 b) 1 c) A'B+AB' d) (AB)'.(AB)'

16. The logic realized by the circuit shown in figure is: **GATE-1992**

a) F=A.C b) F=A+C c) F=B.C d) F=B+C



17. The following boolean expression Y=

$\overline{A}\,\overline{B}\,\overline{C}D + \overline{A}BC\overline{D} + A\overline{B}\,\overline{C}D + AB\overline{C}\,\overline{D}$ can be minimized to

GATE-2007

a) $\overline{A}\,\overline{B}\,\overline{C}D + \overline{A}B\overline{C} + A\overline{C}D$ b) $\overline{A}\,\overline{B}\,\overline{C}D + BC\overline{D} + A\overline{B}\,\overline{C}D$

c) $\overline{A}BC\overline{D} + \overline{B}\,\overline{C}D + A\overline{B}\,\overline{C}D$ d) $\overline{A}BC\overline{D} + \overline{B}\,\overline{C}D + AB\overline{C}\,\overline{D}$

18. The number of product terms in the minimized sum of product expression obtained through the following karnaugh map (where d indicates don't care conditions).

GATE-2006

1	0	0	1
0	D	0	0
0	0	D	1
0	0	0	1

a) 2 b)3 c)4 d)5

19. The boolean expression for the truth table shown is **GATE-2005**

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

a) $B(A + C)(\overline{A} + \overline{C})$ c) $B(A + \overline{C})(A + C)$
b) $\overline{B}(A + \overline{C})(\overline{A} + C)$ d) $B(A + C)(\overline{A} + \overline{C})$

20. The Boolean expression $AC + B\overline{C}$ is equivalent to **GATE-2004**

a) $\overline{A}C + B\overline{C} + AC$ c) $\overline{B}C + AC + B\overline{C} + \overline{A}C\overline{B}$
b) $AC + B\overline{C} + \overline{B}C + ABC$ d) $ABC + \overline{A}B\overline{C} + AB\overline{C} + \overline{A}BC$

UNIT – III

Sequential Logic Circuits

Objectives:

- To familiarize with the concepts of different sequential circuits.

Syllabus:

Design procedure, Flip-flops, Truth tables and excitation tables, Conversion of flip-flops, Design of counters, Ripple counters, Synchronous counters, Ring counter, Johnson counter, Registers, Shift registers, Universal shift register.

Outcomes:

Students will be able to

- understand the functionality of different latches and flip-flops..
- distinguish the working of latch and flip-flop.
- convert from one flip-flop to another flip-flop
- classify various types of registers.
- design synchronous and asynchronous counters.

Learning Material

INTRODUCTION:

- Combinational circuits are those whose output at any instant of time is entirely dependent on the input present at that time.

- On the other hand Sequential circuits are those in which output at any given time is not only dependent on the present input but also on previous outputs. Naturally, such circuits must record the previous outputs which give rise to memory.

- Often, there are requirements of digital circuits whose output remain unchanged, once set, even if the inputs are removed. Such devices are referred as “memory elements”, each of which can hold 1-bit of information. These binary bits can be retained in the memory indefinitely (as long as power is delivered) or until new information is feeded to the circuit.

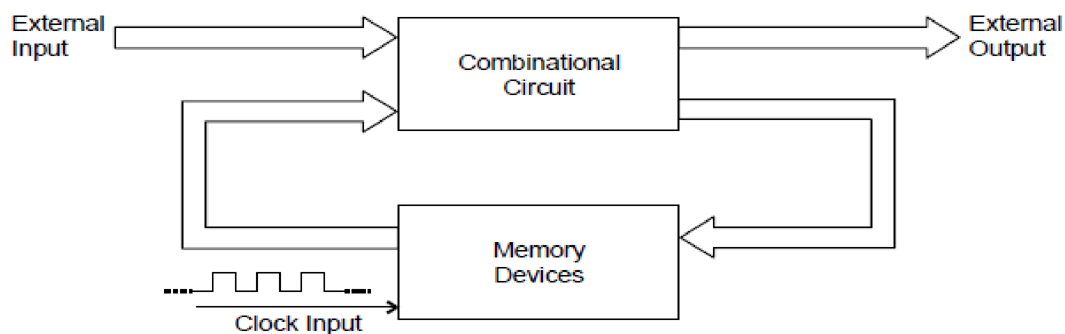


Fig 1: Block diagram of a sequential circuit

- Block diagram of a sequential circuit, which can be regarded as a collection of memory elements and combinational circuit as shown in above Fig.1.

- A feedback path is formed by using memory elements, input to which is the output of combinational circuit.

- The binary information stored in memory element at any given time is defined as the **state** of sequential circuit at that time. Present contents of memory elements are referred as the **present state.**

- The combinational circuit receives the signals from external input and from the memory output and determines the external output.
- They also determine the condition and binary values to change the state of memory. The new contents of the memory elements are referred as **next state** and depend upon the external input and present state.
- Hence, a sequential circuit can be completely specified by a time sequence of inputs, outputs and the internal states. In general, clock is used to control the operation. The clock frequency determines the speed of operation of a sequential circuit.

CLASSIFICATION OF SEQUENTIAL CIRCUITS:

There exist two main categories of sequential circuits, namely synchronous and asynchronous sequential circuits.

i. Asynchronous Sequential Circuits:

- Sequential circuits whose behavior depends upon the sequence, in which the inputs are applied, are called **Asynchronous Sequential Circuits**.
- In these circuits, outputs are affected whenever a change in inputs is detected. Memory elements used in asynchronous circuits mostly are time delay devices.
- The memory capability of time delay devices is due to the propagation delay of the devices. Propagation delay produced by the logic gates is sufficient for this purpose.
- Hence "An Synchronous sequential circuit can be regarded as a combinational circuit with feedback". However feedback among logic gates makes the asynchronous sequential circuits, often susceptible to instability.
- As a result they may become unstable. This makes the design of asynchronous circuits very tedious and difficult.

ii. Synchronous Sequential Circuit:

- It may be defined as a sequential circuit, whose state can be affected only at the discrete instants of time.
- The synchronization is achieved by using a timing device, termed as **System Clock Generator**, which generates a periodic train of clock pulses.

- The clock pulses are feed to entire system in such a way that internal states (i.e. memory contents) are affected only when the clock pulses hit the circuit.

STORAGE ELEMENTS:

- **Latches**

- A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states.

- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.

- Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches; those controlled by a clock transition are flip-flops. Latches are said to be level sensitive devices; flip-flops are edge sensitive devices.

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

i. SR Latch

- The *SR* latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled *S* for set and *R* for reset.

- Both the versions are shown in Fig 2(a) & Fig 2(b). The latch has two useful states. When output *Q* = 1 and *Q*' = 0, the latch is said to be in the *set state*. When *Q* = 0 and *Q*' = 1, it is in the *reset state*. **SR**

SR Latch with Control Input

- The operation of the basic SR latch can be modified by providing an additional input signal that determines (controls) when the state of the latch can be changed by determining whether S and R (or S' and R') can affect the circuit.

- An SR latch with a control input is shown in Fig 2(c) which consists of the basic SR latch and two additional NAND gates. The control input En acts as an enable signal for the other two inputs.

- **The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0.** This is the quiescent condition for the SR latch. When the enable input goes to 1, information from the S or R input is allowed to affect the latch.

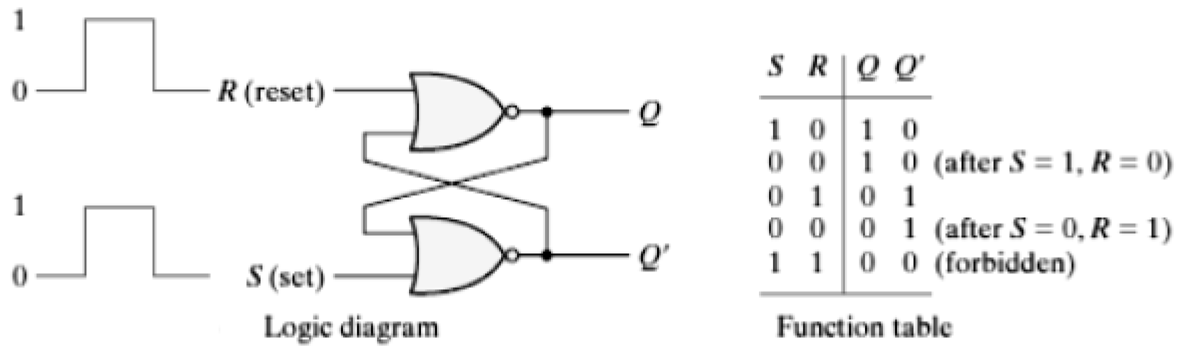


Fig 2.a: SR Latch with NOR gates

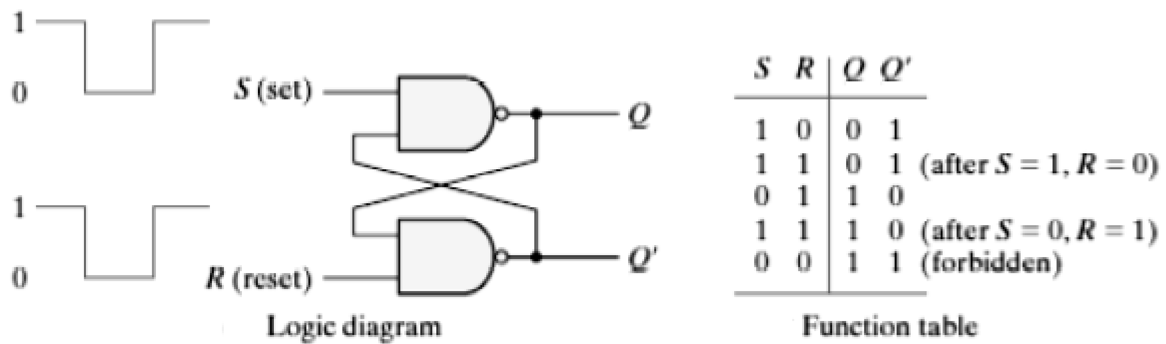


Fig 2.b: SR Latch with NAND gates

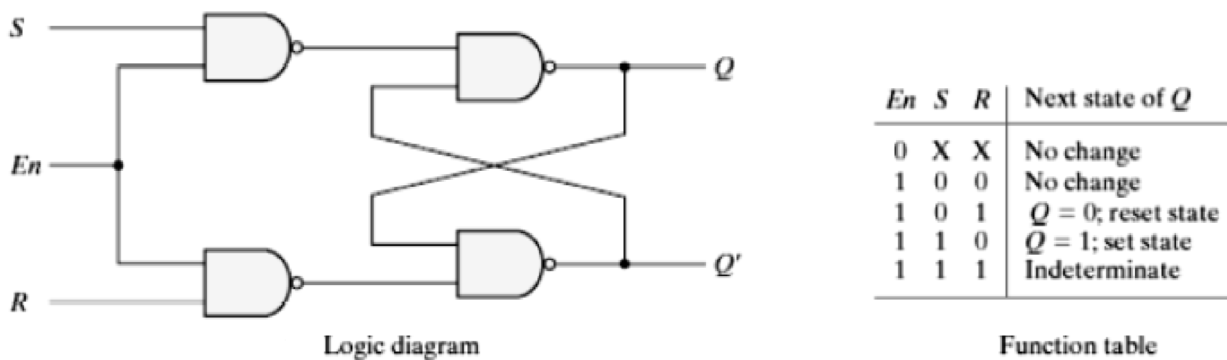


Fig 2.c: SR Latch with Control Input

- The set state is reached with $S = 1$, $R = 0$, and $En = 1$ (active-high enabled). To change to the reset state, the inputs must be $S = 0$, $R = 1$, and $En = 1$. In either case, when En returns to 0, the circuit remains in its current state.

- The control input disables the circuit by applying 0 to En, so that the state of the output does not change regardless of the values of S and R. Moreover, when $En = 1$ and both the S and R inputs are equal to 0, the state of the circuit does not change.

- These conditions are listed in the function table accompanying the diagram. An indeterminate condition occurs when all three inputs are equal to 1. This condition places 0's on both inputs of the basic *SR* latch, which puts it in the undefined state.

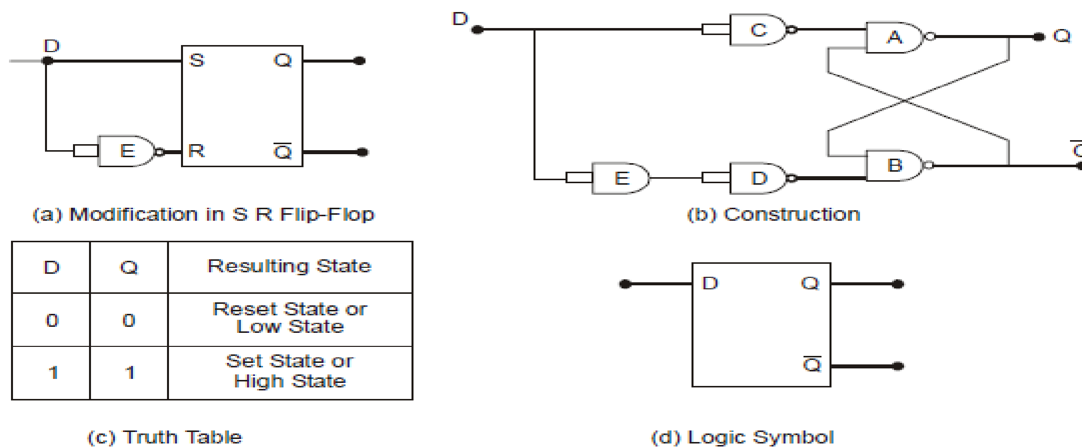
ii. D Latch (Transparent latch)

- One way to eliminate the undesirable condition of the indeterminate state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time.

- This is done in the *D* latch, shown in Fig.3. This latch has only two inputs: *D* (data) and *En* (enable). The *D* input goes directly to the *S* input, and its complement is applied to the *R* input.

- As long as the enable input is at 0, the cross-coupled *SR* latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of *D*. The *D* input is sampled when *En* = 1.

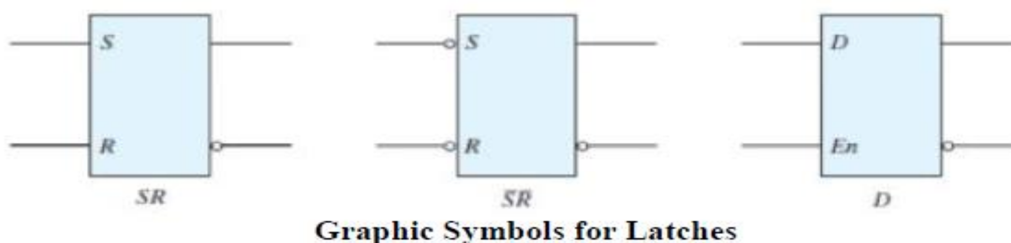
- If *D* = 1, the *Q* output goes to 1, placing the circuit in the set state. If *D* = 0, output *Q* goes to 0, placing the circuit in the reset state.



D	Q	Resulting State
0	0	Reset State or Low State
1	1	Set State or High State

Fig. 3 D-latch

The graphical representation of S-R and D-latch is as shown below



Graphic Symbols for Latches

FLIPFLOPS

- The synchronous sequential circuit which uses clock at the input of memory element is referred as **Clocked Sequential circuit** and the memory element in this circuit known as **Flip-Flop** that can store 1-bit of information, and thus forms a 1-bit memory cell.

- These circuits have two outputs, one giving the value of binary bit stored in it and the other gives the complemented value.

- The real differences among various flip-flops are the number of inputs and the manner in which binary information can be entered into it

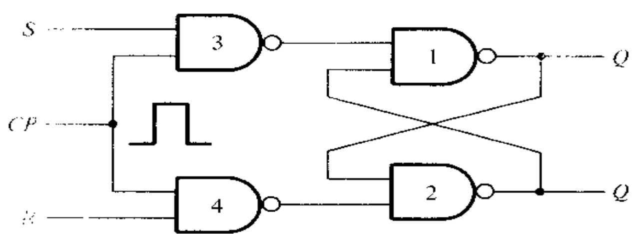
- The flip-flops are 1-bit memory cells that can maintain the stored bit for desired period of time which consists of two stable stages so it is called as **Bi-stable** device and states are 0V and + 5V corresponding to Logic 0 and Logic 1 respectively

i. RS Flip-Flop

- A flip-flop circuit can be constructed either by using two 2-input OR gate or NAND gates. These circuits consists of a cross coupled connection from output of one gate to the input of the other gate constitutes a feedback path. Each flip-flop has two outputs, Q and Q', and two inputs, set, reset.

- The operation of basic flip-flop can be modified by proving an additional control input that determines when the state of the circuit is to be changed.

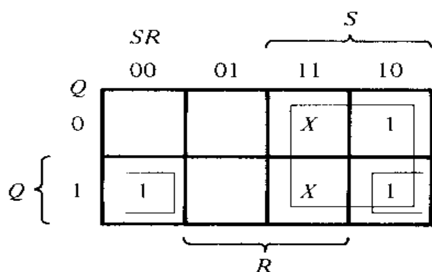
- An RS flip-flop with a clock pulse (CP) input, which consists of a basic flip-flop circuit and two additional NAND gates, is as shown in Fig. 4.



(a) Logic diagram

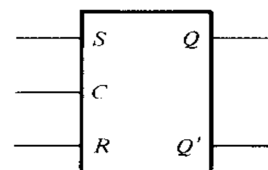
Q	S	R	Q(t + 1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Indeterminate
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Indeterminate

(b) Characteristic table



$$Q(t+1) = S + R'Q$$
$$SR = 0$$

(c) Characteristic equation



(d) Graphic symbol

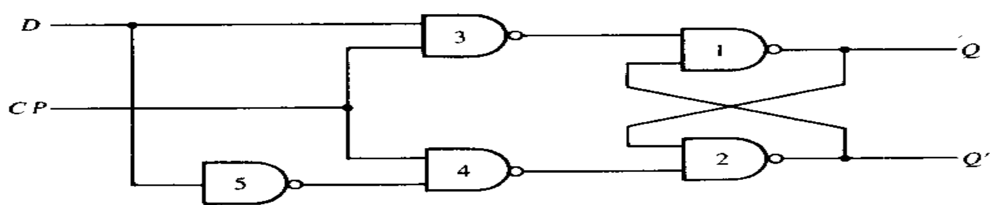
Fig. 4. RS flip-flop with NAND gates

ii. D-Flip-flop

- The SR latch, which has two inputs S and R. At any time to store a bit, must activate both the inputs simultaneously. This may be troubling in some applications. Use of only one data line is convenient in such applications.

- Moreover the forbidden input combination S = R = 1 may occur unintentionally, thus leading the flip-flop to indeterminate state. In order to deal such issues, SR flip-flop is further modified as shown in Fig 5.

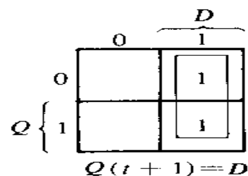
- The resultant is referred as D flip-flop which has only one input labelled D (called as Data input). An external NAND gate (connected as inverter) is used to ensure that S and R inputs are always complement to each other. Thus to store information in this latch, only one signal has to be generated.



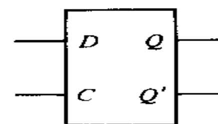
(a) Logic diagram

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

(b) Characteristic table



(c) Characteristic equation



(d) Graphic symbol

Fig 5: D flip-flop or D latch

- Operation of this flip-flop is straight forward. At any instant of time the output Q is same as D (i.e. Q = D). Since output is exactly same as the input, the latch may be viewed as a delay unit.

- The flip-flop always takes some time to produce output, after the input is applied. This is called propagation delay.

- Thus it is said that the information present at point D (i.e. at input) will take a time equal to the propagation delay to reach to Q. Hence the information is delayed. For this reason it is often called as **Delay (D) Flip-Flop**.

iii. JK FLIPFLOP

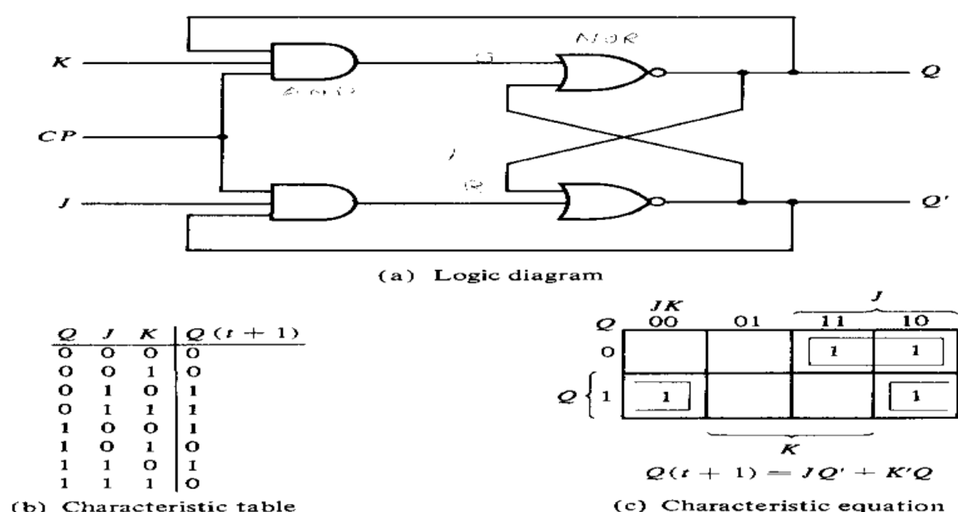
- A JK flip-flop is a refinement of the SR flip-flop in that the indeterminate state of the SR type is defined in the JK type.

- Inputs J and K behave like inputs S and R to set and clear the flip-flop (note that in a JK flip-flop, the letter J is for set and the letter K is for clear).

- When logic 1 inputs are applied to both J and K simultaneously, the flip-flop switches to its complement state, ie., if Q=1, it switches to Q=0 and vice versa. A clocked JK flip-flop is shown in Fig. 6.

- Output Q is ANDed with K and CP inputs so that the flip-flop is cleared during a clock pulse only if Q was previously 1.

- Similarly, ouput Q' is ANDed with J and CP inputs so that the flip-flop is set with a clock pulse only if Q' was previously 1.



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

(b) Characteristic table

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

Fig.6 JK Flip-flop

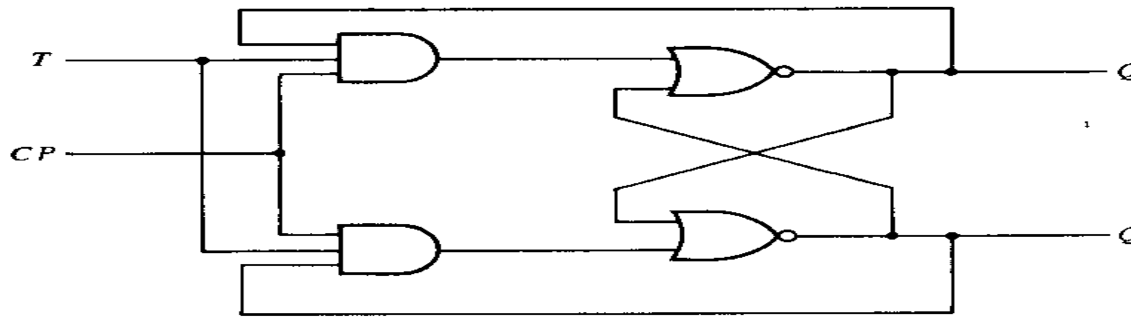
- Note that because of the feedback connection in the JK flip-flop, a CP signal which remains a 1 (while J=K=1) after the outputs have been complemented once will cause repeated and continuous transitions of the outputs.

- To avoid this, the clock pulses must have a time duration less than the propagation delay through the flip-flop.

- The restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction. The same reasoning also applies to the T flip-flop presented next.

iv. T Flip-Flop

- The T flip-flop is a single input version of the JK flip-flop which is shown, in Fig.7and it is obtained from the JK type if both inputs are tied together. The output of the T flip-flop "toggles" with each clock pulse.



(a) Logic diagram

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

(b) Characteristic table

Q \ T	0	1
0		1
1	1	

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

(c) Characteristic equation

Fig. 7 Clocked T flip-flop

Race around Condition and Solution

- Whenever the width of the trigger pulse is greater than the propagation time of the flip-flop, then flip-flop continues to toggle 1-0-1-0 until the pulse turns 0.
- When the pulse turns 0, unpredictable output may result i.e. the state and output not known. This is called race around condition.
- In level-triggered flip-flop circuits, the circuit is always active when the clock signal is high, and consequently unpredictable output may result. For example, during this active clock period, the output of a T-FF may toggle continuously.
- The output at the end of the active period is therefore unpredictable. To overcome this problem, *edge triggered* circuits can be used whose output is determined by the edge, instead of the level, of the clock signal, for example, the rising (or trailing) edge.
- Another way to resolve the problem is the Master-Slave circuit shown in Fig 8.

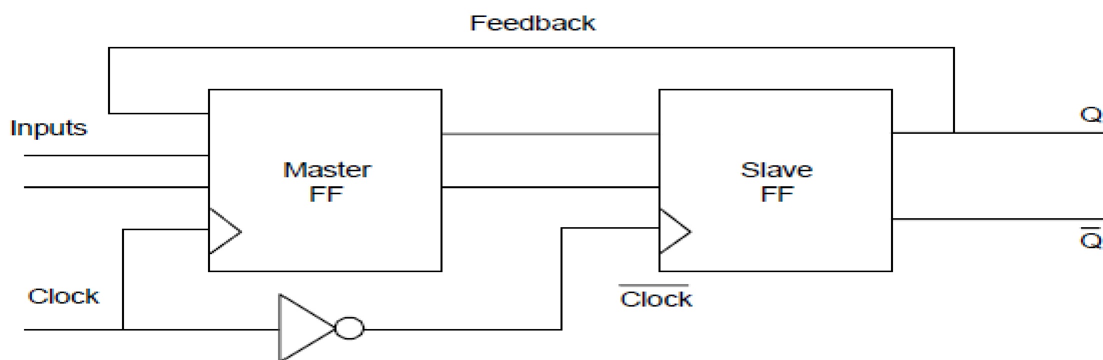


Fig 8: Master slave circuit

The operation of a Master-Slave FF has two phases as shown in Fig.8

- During the high period of the clock, the master FF is active, taking both inputs and feedback from the slave FF. The slave FF is de-activated during this period by the negation of the clock so that the new output of the master FF won't affect it.
- During the low period of the clock, the master FF is deactivated while the slave FF is active. The output of the master FF can now trigger the slave FF to properly set its output. However, this output will not affect the master FF through the feedback as it is not active.

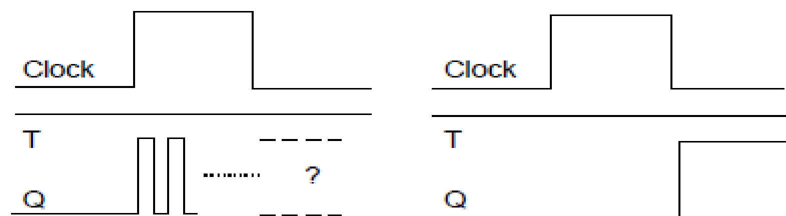


Fig 9: Master slave operation

- It is seen that the trailing edge of the clock signal will trigger the change of the output of the Master-Slave FF. The master-slave combination can be constructed for any type of flip-flop by adding a clocked RS flip-flop with an inverted clock to form the slave. A master-slave JK flip-flop constructed with NAND gates is shown in Fig.10.

- It consists of two flip-flops; gates1 through 4 form the master flip-flop, and gates 5 through 8 form the slave flip-flop. The information presented at the J and K inputs is transmitted to the master flip-flop on the positive edge of the clock pulse and is held there until the negative edge of the clock pulse occurs, after which it is allowed to pass through to the slave flip-flop.

- The clock input is normally 0, which keeps the outputs of gates 1 and 2 at the 1 level. This prevents the J and K inputs from affecting the master flip-flop.

- The slave flip-flop is a clocked RS type, with the master flip-flop supplying the inputs and the clock input being inverted by gate 9. When the clock is 0, the output of gate 9 is 1, so that output Q is equal to Y, and Q' is equal to Y'.

- When the positive edge of a clock pulse occurs, the master flip-flop is affected and may switch states. The slave flip-flop is isolated as long as the clock is at the level1, because the output of gate 9 provides a 1 to both inputs of the NAND basic flip-flop of gates 7 and 8.

- When the clock input returns to 0, the master flip-flop is isolated from J and K inputs and the slave flip-flop goes to the same state as the master flip-flop.

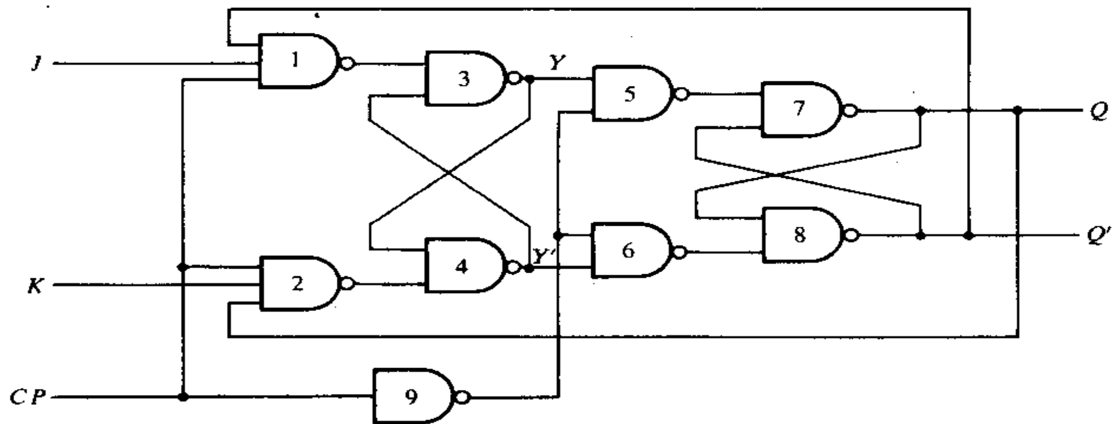


Fig.10 Clocked master-slave JK flip-flop

Operating Characteristics of Flip-flops

The operation characteristics specify the performance, operating requirements, and operating limitations of the circuits. The operation characteristics mentions here apply to all flip-flops regardless of the particular form of the circuit.

Propagation Delay Time—is the interval of time required after an input signal has been applied for the resulting output change to occur.

Set-up Time—is the minimum interval required for the logic levels to be maintained constantly on the inputs (J and K, or S and R, or D) prior to the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip-flop.

Hold Time—is the minimum interval required for the logic levels to remain on the inputs after the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip-flop.

Maximum Clock Frequency—is the highest rate that a flip-flop can be reliably triggered.

Power Dissipation—is the total power consumption of the device.

Pulse Widths—are the minimum pulse widths specified by the manufacturer for the Clock, SET and CLEAR inputs.

Flip-Flop Applications

- Frequency Division
- Parallel Data Storage

FLIP-FLOP EXCITATION TABLE

- The characteristic table is useful during the analysis of sequential circuits when the value of flip-flop inputs is known and if the value of the flip-flop output Q after the rising edge of the clock signal. As with any other truth table, the map method is used to derive the characteristic equation for each flip-flop.

- During the design process the transition from present state to the next state s usually known and flip-flop input conditions are found that will cause the required transition. For this reason a table that lists the required inputs for a given change of state is needed. Such a list is called the *excitation table.*

- There are four possible transitions from present state to the next state. The required input conditions are derived from the information available in the characteristic table.
- The symbol X in the table represents a "don't care" condition, that is, it does not matter whether the input is 1 or 0.
- The different types of flip flops (RS, JK, D, T) can also be described by their excitation, table as shown in Fig. The left side shows the desired transition from Qn to Qn+1, the right side gives the triggering signals of various types of FFs needed for the transitions.

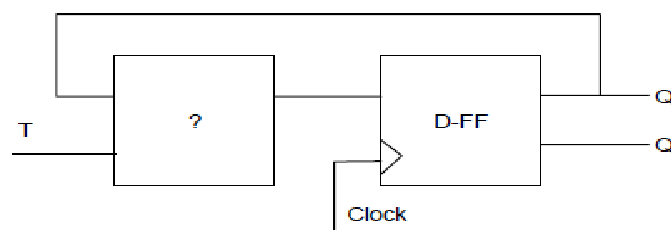
Table 1:Excitation table

Desired transition		*Triggering signal needed*					
Q_n	Q_{n+1}	S	R	J	K	D	T
0	0	0	x	0	x	0	0
0	1	1	0	1	x	1	1
1	0	0	1	x	1	0	1
1	1	x	0	x	0	1	0

FLIP-FLOP CONVERSIONS

- To convert a given type A FF to a desired type B FF some conversion logic is used and the key here is to use the excitation table specified in Table 1 which shows the necessary triggering signal (S, R, J, K, D and T) for a desired flip flop state transition Qn →Qn+1 is reproduced here.

Example 1. *Convert a D-FF to a T-FF:*



A circuit is to be designed which is used to generate the triggering signal D as a function of T and Q : D = f (T, Q)

Consider the excitation table:

Q_n	Q_{n+1}	T	D
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

Treating D as a function of T and current FF state Q Qn we have $D = T'Q + TQ = T \oplus Q$

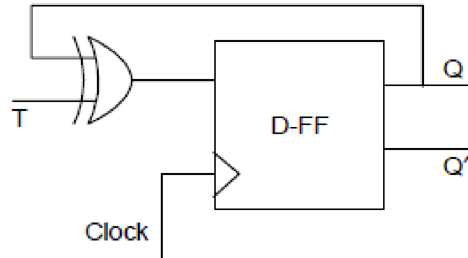
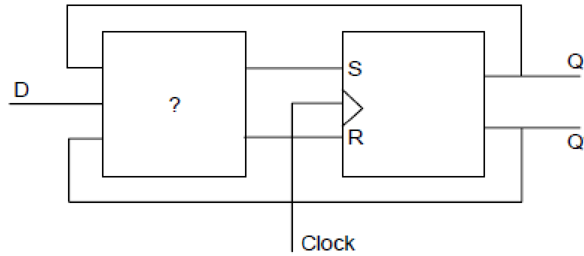


Fig. 11 Convert a D-FF to a T-FF

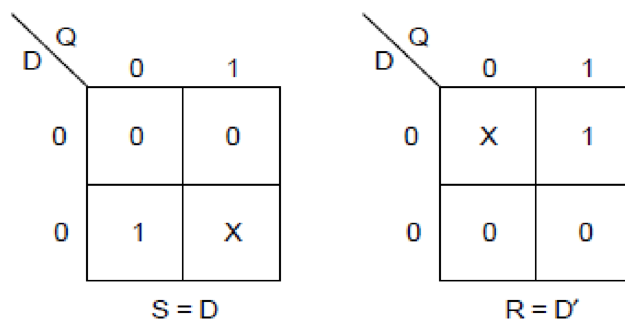
Example 2. *Convert a RS-FF to a D-FF:*



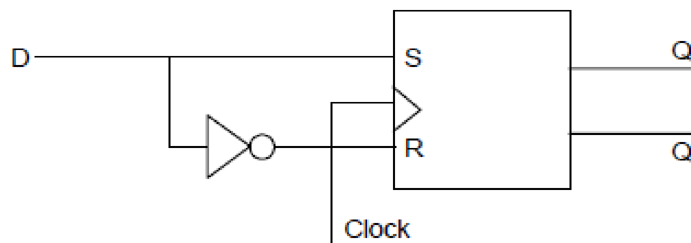
A circuit is to be designed which can generate the triggering signals S and R as functions of D and Q. Consider the excitation table:

Q_n	Q_{n+1}	D	S	R
0	0	0	0	X
0	1	1	1	0
1	0	0	0	1
1	1	1	X	0

The desired signal S and R can be obtained as functions of T and current FF state Q from the Karnaugh maps:

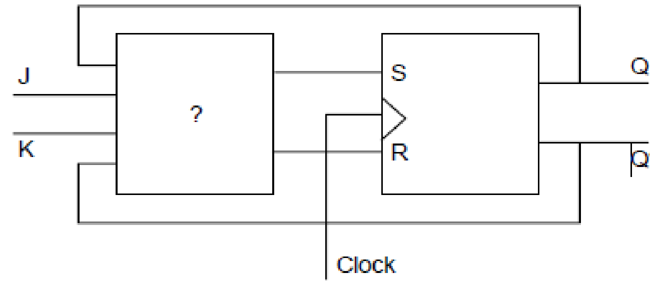


$$S = D, R = D'$$



Fg.12 Convert a RS-FF to a D-FF

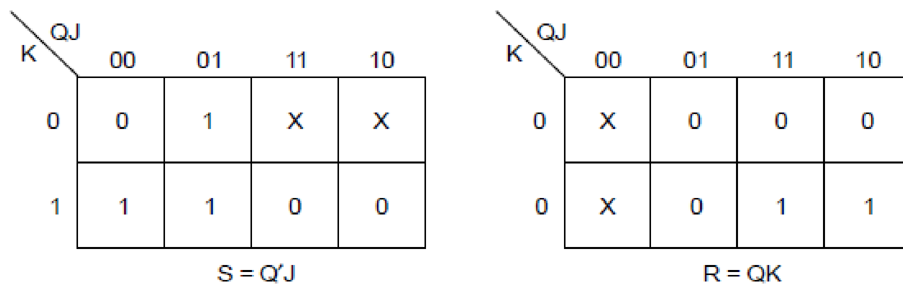
Example 3. *Convert a RS-FF to a JK-FF.*



We need to design the circuit to generate the triggering signals S and R as functions of J, K and Q. Consider the excitation table.

Q_n	Q_{n+1}	J	K	S	R
0	0	0	x	0	x
0	1	1	x	1	0
1	0	x	1	0	1
1	1	x	0	x	0

The desired signals S and R as function J, K and current FF state Q can be obtained from the Karnaugh maps:



S = Q'J, R = QK

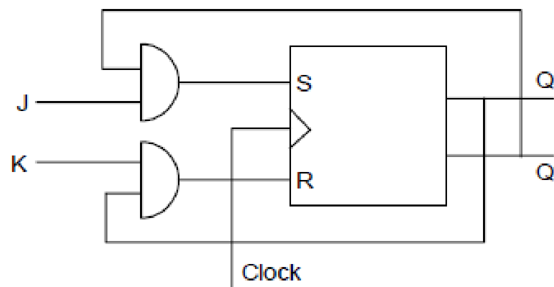


Fig.13 Convert a RS-FF to a JK-FF

RIPPLE COUNTER

- A register that goes through a prescribed sequence of states upon the application of input pulses is called a counter.
- The input pulses may be clock pulses, or they may originate from some external source and may occur at a fixed interval of time or at random.
- The sequence of states may follow the binary number sequence or any other sequence of states.

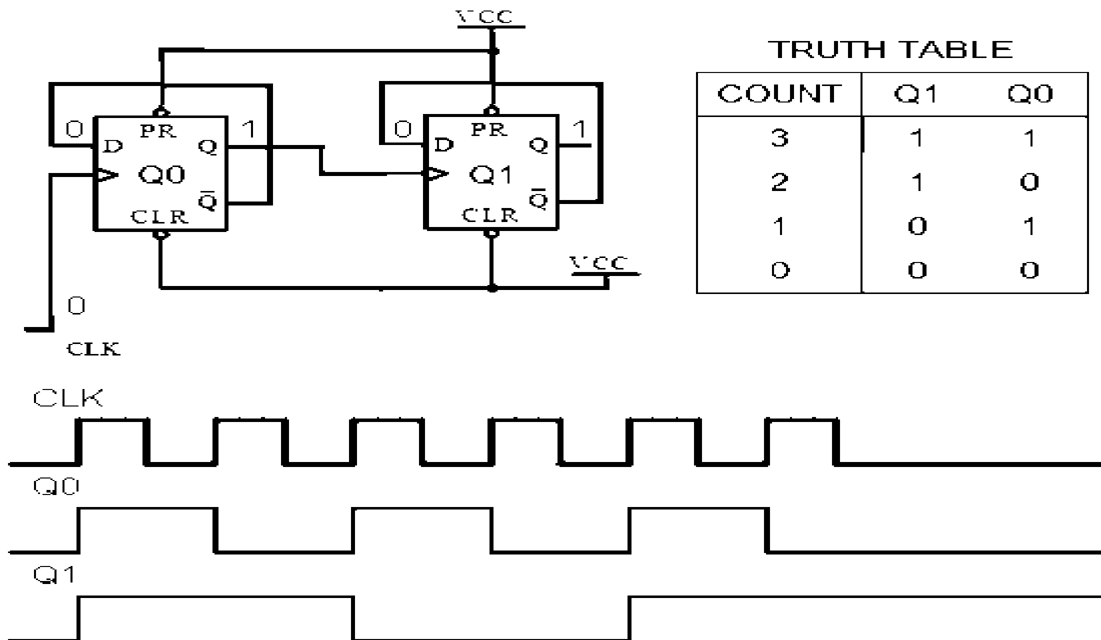
- A counter that follows the binary number sequence is called a binary counter. An n -bit binary counter consists of n flip-flops and can count in binary from 0 through 2n - 1.
- Counters are available in two categories: ripple counters and synchronous counters.
- In a ripple counter, a flip-flop output transition serves as a source for triggering other flip-flops. In other words, the C (clock)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 ripple counter is an asynchronous counter where only the first flip-flop is clocked by an external clock. All subsequent flip-flops are clocked by the output of the preceding flip-flop.
- Asynchronous counters are also called ripple-counters because of the way the clock pulse ripples it way through the flip-flops.
- The MOD of the ripple counter or asynchronous counter is 2^n if n flip-flops are used. For a 4-bit counter, the range of the count is 0000 to 1111.
- A counter may count up or count down or count up and down depending on the input control. The count sequence usually repeats itself. When counting up, the count sequence goes from 0000, 0001, 0010, ... 1110 , 1111 , 0000, 0001, ... etc.
- When counting down the count sequence goes in the opposite manner: 1111, 1110, ... 0010, 0001, 0000, 1111, 1110, ... etc.
- The complement of the count sequence counts in reverse direction. If the uncomplemented output counts up, the complemented output counts down. If the uncomplemented output counts down, the complemented output counts up.
- There are many ways to implement the ripple counter depending on the characteristics of the flip flops used and the requirements of the count sequence.
    - Clock Trigger: Positive edged or Negative edged
    - JK or D flip-flops
    - Count Direction: Up, Down, or Up/Down
- Asynchronous counters are slower than synchronous counters because of the delay in the transmission of the pulses from flip-flop to flip-flop.

- With a synchronous circuit, all the bits in the count change synchronously with the assertion of the clock. Examples of synchronous counters are the Ring and Johnson counter.

- It can be implemented using D-type flip-flops or JK-type flip-flops. The circuit below uses 2 D flip-flops to implement a divide-by-4 ripple counter ($2^n = 2^2 = 4$). It counts down.



COUNT	Q1	Q0
3	1	1
2	1	0
1	0	1
0	0	0

Fig. 13 Two bit Ripple Conter

- Click on CLK (Red) switch and observe the changes in the outputs of the flip flops. The CLK switch is a momentary switch (similar to a door bell switch - normally off).
- PR and CLR are both connected to VCC (set to 1)
- The D flip flop clock has a rising edge CLK input. For example Q0 behaves as follows
    - The D input value just before the CLK rising edge is noted (Q00).
    - When CLK rising edge occurs, Q0 is assigned the previously noted D value (Q00).
    - Thus, whenever a rising edge appears at the CLK of the D flip flop, the output Q changes state (or toggles).
- The MOD or number of unique states of this 2 flip flop ripple counter is 4 (22).
- Simulate and Breadboard the Ripple Counter circuit.
- A Truncated Ripple Counter is used if a MOD of less than 2n is required. For example, if you want to change the sequence from 3,2,1,0,3,2,1,0 ... to 3,2,0,3,2,0 ...

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 of Fig14. 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).

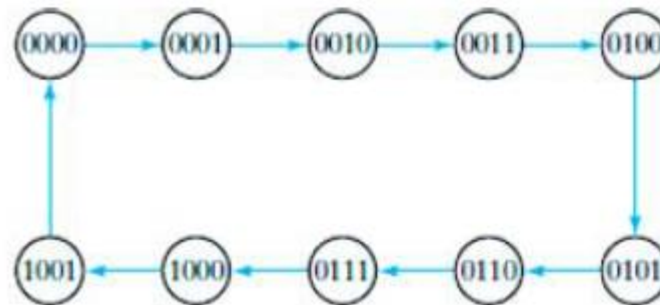


Fig.14 State Diagram of BCD counter

- The logic diagram of a BCD ripple counter using *JK* flip-flops is shown in Fig.15, 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 $J = K = 0$.

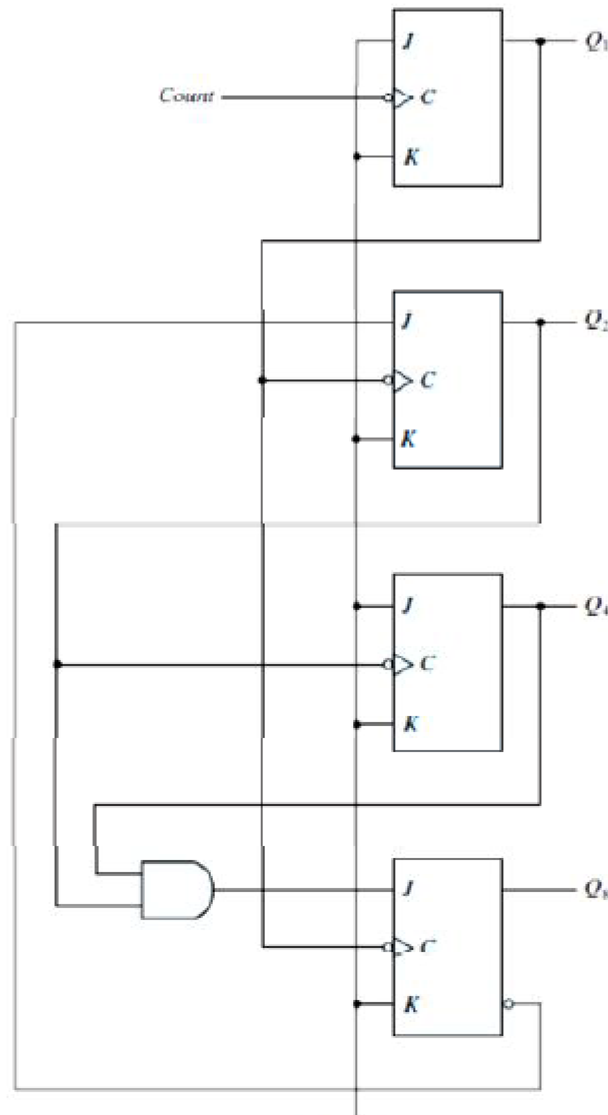


Fig.15 BCD counter

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.

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 A3A2A1A0 = 0011, the next count is 0100. A0 is always complemented.

- A1 is complemented because the present state of A0 = 1. A2 is complemented because the present state of A1A0 = 11. However, A3 is not complemented, because the present state of A2A1A0 = 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 four-bit counter depicted in Fig. 16 below.

- The C inputs of all flip-flops are connected to a common clock. The counter is enabled by Count Enable. If the enable input is 0, all J and K inputs are equal to 0 and the clock does not change the state of the counter.

- The first stage, A0, has its J and K equal to 1 if the counter is enabled. The other J and K inputs are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled.

- The chain of AND gates generates the required logic for the J and K inputs in each stage. The counter can be extended to any number of stages, with each stage having an additional flip-flop and an AND gate that gives an output of 1 if all previous flip-flop outputs are 1.

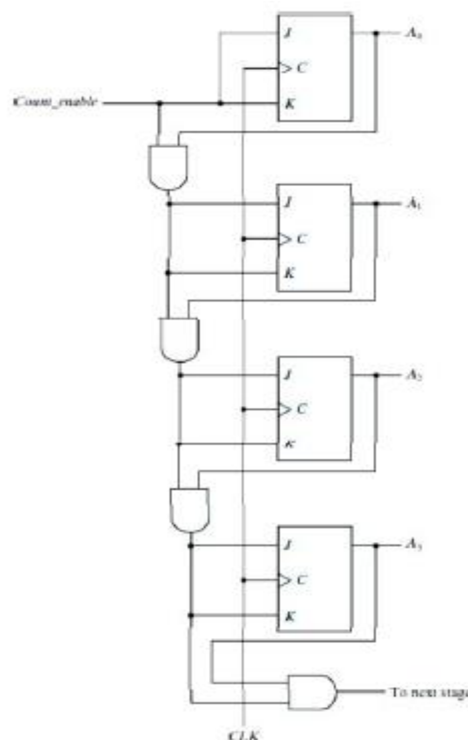


Fig.16 Four Bit Synchrounous Binary Counter

Binary Up/Down 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 Fig.17 below, 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 an up–down binary counter using T flip-flops is shown in Fig.17 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 in the same count.
- When the up and down inputs are both 1, the circuit counts up. This set of conditions ensures that only one operation is performed at any given time. Note that the up input has priority over the down input.

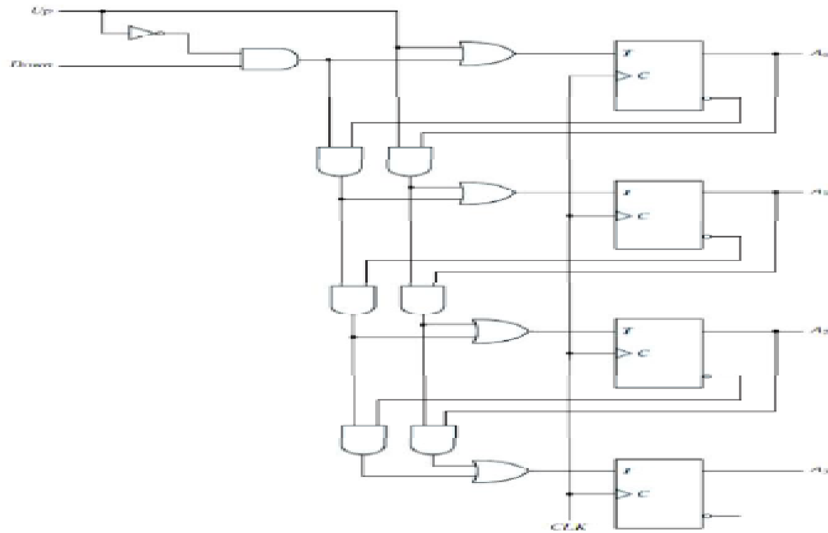


Fig 17: Four-bit up-down binary counter

Ring Counter

- A ring counter is a Shift Register (a cascade connection of flip-flops) with the output of the last flip flop connected to the input of the first. It is initialized such that only one of the flip flop output is 1 while the remainder is 0.

- The 1 bit is circulated so the state repeats every n clock cycles if n flip-flops are used. The "MOD" or "MODULUS" of a counter is the number of unique states. The MOD of the n flip flop ring counter is n. It can be implemented using D-type flip-flops (or JK-type flip-flops).

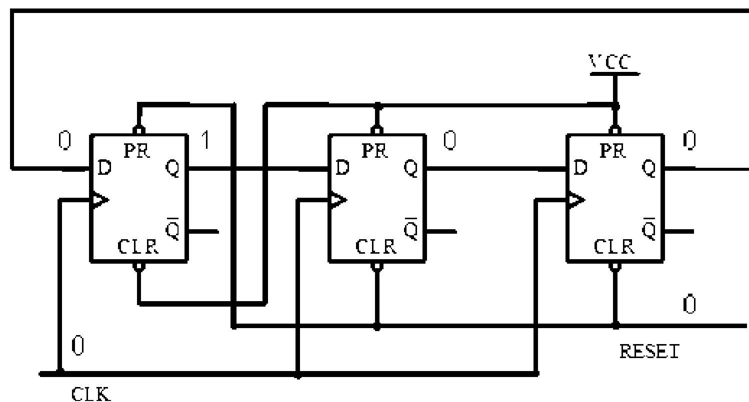


Fig 18: Ring Counter

Notes:

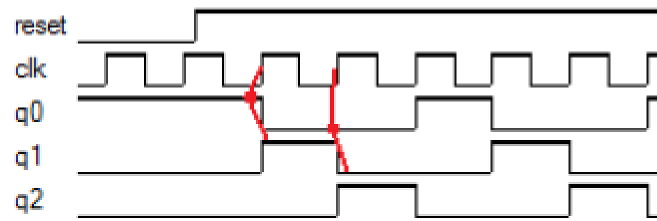
· Enable the flips flops by clicking on the RESET (Green) switch. The RESET switch is a on/off switch (similar to a room light switch)

· Click on CLK (Red) switch and observe the changes in the outputs of the flip flops. The CLK switch is a momentary switch (similar to a door bell switch - normally off).

· The D flip flop clock has a rising edge CLK input. For example Q1 behaves as follows:

-The D input value just before the CLK rising edge is noted (Q0).

-When CLK rising edge occurs, Q1 is assigned the previously noted D value (Q0).



The MOD or number of unique states of this 3 flip flop ring counter is 3.

Truth Table

State	Q0	Q1	Q2
0	1	0	0
1	0	1	0
2	0	0	1

Johnson Counter

- A Johnson counter is a modified ring counter, where the inverted output from the last flip flop is connected to the input to the first.
- The register cycles through a sequence of bit-patterns. The MOD of the Johnson counter is 2n if n flip-flops are used.
- The main advantage of the Johnson counter is that it only needs half the number of flip-flops compared to the standard ring counter for the same MOD.
- It can be implemented using D-type flip-flops (or JK-type flip-flops).

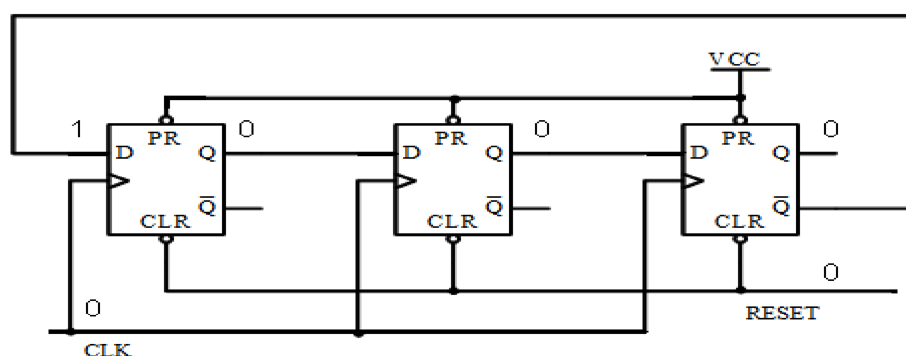
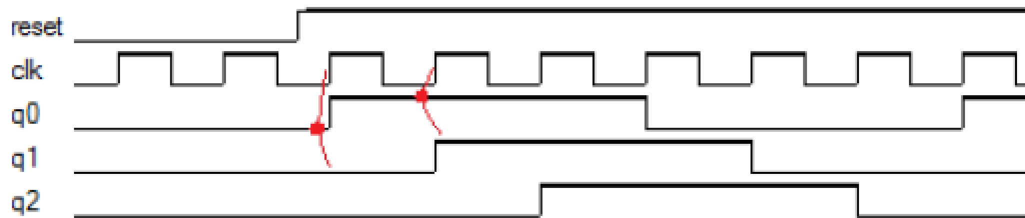


Fig 19: Johnson Counter

Notes:

- Enable the flips flops by clicking on the RESET (Green) switch. The RESET switch is a on/off switch (similar to a room light switch)
- Click on CLK (Red) switch and observe the changes in the outputs of the flip flops. The
- CLK switch is a momentary switch (similar to a door bell switch - normally off).
    - The D flip flop clock has a rising edge CLK input. For example Q1 behaves as follows:

- The D input value just before the CLK rising edge is noted (Q0).
- When CLK rising edge occurs, Q1 is assigned the previously noted D value (Q0).



The MOD or number of unique states of this 3 flip flop Johnson counter is 6.

Truth Table

State	Q0	Q1	Q2
0	0	0	0
1	1	0	0
2	1	1	0
3	1	1	1
4	0	1	1
5	0	0	1

REGISTER:

- A clocked sequential circuit consists of a group of flip-flops and combinational gates. The flip-flops are essential because, in their absence, the circuit reduces to a purely combinational circuit (provided that there is no feedback among the gates).

- A circuit with flip-flops is considered a sequential circuit even in the absence of combinational gates. Circuits that include flip-flops are usually classified by the function they perform rather than by the name of the sequential circuit. Two such circuits are registers and counters.

- A register is a group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information. An n -bit register consists of a group of n flip-flops capable of storing n bits of binary information.

- In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops together with gates that affect their operation.

- The flip-flops hold the binary information, and the gates determine how the information is transferred into the register.

- A counter is essentially a register that goes through a predetermined sequence of binary states. The gates in the counter are connected in such a way as to produce the prescribed sequence of states.

- Although counters are a special type of register, it is common to differentiate them by giving them a different name.

- Various types of registers are available commercially. The simplest register is one that consists of only flip-flops, without any gates.

- A register constructed with four D -type flip-flops to form a four-bit data storage register is shown in figure below.

- The common clock input triggers all flip-flops on the positive edge of each pulse, and the binary data available at the four inputs are transferred into the register.

- The value of (I 3 , I 2 , I 1 , I 0) immediately before the clock edge determines the value of (A 3 , A 2 , A 1 , A 0) after the clock edge.

- The four outputs can be sampled at any time to obtain the binary information stored in the register.

- The input Clear_b goes to the active-low R (reset) input of all four flip-flops. When this input goes to 0, all flip-flops are reset asynchronously.

- The Clear_b input is useful for clearing the register to all 0's prior to its clocked operation. The R inputs must be maintained at logic 1 (i.e., de-asserted) during normal clocked operation.

- Note that, depending on the flip-flop, either Clear, Clear_b, reset, or reset_b can be used to indicate the transfer of the register to an all 0's state.

SHIFT REGISTERS:

- A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a *shift register.*

- The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop.

- All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next. The simplest possible shift register is one that uses only flip-flops, as shown in Fig.20

- The output of a given flip-flop is connected to the D input of the flip-flop at its right. This shift register is unidirectional (left-to-right).

- Each clock pulse shifts the contents of the register one bit position to the right. The configuration does not support a left shift.
- The serial input determines what goes into the leftmost flip-flop during the shift. The serial output is taken from the output of the rightmost flip-flop.

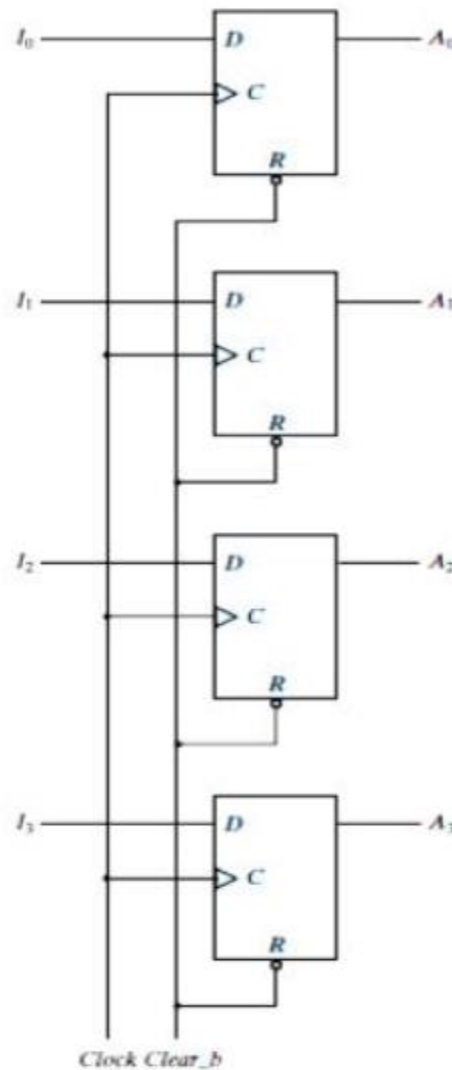


Fig 20: 4 –Bit Register

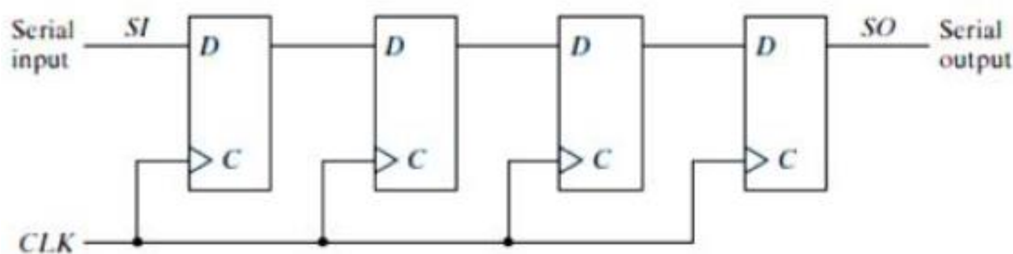


Fig 21: **Four bit Shift register**

- Sometimes it is necessary to control the shift so that it occurs only with certain pulses, but not with others. As with the data register discussed in the previous section, the clock's signal can be suppressed by gating the clock signal to prevent the register from shifting.

- A preferred alternative in high speed circuits is to suppress the clock action, rather than gate the clock signal, by leaving the clock path unchanged, but recirculating the output of each register cell back through a two-channel mux whose output is connected to the input of the cell.

- When the clock action is not suppressed, the other channel of the mux provides a data path to the cell.

- Shift registers have found considerable application in arithmatic operations. Since, moving a binary number one bit to the left is equivalent to multiplying the number by 2 and moving the number one bit position to the right amounts to dividing the number by 2.

- Thus, multiplications and divisions can be accomplished by shifting data bits. Shift registers find considerable application in generating a sequence of control pulses.

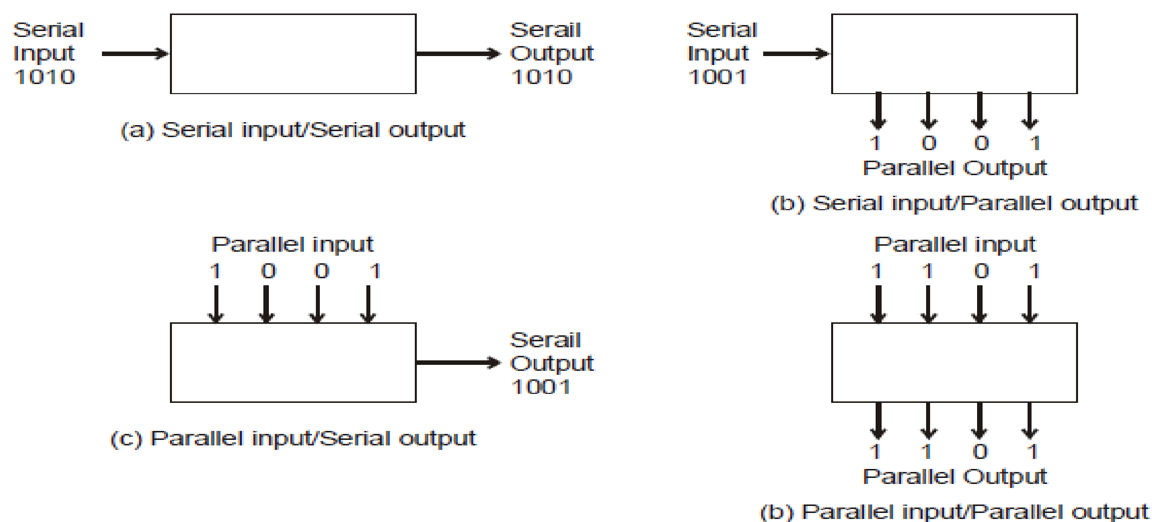


Fig 22: Data Transmission in Shift Register

Bidirectional Shift Registers

The registers discussed so far involved only right shift operations. Each right shift operation has the effect of successively dividing the binary number by two.

- If the operation is reversed (left shift), this has the effect of multiplying the number by two. With suitable gating arrangement a serial shift register can perform both operations.

A bi-directional, or reversible shift register is one in which the data can be shift either left or right. A four-bit bi-directional shift register using D-flip-flops is shown in Fig 23.

Here a set of NAND gates are configured as OR gates to select data inputs from the right or left adjacent bistables, as selected by the $\overline{\text{LEFT}}$_/RIGHT control line.

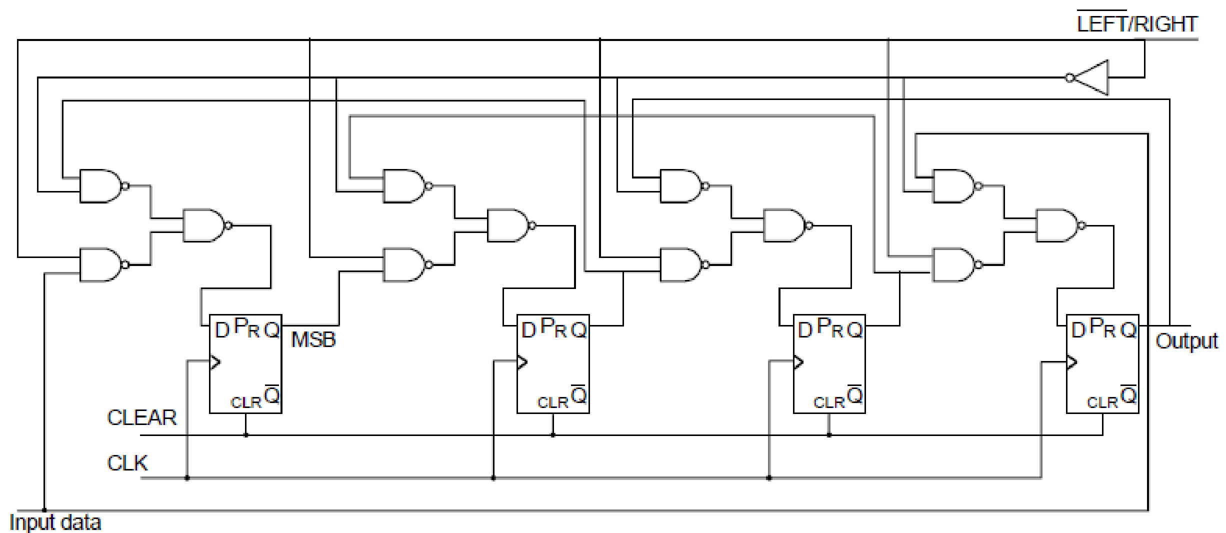


Fig 23: 4 Bit Bidirectional Shift Register

Universal Shift Register:

A Universal Shift register can shift the data directional along with the parallel load operation. The following are the functions done by a Universal Shift register.

- A clear control to clear the register to 0.
- A CP input for clock pulse to synchronize all operations
- A shift-right control to enable the shift-right operation and the serial input and output lines associated with the shift right.
- A shift-left control to enable the shift-left operation and the serial input and output lines associated with the shift left.
- A parallel-load control to enable a parallel transfer and n input lines associated with the parallel transfer.
- N parallel output lines.
- A control state that leaves the information in the register unchanged even though clock pulses are continuously applied.

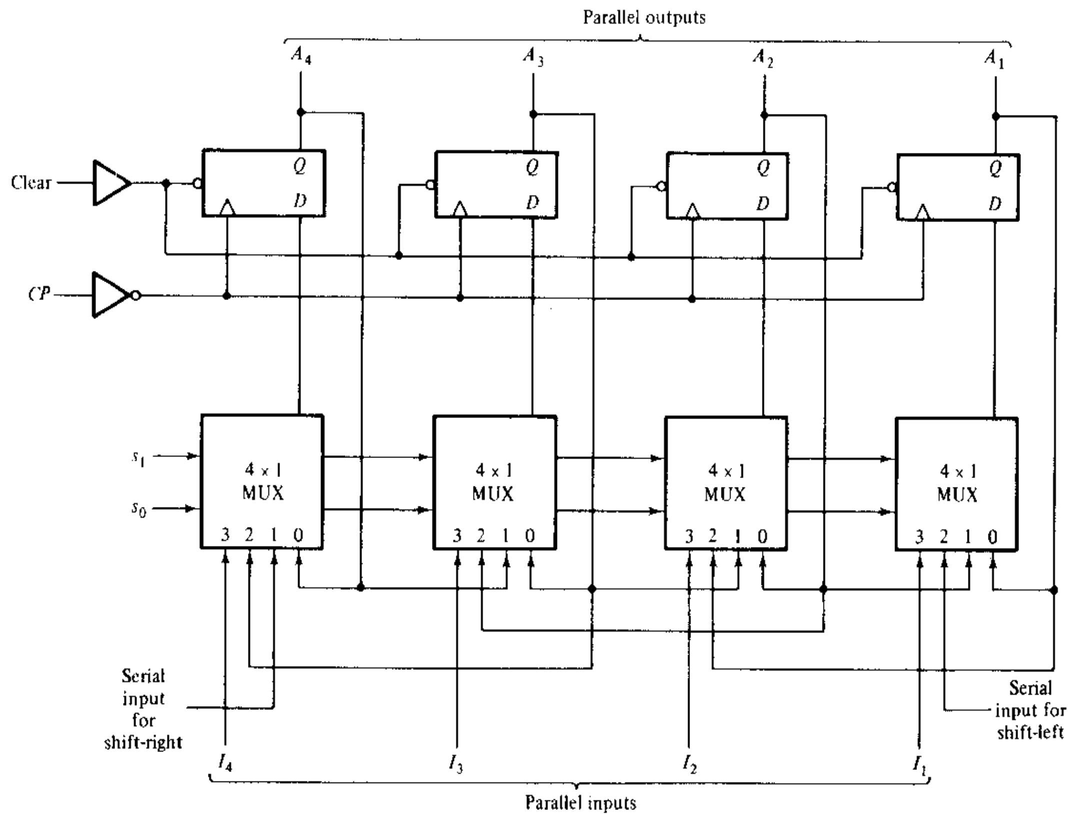


Fig 24: 4- Bit Universal Shift Register

- The Universal Shift Register that is shown in Fig. 24 has all the capabilities that are listed above. It consists f four D-flip-flops and four multiplexers which has two selection lines. The S_1 and S_0 inputs control the mode of operation of the register which is specified in Table.1
-
- When S_1 S_0=00, the present value of the register is applied to the D-inputs of the flip-flops which forms a path from output of each flip-flop into the input of the same flip-flop. So no change of state occurs.
- When S_1 S_0=01, terminal 1 of the multiplexer inputs have a path to the D inputs of the flip-flops which causes a shift-right operation.
- When S_1 S_0=10, a shift-left operation results, with the other serial input going into flip-flop A_1.
- Finally, when S_1 S_0=11, the binary information on the parallel input lines is transferred into the register simultaneously during next clock pulse.

Table.1: Functional Table for Universal Shift Register

Mode Control		**Register Operation**
S_1	**S_0**	

0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel Load

APPLICATIONS OF SHIFT REGISTERS

Shift registers can be found in many applications. Here is a list of a few.

- To Produce Time Delay
- To Simplify Combinational Logic
- To Convert Serial Data to Parallel Data

Assignment-Cum-Tutorial Questions

Section-A

1. A sequential logic circuit

   A) Must contain flip-flops B) may contain flip-flops

   C) does not contain flip-flops D) contain latches

2. A sequential circuit does not use clock pulses. It is

   A) an asynchronous sequential circuit

   B) a synchronous sequential circuit

   C) a counter D) a shift register

3. A flip-flop can store

   A) one bit of data B) two bits of data

   C) tree bits of data D) any number of bits of data

4. The characteristic equation of a J-K flip-flop is_____.
5. The characteristic equation of a D flip-flop is_____.
6. The transparent flip-flop is

   A) an S-R flip-flop B) a D flip-flop

   C) a T flip-flop D) a J-K flip flop

7. A universal register

   A) accepts serial input B) accepts parallel input

   C) gives serial and parallel D) is cable of all of the above

8. The output Q_n of a J-K flip-flop is 1. It changes to 0 when a clock pulse is applied. Then the inputs J_n and K_n are respectively

   A) 0 and X B) 1 and X C) X and 1 D) 0 and X

9. The output Q_n of a S-R flip-flop is 0.It changes to 1 when a clock pulse is applied. Then the inputs S_n and R_n are respectively

   A) X and 1 B) 0 and 1 C) X and 0 D) X and 1

10. A 4-bit binary ripple counter uses flip-flops with propagation delay time of 25ns each. The maximum possible time required for change of state will be

   A) 25ns B) 50ns C) 75ns D) 100ns

11. A mod-2 counter followed by a mod-5 counter is

   A) the same as a mod-5 counter followed by a mod-2 counter

   B) a decade counter C) a mod-7 counter D) none of above

12. A sequential circuit with ten states will have

   A) 10 flip-flops B) 5 flop-flops C) 4 flip-flops D) 0 flip-flops

13. The minimum number of flop-flops required for a mod-12 ripple counter is

   A) 3 B) 4 C) 6 D) 12

Section-B

1. Distinguish between combinational and sequential circuits.
2. Find the characteristic equation for JK flip-flop.
3. Convert a J-K flip-flop into T flip-flop
4. Convert an SR flip-flop into JK flip-flop
5. What is a universal shift register and explain it's working.
6. Write the Excitation tables of D, T, SR, JK Flip Flops
7. What are shift register counters? Draw Ring Counter and explain the operation with Truth Table.
8. Design a 3 bit synchronous up/down counter using JK flip-flop.
9. Design a mod 7 asynchronous counter using JK flip-flop.
10. Design a mod 12 synchronous counter using T flip-flop.

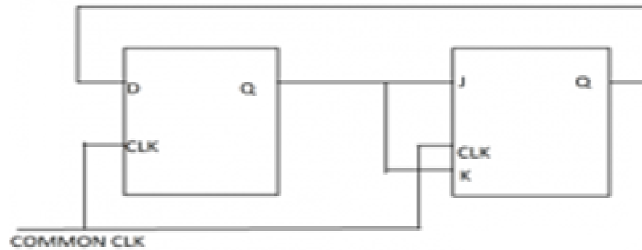
Section-C

1. A synchronous counter counts the sequence 0-1-0-2-0-3 and then repeats. The minimum number of J-K flip-flops required to implement this counter is **GATE-2016**

   A) 1 B) 2 C) 4 D) 5

2. A positive edge-triggered D flip-flop is connected to a positive edge-triggered JK flip flop as follows. The Q output of the D flip-flop is connected to both the J and K inputs of the JK flip-flop, while the Q output of the JK flip-flop is connected to the input of the D flip-flop. Initially, the output of the D flip-flop is set to logic one and the output of the JK flip-flop is cleared. Which one of the following is the bit sequence (including the initial state) generated at the

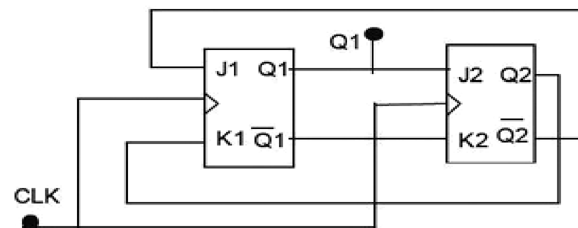
Q output of the JK flip-flop when the flip-flops are connected to a free-running common clock? Assume that J = K = 1 is the toggle mode and J = K = 0 is the state-holding mode of the JK flip-flop. Both the flip-flops have non-zero propagation delays. **GATE-2015**



A) 0110110... B) 0100100...

C) 011101110... D) 011001100...

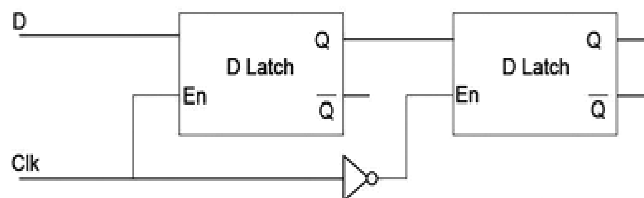
3. The outputs of the two flip-flops Q_1, Q_2 in the figure shown are initialized to 0, 0. The sequence generated at Q_1 upon application of clock signal is **GATE-2014**



A) 01110... B)01010...

C) 00110... D) 01100...

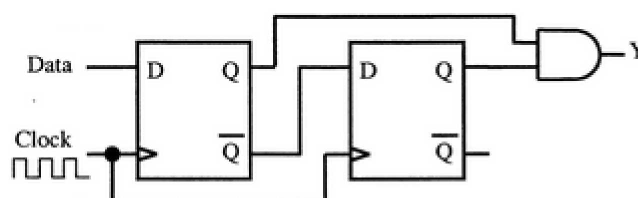
4. The circuit shown in the figure is a **GATE-2014**



A) Toggle flip-flop B) JK flip-flop

C) SR flip-flop D) Master-Slave D flip-flop

5. When the output Y in the circuit below is '1'. It implies that data has **GATE-2011**

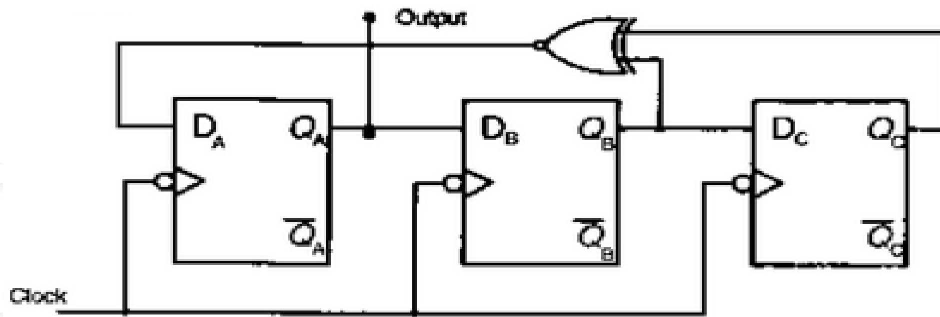


A) Changed from '0' to '1'

B) Changed from '1' to '0'

C) Changed in either direction D) Not changed

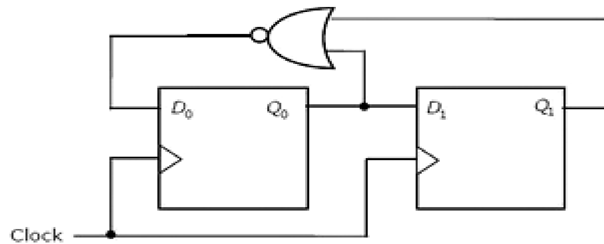
6. Assuming that the all flip-flops are in reset condition initially, the count sequence observed at Q_A in the circuit shown is



A) 0010111.... B) 0001011.... **GATE-2010**

C) 0101111.... D) 0110100....

7. For the circuit shown, the counter state (Q_1Q_0) follows the sequences



A) 00,01,10,11,00,...... **GATE-2007**

B) 00,01,10,00,01,......

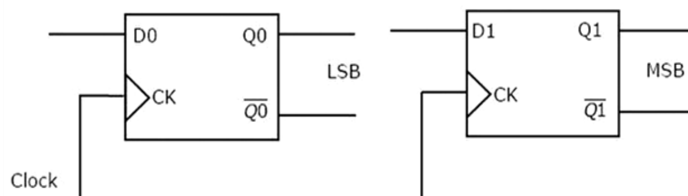
C) 00,01,11,00,01,......

D) 00,10,11,00,10,......

8. Two D flip-flops are to be connected as a synchronous counter as shown below, that goes through the following Q_1 Q_0 sequence

00→01→11→10→00→..............

The inputs D_0 and D_1 respectively should be connected as **GATE-2006**



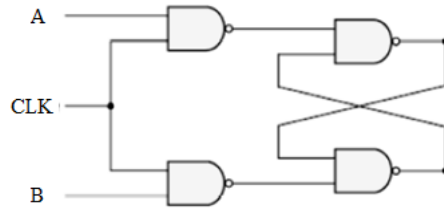
A) $\overline{Q_1}$ and Q_0 B) $\overline{Q_0}$ and Q_1

C) $\overline{Q_1}$ $Q_0$and Q_0 D) $\overline{Q_1}$ $Q_0$and Q_1 Q_0

9. The present output Q_n of an edge triggered JK-Flip Flop is logic '0'. If J=1,then Q_{n+1} is **GATE-2005**

A) Can’t determined
B) Will be logic ‘1’
C) Will be logic ‘0’
D) Will race around

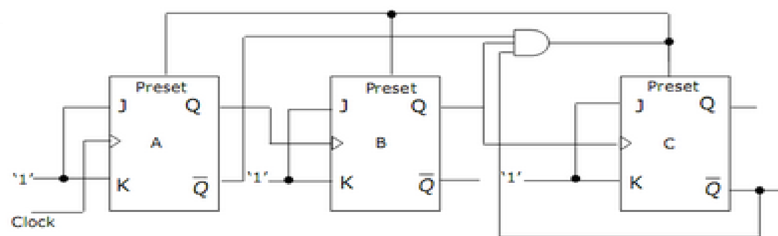
10. Consider the given circuit. In the circuit race around condition will



A) Does not occur **GATE-2005**
B) Occur when CLK=0
C) Occur when CLK=1 and A=B=1
D) Occur when CLK=1 and A=B=0

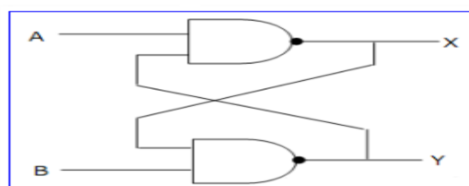
11. A Master-Slave flip-flop has the characteristic that **GATE-2004**
A) Change in the input immediately reflected in the output.
B) Change in the output occurs when the state of the master is affected.
C) Change in the output occurs when the state of the slave is affected.
D) Both the master and slave states are affected at the same time.

12. The ripple counter shown in the given figure is works as a



A) Mod-3 up counter
B) Mod-5 up counter
C) Mod-3 down counter
D) Mod-5 down counter

13. In the figure shown is A=1 and B=1, the input B is now replaced with a sequence 101010...., the output X and Y will be **IES-2005**



A) Fixed at 0 and 1 respectively B) X=1010…. While Y=0101…

C) X=1010…. and Y=1010…. D) Fixed at 1 and 0 respectively

14. A Master Slave flip flop has the characteristic that **IES-2001**

A) Change in input immediately reflected in the output

B) Change in the output occurs when the state of the Master is affected

C) Change in the output occurs when the state of the Slaver is affected

D) Both the master and the slave states are affected at the same time

UNIT – IV

Finite State Machines

Objectives:

- To familiarize with the concepts of Finite state machines.

Syllabus:

Types of FSM, Capabilities and limitations of FSM, State assignment, Realization of FSM using flip-flops, Mealy to Moore conversion and vice-versa, Reduction of state tables using partition technique

Outcomes:

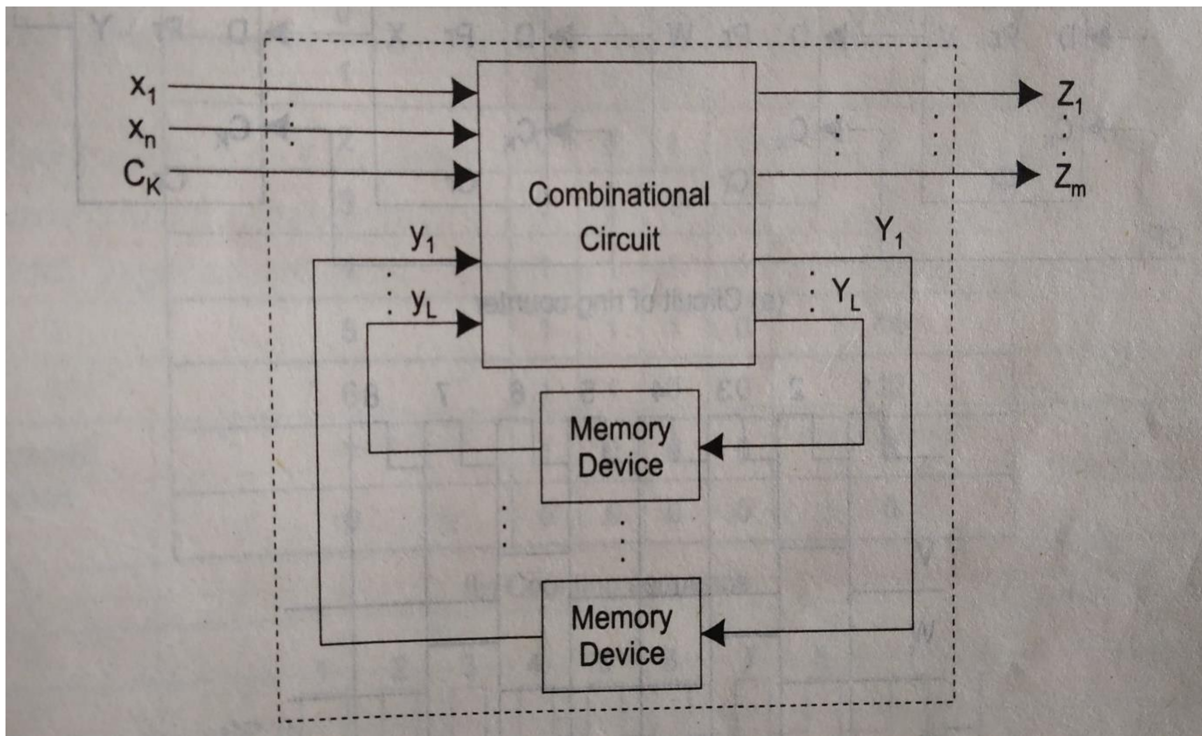
Students will be able to

- design FSM charts using flip flops.
- understand the melay machines and moore machines.
- Reduction methods of state tables.
- partition technique.

Model Of A Finite State Machine (FSM)

It is a Finite State Machine (FSM) also called Finite Automation in the literature pertaining to automata theory. FSM comprises an input set (I), output set (Z),a set of states (S), state transition function (δ), and output function (λ). Thus, the finite state machine M is a quintuple given by M = (I,Z,S, δ, λ),where δ is a function of present state resulting in the next state and λ is a function which enables us to compute the output depending on the present inputs and present state.The previous statement refers to what is generally called the Mealy Machines.

The clock pulses control all timing in the machine . If the clock is removed, the model represents an asynchronous sequential machine with mere delays replacing flip-flops.



Limitations Of Finite State Machines

- No finite state machine can be produced for an infinite sequence.
- No finite state machine can multiply two arbitrary large binary numbers.

No finite state machine can be designed to produce such a non-periodic infinite sequence for a periodic input.

Mealy And Moore Models

A sequential machine M is a quintuple comprising a set of inputs I,a set of outputs Z, a set of states S, a transition function δ which enables finding the next state depending on the present state and present input and finally an output function λ.This is symbolically expressed as M=(I,Z,S, δ, λ).

If the output function depends on the present state and present inputs,it is called the Mealy model,named after G.H.Mealy, a pioneer in the field. If the output is associated only with the present state,it is called the Moore model, named after another pioneer E.F.Moore.The counters are clearly Moore machines as the output depends only on the states of the flip-flops.Likewise, a sequence detector is also a Moore machine.Serial adder is an example of a Mealy machine as each one of its states is reached producing a 0 or 1 output depending on the starting state and the value of the inputs.

Mealy To Moore Conversion

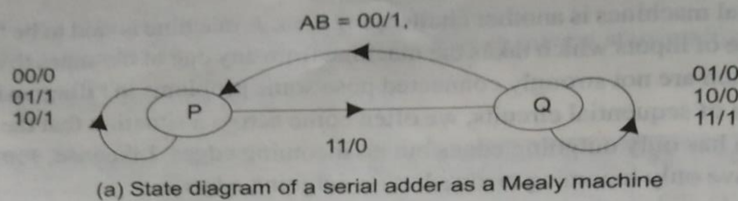
Let us learn how to convert a Mealy machine into a Moore machine .The state diagram and the state table of the synchronous serial adder are given below.Notice that the state P is reached from the state Q on the application of the inputs AB=00 and ,in the process, the machine produces an

output Z=1 indicated on the arc as 00/1.Also notice that the machine produces Z=0 in another transition to P.This transition is indicated as a self loop around P on inputs AB=00/0.For AB=01 or 10 while in P, the machine produces an output 1 and remains in the same state P.

The two important observations are

1.If the Mealy machine has K states, the equivalent Moore machine will have at most 2K states if the output is a binary variable.

2.There is no power-on state,unless specifically defined.A special state may be introduced if the user wants one.



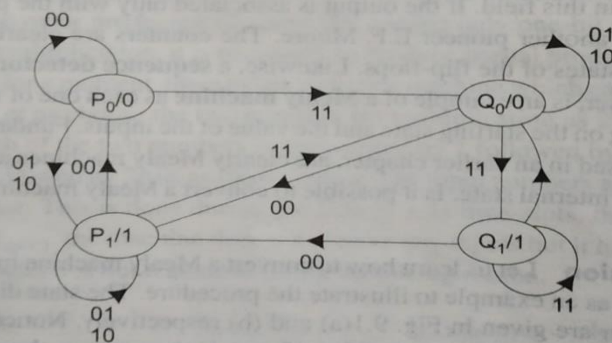
(a) State diagram of a serial adder as a Mealy machine

PS	NS, Z Inputs A B			
	00	01	11	10
P	P, 0	P, 1	Q, 0	P, 1
Q	P, 1	Q, 0	Q, 1	Q, 0

(b) Mealy state table

PS	NS, AB				Z
	00	01	11	10	
P_0	P_0	P_1	Q_0	P_1	0
P_1	P_0	P_1	Q_0	P_1	1
Q_0	P_1	Q_0	Q_1	Q_0	0
Q_1	P_1	Q_0	Q_1	Q_0	1

(c) Moore state table



(d) State diagram of a serial adder as a Moore machine

Moore To Mealy Conversion

It is amazingly simple to convert a Moore to Mealy machine.Let some state si given Moore machine be associated with an output Zi.What we need to do is simply associate the output Zi

where Si occurs as the next state by scanning all the input columns.All states of Moore machine are Z homogeneous.

Reduction Of State Tables Using Partition Technique:

Clearly, the 0 partition P0 contains all the states of the machine in one group indicated by brackets, because by applying 0 inputs, that is ,no inputs at all, it is not possible to distinguish between the states.If you apply any one input, either x=0 or x=1, observe that the outputs ,A,B,F cause the corresponding output pattern to be 00 while the states C,D,E cause the outputs to be 01 in the two input columns of the corresponding rows. Thus, by merely observing the outputs , we may form the 1-partition P1 as (ABF),(CDE).

If each successor pair is within one bracket , we retain the pair intact; otherwise we split the pair. Suppose in P2 ,we consider the transition from pair AB in the x=0 column and x=1 column. This means that "equivalence of A and B implies equivalence of A and B"- a strange partition which is to be ignored.

Continuing the process, we find that neither C and E nor D and E can be equivalent. Hence E parts company from CD in the next partition P3 .Continuing further , we find that P4 is identical to P3 and hence we stop here and conclude that no experiment exists to distinguish between the states AB and CD. Hence we taken them as equivalence classes.

Table 9.1 *Illustrative Example*

(a) Machine M_1

PS	NS, Z	
	X = 0	X = 1
A	F, 0	B, 0
B	F, 0	A, 0
C	E, 0	B, 1
D	E, 0	A, 1
E	C, 0	F, 1
F	B, 0	C, 0

(b) K Equivalence Partitions

$P_0 = (ABCDEF)$

$P_1 = (\widehat{ABF})\ (CDE)$

$P_2 = (AB)\ (F)\ (\widehat{CDE})$

$P_3 = (AB)\ (F)\ (CD)\ (E)$

$P_4 = (AB)\ (F)\ (CD)\ (E) \equiv P_3$

Hence $A \equiv B$, $C \equiv D$.

Table 9.2 *Reduced Machine M_2*

(a) Machine M_2

PS	NS, Z	
	X = 0	X = 1
A	F, 0	A, 0
F	A, 0	C, 0
C	E, 0	A, 1
E	C, 0	F, 1

(b) Row-wise Occurrence of States

AFA
FAC
CEA
ECF
Set $A = S_1$, $F = S_2$
$C = S_3$, $E = S_4$
and rewrite the table.

(c) Transformed Table

PS	NS, Z	
	X = 0	X = 1
S_1	S_2, 0	S_1, 0
S_2	S_1, 0	S_3, 0
S_3	S_4, 0	S_1, 1
S_4	S_3, 0	S_2, 1

(d) RSFST

PS	NS, Z	
	X = 0	X = 1
A	B, 0	A, 0
B	A, 0	C, 0
C	D, 0	A, 1
D	C, 0	B, 1

Derivation Of Flip-Flop Input Equations

After the number of states in a state table has been reduced, the following procedure can be used to derive the flip-flop input equations:

1. Assign flip-flop state values to correspond to the states in the reduced table.

2. Construct a transition table which gives the next states of the flip-flops as a function of the present states and inputs.

3. Derive the next state maps from the transition table.

We could make a straight binary state assignment for which S0 is represented by flip-flops states ABC=000, S1 by ABC =001, S2 by ABC=010, etc. However because the correspondence between flip-flops states and the state names is arbitrary , we could use many different state assignments. Using a different assignment may lead to simpler or more complex flip-flops input equations.

S0=000, S1=110, S2=001 , S3=111, S4=011, S5=101, S6=010

-6 (a) State table

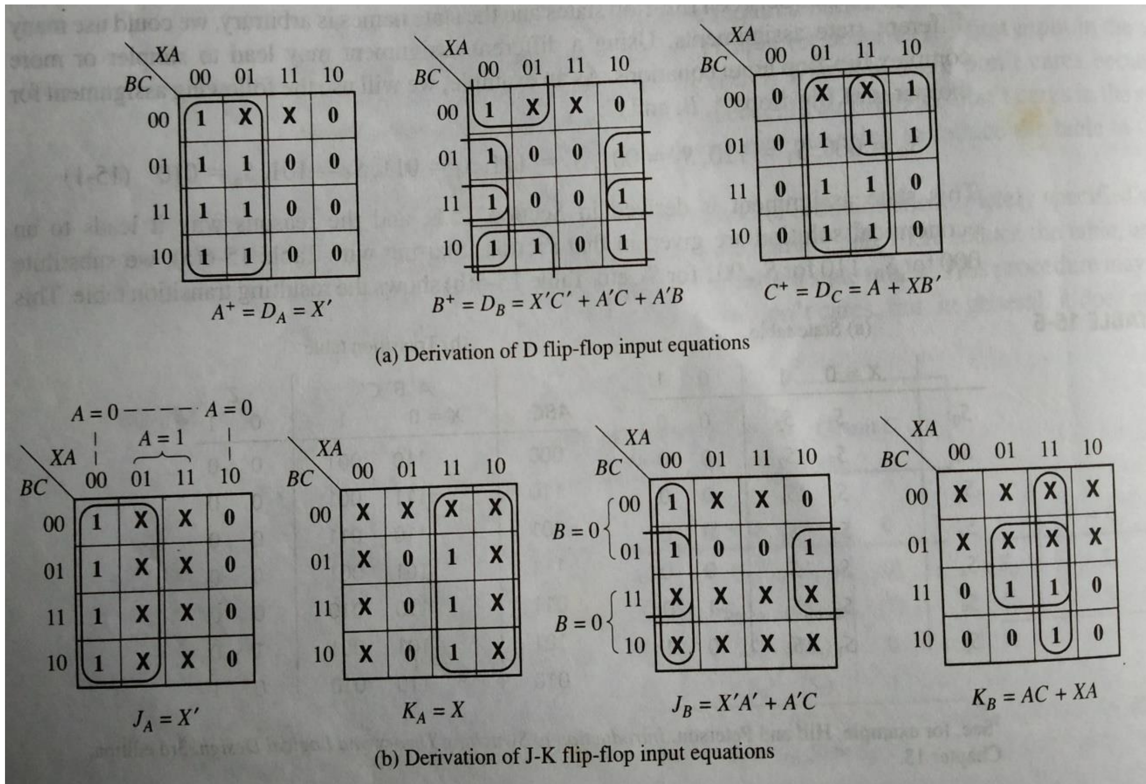
	X = 0	1	0	1
S_0	S_1	S_2	0	0
S_1	S_3	S_2	0	0
S_2	S_1	S_4	0	0
S_3	S_5	S_2	0	0
S_4	S_1	S_6	0	0
S_5	S_5	S_2	1	0
S_6	S_1	S_6	0	1

(b) Transition table

	$A^+B^+C^+$		Z	
ABC	X = 0	1	0	1
000	110	001	0	0
110	111	001	0	0
001	110	011	0	0
111	101	001	0	0
011	110	010	0	0
101	101	001	1	0
010	110	010	0	1

For XABC=0000 the next state entry is 110, so we fill in A+ =1 , B+ =1, C+=0.The below figure shows the D flipflop input equations can be derived directly from the next state maps because DA=A+ , DB=B+, DC=C+. If J-K flip-flops are used, the J and K input equations can be derived from the next state maps as shown below.

A sequential circuit with two inputs (X1 and X2) and two outputs (Z1 and Z2).Note that the column headings are listed in Karnaugh map order because this will facilitate derivation of the flip-flops input equations. Because the table has four states, two flip-flops (A and B) are required to realize the table.



(a) Derivation of D flip-flop input equations

(b) Derivation of J-K flip-flop input equations

(a) State table

P.S.	Next State $X_1X_2 =$ 00	01	11	10	Outputs (Z_1Z_2) $X_1X_2 =$ 00	01	11	10
S_0	S_0	S_0	S_1	S_1	00	00	01	01
S_1	S_1	S_3	S_2	S_1	00	10	10	00
S_2	S_3	S_3	S_2	S_2	11	11	00	00
S_3	S_0	S_3	S_2	S_0	00	00	00	00

(b) Transition table

AB	A^+B^+ $X_1X_2 =$ 00	01	11	10	Outputs (Z_1Z_2) $X_1X_2 =$ 00	01	11	10
00	00	00	01	01	00	00	01	01
01	01	10	11	01	00	10	10	00
11	10	10	11	11	11	11	00	00
10	00	10	11	00	00	00	00	00

If J-K, T, or S-R flip-flops are used, the flip-flops input maps can be derived from the next state maps.

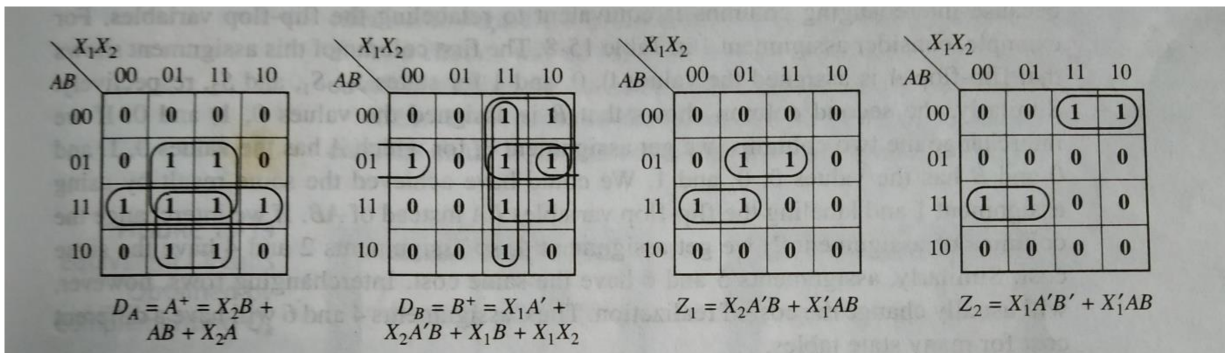
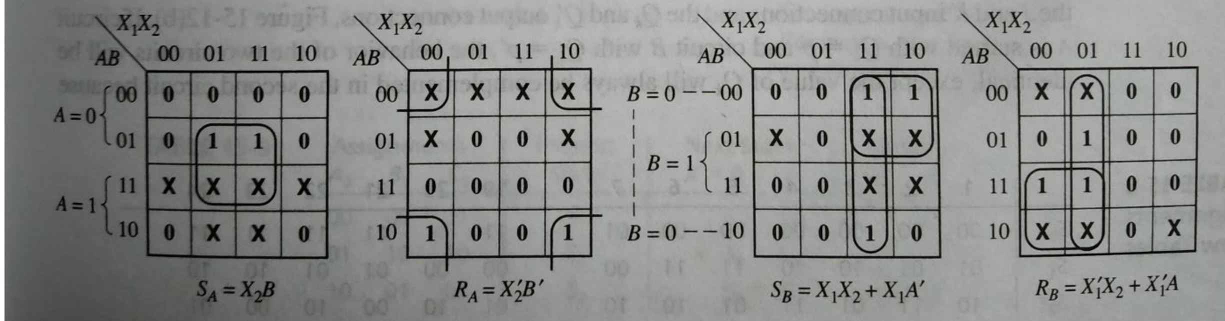


FIGURE 15-11 Derivation of S-R Equations for Table 15-7

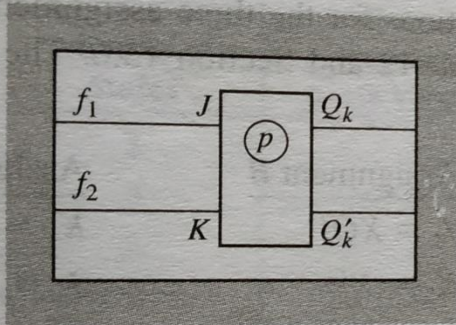


Equivalent State Assignments

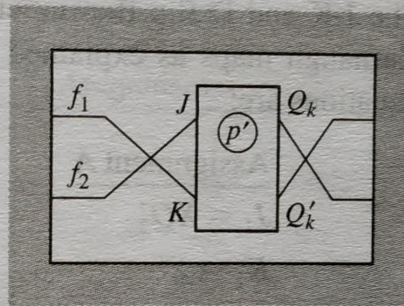
After the number of states in a state table has been reduced, the next step in realizing the table is to assign flip-flop states to correspond to the states in the table. The trail-and-error method described next is useful for only a small number of states. If the number of states is small, it may be feasible to try all possible state assignments, evaluate the cost of the realization for each assignment, and choose the assignment with low cost.

If symmetrical flip-flops such as T, J-K, S-R are used, complementing one or more columns of the state assignment will have no effect on the cost of realization. Consider a J-K flip-flip imbedded in a circuit. Leave the circuit unchanged and interchange the J and K input connections.

	1	2	3	4	5	6	7		19	20	21	22	23	24
S_0	00	00	00	00	00	00	01	···	11	11	11	11	11	11
S_1	01	01	10	10	11	11	00		00	00	01	01	10	10
S_2	10	11	01	11	01	10	10		01	10	00	10	00	01



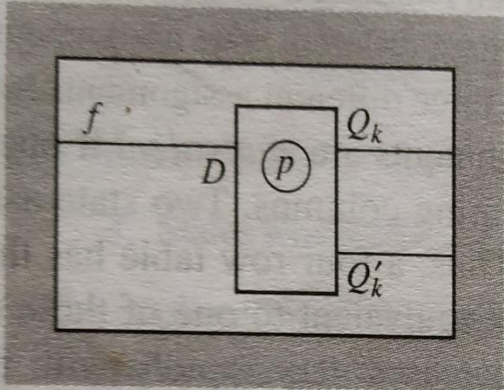
(a) Circuit A



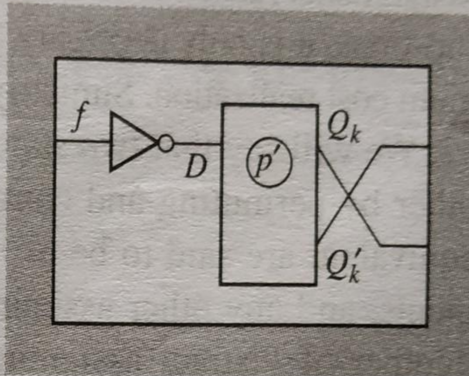
(b) Circuit B
(identical to A except leads to flip-flop Q_k are crossed)

If unsymmetrical flip-flops are used such as D flip-flop , it is still true that permuting columns in the state assignment will not affect the cost; however complementing a column may require adding an inverter to the circuit .

If different types of gates are available the circuit can generally be redesigned to eliminate the inverter and use the same number of gates as the original.



(a) Circuit *A*



(b) Circuit *B*
(identical to *A* except for connections to flip-flop Q_k)

Assignments A_3	B_3	C_3	Present State	Next State $X = 0$	1	Output 0	1
00	00	11	S_1	S_1	S_3	0	0
01	10	10	S_2	S_2	S_1	0	1
10	01	01	S_3	S_2	S_3	1	0

The J-K and D flip-flops input equations for the three assignments can be derived using Karnaugh maps.

Assignment A	Assignment B	Assignment C
$J_1 = XQ_2'$	$J_2 = XQ_1'$	$K_1 = XQ_2$
$K_1 = X'$	$K_2 = X'$	$J_1 = X'$
$J_2 = X'Q_1$	$J_1 = X'Q_2$	$K_2 = X'Q_1'$
$K_2 = X$	$K_1 = X$	$J_2 = X$
$Z = X'Q_1 + XQ_2$	$Z = X'Q_2 + XQ_1$	$Z = X'Q_1' + XQ_2'$
$D_1 = XQ_2'$	$D_2 = XQ_1'$	$D_1 = X' + Q_2'$
$D_2 = X'(Q_1 + Q_2)$	$D_1 = X'(Q_2 + Q_1)$	$D_2 = X + Q_1Q_2$

We will say that two state assignments are equivalent if one can be derived from the other by permuting and complementing columns.Two state assignments which are not equivalent are said to be distinct. Hand solution is feasible for two, three,or four states ; computer solution is feasible for five through eight states; but more than nine states it is not practically to try all assignments even if high-speed computer is used.

Number of States	Minimum Number of State Variables	Number of Distinct Assignments
2	1	1
3	2	3
4	2	3
5	3	140
6	3	420
7	3	840
8	3	840
9	4	10,810,800
⋮	⋮	⋮
16	4	$\approx 5.5 \times 10^{10}$

IMPORTANT TERMS:

Terminal state:

A terminal state is a state with no incoming arcs which start from other states and terminate on it.

Strongly connected machine:

A sequential machine M is said to be strongly connected, if for every pair of states s_i, sj of the sequential machine,there exists an input sequence which takes the machine M from s_i to sj.

Redundant states:

Redundant states are states whose functions can be accomplished by other states.

Equivalent states:

Two states are said to be equivalent if for every possible set of inputs they generate exactly the same output and the same next state.

When equivalent states are there, one of them can be retained and all others can be removed without altering the input-output relationship because they are redundant. This results in reduction of states which in turn reduces the number of required flip-flops and logic gates reducing the cost of final circuit.

Unit – V

Programmable Logic Devices & HDL

Objectives:

- To familiarize with the concepts of Programmable Logic Devices (PLDs): PROM, PAL, PLA, CPLDs, FPGAs, etc.
- How to implement various logic circuits using PLDs.
- To explore about Verilog HDL models.

Syllabus:

Types of PLD's- PROM, PAL, PLA, Basic structure of CPLD and FPGA, Advantages of FPGA's, Introduction to Verilog - structural Specification of logic circuits, Behavioral specification of logic circuits, Hierarchical Verilog Code.

Outcomes:

Students will be able to

- draw the basic structures of PLDs.
- realize various logic functions using PLDs.
- explain various models in verilog HDL.

Learning Material

5.1 Programmable Logic Devices (PLDs)

Introduction:

An IC that contains large numbers of gates, flip-flops, etc. that can be *configured by the user* to perform different functions is called a ***Programmable Logic Device (PLD)***. The internal logic gates and/or connections of PLDs can be changed/configured by a programming process. One of the simplest programming technologies is to use fuses. In the original state of the device, all the fuses are intact. Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function. PLDs are typically built with an *array* of AND gates (AND-array) and an *array* of OR gates (OR-array) is shown in figure 5.1.

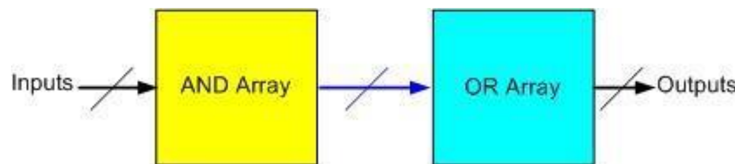


Fig 5.1 Basic PLD structure

Advantages of PLDs:

Problems of using standard ICs:

Problems of using standard ICs in logic design are that they require hundreds or thousands of these ICs, considerable amount of circuit board space, a great deal of time and cost in inserting, soldering, and testing. Also require keeping a significant inventory of ICs.

Advantages of using PLDs:

Advantages of using PLDs are less board space, faster, lower power requirements (i.e., smaller power supplies), less costly assembly processes, higher reliability (fewer ICs and circuit connections means easier troubleshooting), and availability of design software.

Types of PLDs:

There are three fundamental types of standard PLDs: **PROM, PAL,** and **PLA**. A fourth type of PLD is the **Complex Programmable Logic Device (CPLD)** and next one is **Field Programmable Gate Array (FPGA).** A typical PLD may have hundreds to millions of gates. In order to show the internal logic diagram for such technologies in a concise form, it is necessary to have special symbols for array logic. Figure 5.2 shows the conventional and array logic symbols for a multiple input AND and a multiple input OR gate.

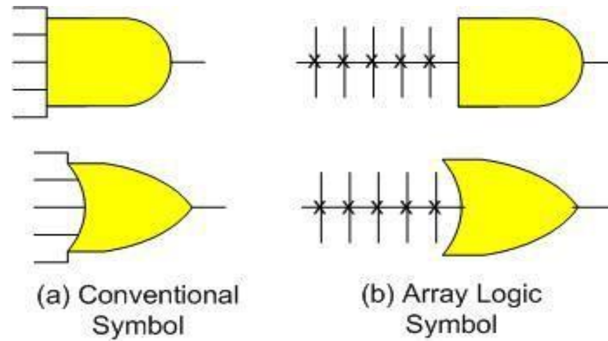


Fig 5.2 Symbol for both conventional and array

Three Fundamental Types of PLDs:

The three fundamental types of PLDs differ in the placement of programmable connections in the AND-OR arrays. Figure 5.3 shows the locations of the programmable connections for the three types.

The ***PROM (Programmable Read Only Memory)*** has a fixed AND array (constructed as a decoder) and programmable connections for the output OR gates array. The PROM implements Boolean functions in sum-of-minterms form.

The ***PAL (Programmable Array Logic)*** device has a programmable AND array and fixed connections for the OR array.

The ***PLA (Programmable Logic Array)*** has programmable connections for both AND and OR arrays. So it is the most flexible type of PLD.

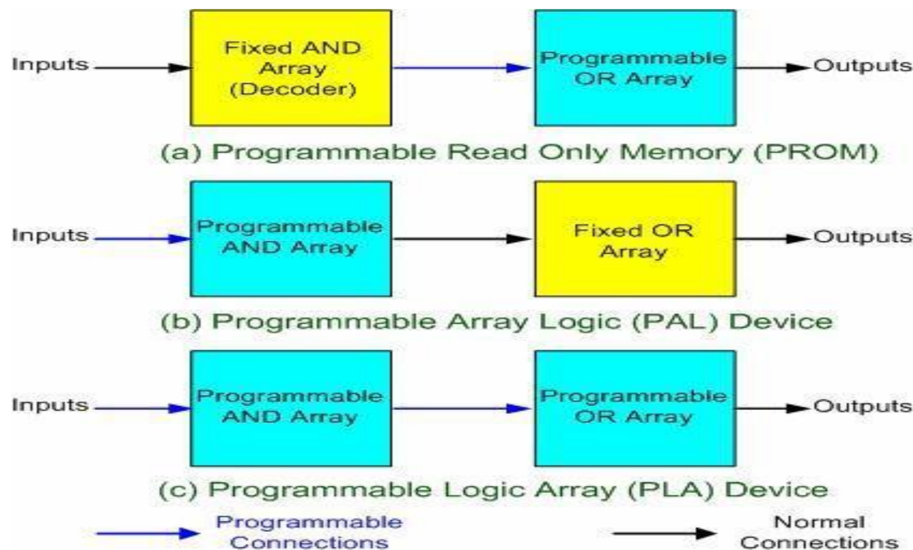


Fig 5.3 General structure notations for various types of PLDs

The ROM (Read Only Memory) or PROM (Programmable Read Only Memory):

The input lines to the AND array are hard-wired and the output lines to the OR array are programmable. Each AND gate generates one of the possible AND products (i.e., minterms). implement the following Boolean functions using PROM.

Example:

A(X,Y,Z)=∑m(5,6,7), B(X,Y,Z)=∑m(3,5,6,7)

The given two functions are in sum of min terms form and each function is having three variables X, Y & Z. So, it requires a 3 to 8 decoder and two programmable OR gates for producing these two functions. The corresponding PROM is shown in the following figure 5.4.

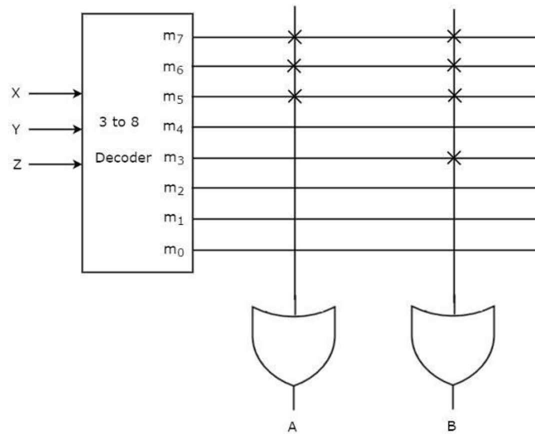


Fig 5.4 PROM design for given example

Here, 3 to 8 decoder generates eight min terms. The two programmable OR gates have the access of all these min terms. But, only the required min terms are programmed in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.

The PLA (Programmable Logic Array):

In PLAs, instead of using a decoder as in PROMs, a number (k) of AND gates is used where k < 2n, (n is the number of inputs). Each of the AND gates can be programmed to generate a product term of the input variables and does not generate all the minterms as in the ROM. The AND and OR gates inside the PLA are initially fabricated with the links (fuses) among them. The specific Boolean functions are implemented in sum of products form by opening appropriate links and leaving the desired connections. A block diagram of the PLA is shown in the figure 5.5. It consists of n inputs, m outputs, and k product terms.

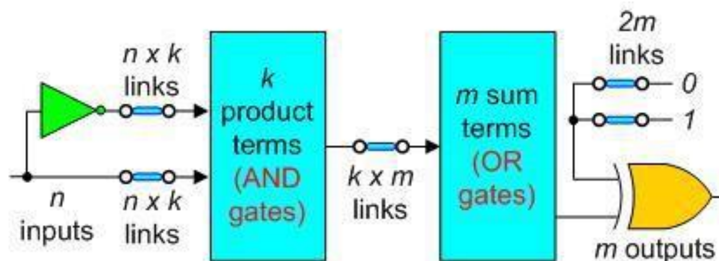


Fig 5.5 Block diagram for PLA

The product terms constitute a group of ***k*** AND gates each of *2n* inputs. Links are inserted between all ***n*** inputs and their complement values to each of the AND gates. Links are also

provided between the outputs of the AND gates and the inputs of the OR gates. Since PLA has ***m***-outputs, the number of OR gates is ***m***. The output of each OR gate goes to an XOR gate, where the other input has two sets of links, one connected to logic 0 and other to logic 1. It allows the output function to be generated either in the **true** form or in the **complement** form. The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = X'$). The output does

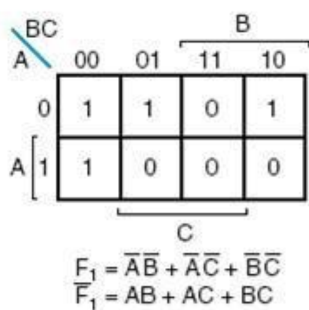
not change when the XOR input is connected to 0 (since $X \oplus 0 = X$). Thus, the total number of programmable links is $2n \times k + k \times m + 2m$. The size of the PLA is specified by the number of inputs ***(n)***, the number of product terms ***(k)***, and the number of outputs ***(m)***, (the number of sum terms is equal to the number of outputs).

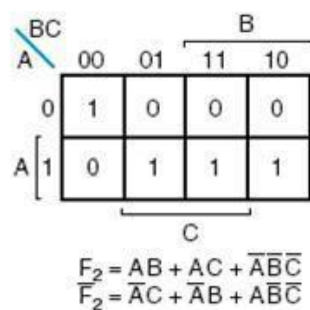
Example:

Implement the combinational circuit having the shown truth table, using PLA.

A	*B*	*C*	F_1	F_2
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Each product term in the expression requires an AND gate. To minimize the cost, it is necessary to simplify the function to a minimum number of product terms.





Designing using a PLA, a careful investigation must be taken in order to reduce the distinct product terms. Both the true and complement forms of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions. The combination that gives a minimum number of product terms is:

$F_1' = AB + AC + BC$ *or* $F_1 = (AB + AC + BC)'$

$F_2 = AB + AC + A'B'C'$

This gives only 4 distinct product terms: *AB, AC, BC,* and *A'B'C'*.

So the PLA table will be as follows:

PLA programming table

	Product term	Inputs A B C	Outputs (C) F_1	Outputs (T) F_2
AB	1	1 1 –	1	1
AC	2	1 – 1	1	1
BC	3	– 1 1	1	–
$\overline{A}\overline{B}\overline{C}$	4	0 0 0	–	1

For each product term, the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1. A 1 in the **Inputs** column specifies a path from the corresponding input to the input of the AND gate that forms the product term. A 0 in the **Inputs** column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection. The appropriate fuses are blown and the ones left intact form the desired paths. It is assumed that the open terminals in the AND gate behave like a 1 input. In the outputs column, a **T (true)** specifies that the other input of the corresponding XOR gate can be connected to 0, and a **C (complement)** specifies a connection to 1. Note that output F_1 is the normal (or true) output even though a **C** (for complement) is marked over it. This is because F_1' is generated with AND-OR circuit prior to the output XOR. The output XOR complements the function F_1' to produce the true F_1 output as its second input is connected to logic 1 is shown in figure 5.6.

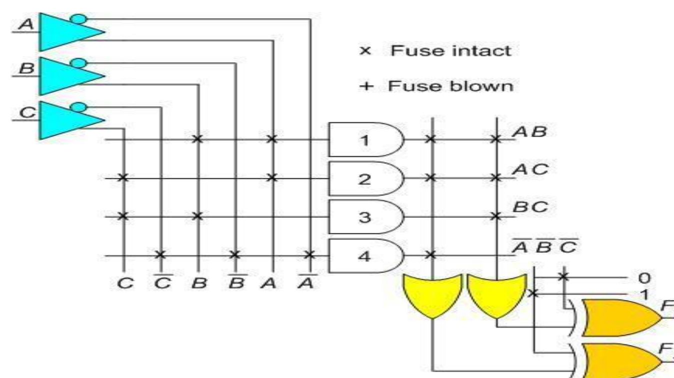


Fig 5.6 PLA realization for given example

The PAL (Programmable Array Logic):

The PAL device is a PLD with a fixed OR array and a programmable AND array. As only AND gates are programmable, the PAL device is easier to program but it is not as flexible as the PLA.

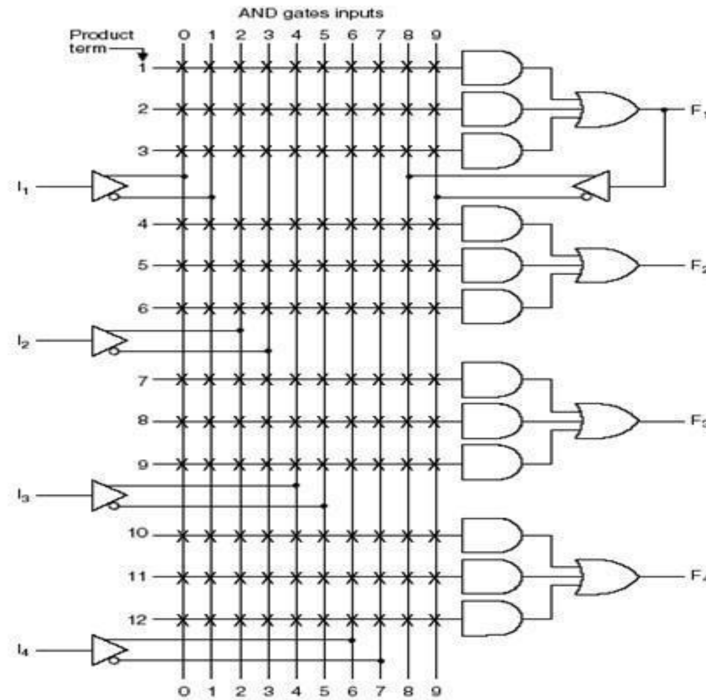


Fig 5.7 4 inputs and 4 outputs PAL Example

The device shown in the figure 5.7 has 4 inputs and 4 outputs. Each input has a buffer- inverter gate, and each output is generated by a fixed OR gate. The device has 4 sections, each composed of a 3-wide AND-OR array, meaning that there are 3 programmable AND gates in each section. Each AND gate has 10 programmable input connections indicating by
10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple input configuration of an AND gate. One of the outputs F_1 is connected to a buffer-inverter gate and is fed back into the inputs of the AND gates through programmed connections. Designing using a PAL device, the Boolean functions must be simplified to fit into each section. The number of product terms in each section is fixed and if the number of terms in the function is too large, it may be necessary to use two or more sections to implement one Boolean function.

Example:

Implement the following Boolean functions using the PAL device as shown above:

$W(A, B, C, D) = \sum m(2, 12, 13)$
$X(A, B, C, D) = \sum m(7, 8, 9, 10, 11, 12, 13, 14, 15)$
$Y(A, B, C, D) = \sum m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$
$Z(A, B, C, D) = \sum m(1, 2, 8, 12, 13)$

Simplifying the 4 functions to a minimum number of terms results in the following Boolean functions:

W = ABC' + A'B'CD'
X = A + BCD
Y = A'B + CD + B'D'
Z = ABC' + A'B'CD + AC'D' + A'B'C'D
=W +AC'D' + A'B'C'D

Note that the function for ***Z*** has four product terms. The logical sum of two of these terms is equal to ***W***. Thus, by using ***W***, it is possible to reduce the number of terms for ***Z*** from four to three, so that the function can fit into the given PAL device. The PAL programming table is similar to the table used for the PLA, except that only the inputs of the AND gates need to be programmed.

Product term	AND Inputs					Outputs
	A	B	C	D	W	
1	1	1	0	—	—	$W = AB\overline{C}$
2	0	0	1	0	—	$+\overline{A}\,\overline{B}C\overline{D}$
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$
5	—	1	1	1	—	$+BCD$
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A}B$
8	—	—	1	1	—	$+CD$
9	—	0	—	0	—	$+\overline{B}\,\overline{D}$
10	—	—	—	—	1	$Z = W$
11	1	—	0	0	—	$+A\overline{C}\,\overline{D}$
12	0	0	0	1	—	$+\overline{A}\,\overline{B}\,\overline{C}D$

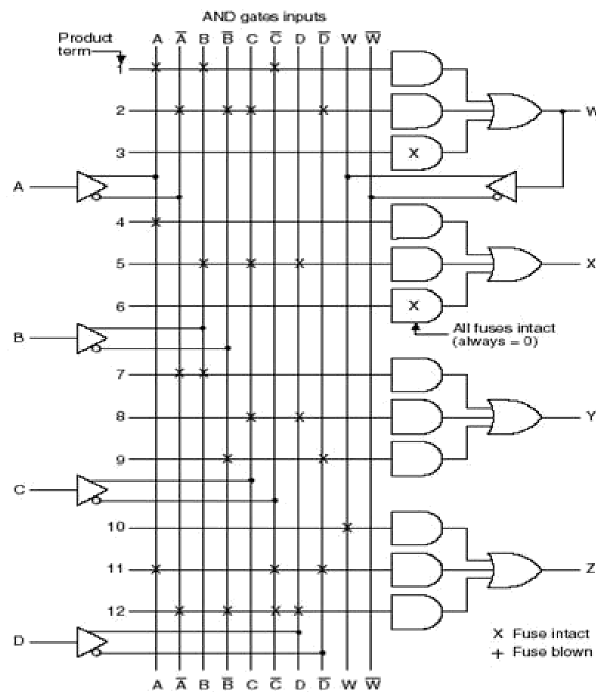


Fig 5.8 4 inputs and 4 outputs PAL Example with three AND gates

The figure 5.8 shows the connection map for the PAL device, as specified in the programming table. Since both W and X have two product terms, third AND gate is not used. If all the inputs to this AND gate left intact, then its output will always be 0, because it receives both the true and complement of each input variable i.e., AA' =0

Comparison between PROM, PLA and PAL

PROM	PLA	PAL
AND array is fixed and OR array is programmable.	Both AND and OR arrays are programmable.	OR array is fixed and AND array is programmable.
Cheaper and simple to use.	Costliest and complex than PAL and PROMs,	Cheaper and simpler.
Ail minterms are decoded.	AND array can be programmed to get desired minterms.	AND array can be programmed to get desired minterms.
Onfy Boolean functions in Standard SOP form can be implemented using PROM	Any Boolean functions In SOP form can be implemented using PLA.	Any Boolean functions In SOP form can be implemented using PLA.

Complex Programmable Logic Devices (CPLDs):

With the advancement of technology, it has become possible to produce devices with higher capacities than SPLD's. As chip densities increased, it was natural for the PLD manufacturers to evolve their products into larger (logically, but not necessarily physically) parts called Complex Programmable Logic Devices (CPLDs). For most practical purposes, CPLDs can be thought of as multiple PLDs (plus some programmable interconnect) in a single chip. The larger size of a CPLD allows to implement either more logic equations or a more complicated design. A CPLD contains a bunch of PLD blocks whose inputs and outputs are connected together by a global interconnection matrix. Thus a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed. For most practical purposes, CPLDs can be thought of as multiple PLDs (plus some programmable interconnect) in a single chip. The larger size of a CPLD allows to implement either more logic equations or a more complicated design. A CPLD contains a bunch of PLD blocks whose inputs and outputs are connected together by a global interconnection matrix. Thus a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed.

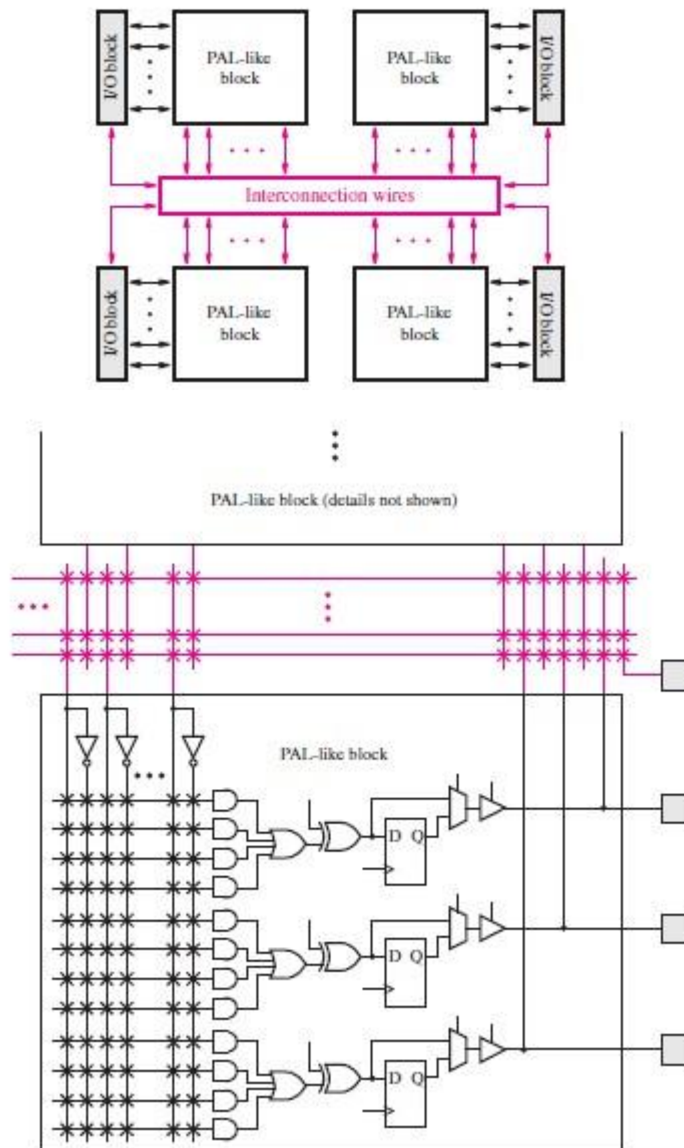


Fig. 5.9 Internal structure and one section of CPLD

In other words, some of the theoretically possible connections between logic block outputs and inputs may not actually be supported within a given CPLD. The effect of this is most often to make 100% utilization of the macrocells very difficult to achieve. Some hardware designs simply won't fit within a given CPLD, even though there are sufficient logic gates and flip-flops available. Because CPLDs can hold larger designs than PLDs, their potential uses are more varied. They are still sometimes used for simple applications like address decoding, but more often contain high-performance control-logic or complex finite state machines. At the high-end (in terms of numbers of gates), there is also a lot of overlap in potential applications with FPGAs.

Traditionally, CPLDs have been chosen over FPGAs whenever high-performance logic is required. Because of its less flexible internal architecture, the delay through a CPLD (measured in nanoseconds) is more predictable and usually shorter.

Field Programmable Gate Arrays (FPGAs):

The development of the FPGA was distinct from the SPLD/CPLD evolution. FPGAs offer the highest amount of logic density, the most features, and the highest performance. The largest FPGA now shipping, part of the Xilinx Virtex™ line of devices, provides eight million "system gates" (the relative density of logic). These advanced devices also offer features such as built-in hardwired processors (such as the IBM Power PC), substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies. FPGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing. The value of programmable logic has always been its ability to shorten development cycles for electronic equipment manufacturers and help them get their product to market faster. As PLD (Programmable Logic Device) suppliers continue to integrate more functions inside their devices, reduce costs, and increase the availability of time- saving IP cores, programmable logic is certain to expand its popularity with digital designers. An FPGA is a device that contains a matrix of reconfigurable gate array logic circuitry. When a FPGA is configured, the internal circuitry is connected in a way that creates a hardware implementation of the software application. Unlike processors, FPGAs use dedicated hardware for processing logic and do not have an operating system. FPGAs are truly parallel in nature so different processing operations do not have to compete for the same resources. As a result, the performance of one part of the application is not affected when additional processing is added. Also, multiple control loops can run on a single FPGA device at different rates. FPGA-based control systems can enforce critical interlock logic and can be designed to prevent I/O forcing by an operator. However, unlike hard-wired printed circuit board (PCB) designs which have fixed hardware resources, FPGA-based systems can literally rewire their internal circuitry to allow reconfiguration after the control system is deployed to the field. FPGA devices deliver the performance and reliability of dedicated hardware circuitry.

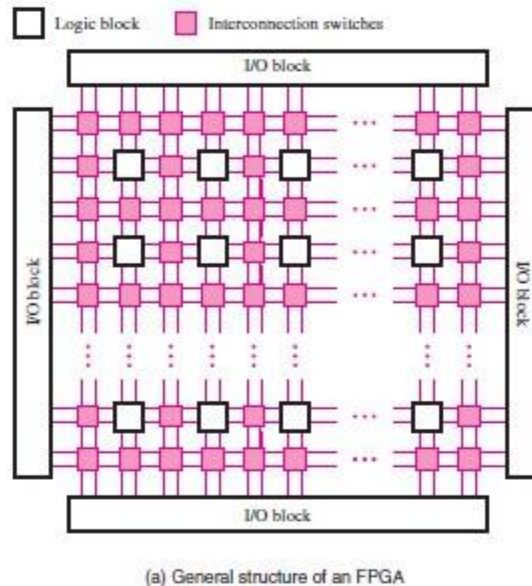


Fig. 5.10 Internal Structure of FPGA

A single FPGA can replace thousands of discrete components by incorporating millions of logic gates in a single integrated circuit (IC) chip. The internal resources of an FPGA chip consist of a matrix of configurable logic blocks (CLBs) surrounded by a periphery of I/O blocks shown in Fig. 5.10. Signals are routed within the FPGA matrix by programmable interconnect switches and wire routes. In an FPGA logic blocks are implemented using multiple level low fan-in gates, which gives it a more compact design compared to an implementation with two-level AND-OR logic. FPGA provides its user a way to configure:

1. The intersection between the logic blocks and
2. The function of each logic block.

Logic block of an FPGA can be configured in such a way that it can provide functionality as simple as that of transistor or as complex as that of a microprocessor. It can used to implement different combinations of combinational and sequential logic functions. Logic blocks of an FPGA can be implemented by any of the following:

1. Transistor pairs
2. combinational gates like basic NAND gates or XOR gates
3. n-input Lookup tables; two input LUT is shown in figure 5.11.
4. Multiplexers
5. Wide fan-in And-OR structure.

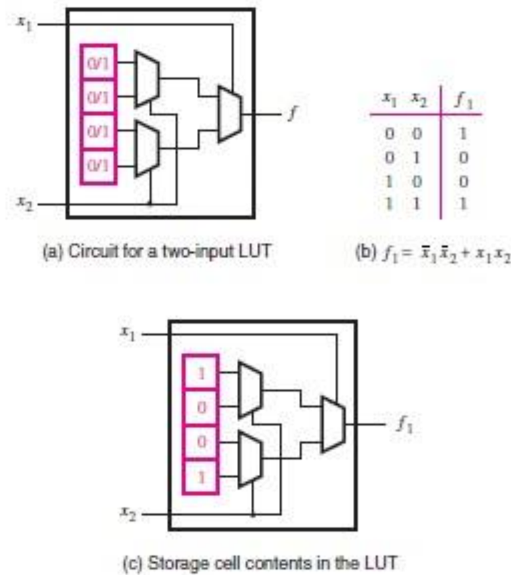


Fig. 5.11 A Two input LUT

The FPGA consists of three main structures: 1) Programmable logic structure, 2) Programmable routing structure, and 3) Programmable Input/Output (I/O).

1.Programmable logic structure

The programmable logic structure FPGA consists of a 2-dimensional array of configurable logic blocks (CLBs). Each CLB can be configured (programmed) to implement any Boolean function of its input variables. Typically CLBs have between 4-6 input variables. Functions of larger number of variables are implemented using more than one CLB. In addition, each CLB typically contains 1 or 2 FFs to allow implementation of sequential logic. Large designs are partitioned and mapped to a number of CLBs with each CLB configured (programmed) to perform a particular function. These CLBs are then connected together to fully implement the target design. Connecting the CLBs is done using the FPGA programmable routing structure.

2. Programmable routing structure

To allow for flexible interconnection of CLBs, FPGAs have 3 *programmable* routing resources: Vertical and horizontal routing channels which consist of different length wires that can be connected together if needed. These channel run vertically and horizontally between columns and rows of CLBs as shown in the figure 5.12 (a). Connection boxes, which are a set of programmable links that can connect input and output pins of the CLBs to wires of the vertical or the horizontal routing channels. Switch boxes, located at the intersection of the vertical and

horizontal channels. These are a set of programmable links that can connect wire segments in the horizontal and vertical channels.

Fig. 5.12 (a) Programmable routing structure

3. Programmable I/O

These are mainly buffers that can be configured either as input buffers, output buffers or input/output buffers. They allow the pins of the FPGA chip to function either as input pins, output pins or input/output pins. programmable I/O block is shown in figure 5.12 (b).

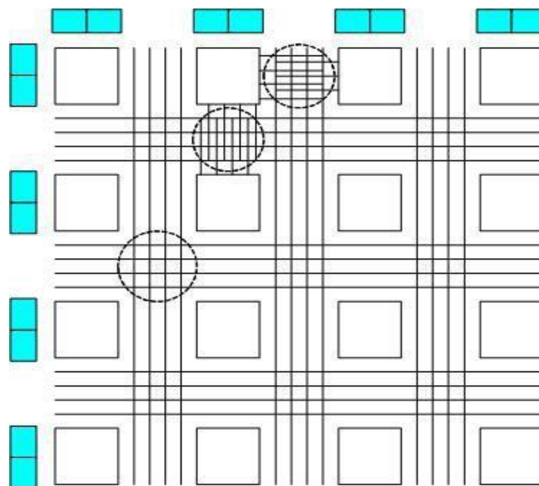


Fig. 5.12 (b) Programmable I/O block

Advantages of FPGA's:

- Short Development time
- Reconfigurable
- Saves board space
- Flexible to changes
- No need for ASIC expensive design and production
- Fast time to market
- Bugs can be fixed easily
- Of the shelf solutions are available

INTRODUCTION TO VERILOG

In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. Verilog was produced as a part of that effort. The original version of Verilog was developed by Gateway Design Automation, which was later acquired by Cadence Design Systems. In 1990 Verilog was put into the public domain, and it has since become one of the most popular languages for describing digital circuits. In 1995 Verilog was adopted as an official IEEE Standard, called 1364-1995. An enhanced version of Verilog, called Verilog 2001, was adopted as IEEE Standard 1364-2001 in 2001. While this version introduced a number of new features, it also supports all of the features in the original Verilog standard. Verilog was originally intended for simulation and verification of digital circuits. Sub- sequently, with the addition of synthesis capability, Verilog has also become popular for use in design entry in CAD systems. The CAD tools are used to synthesize the Verilog code into a hardware implementation of the described circuit. In this book our main use of Verilog will be for synthesis. Verilog is a complex, sophisticated language. Learning all of its features is a daunting task. However, for use in synthesis only a subset of these features is important.

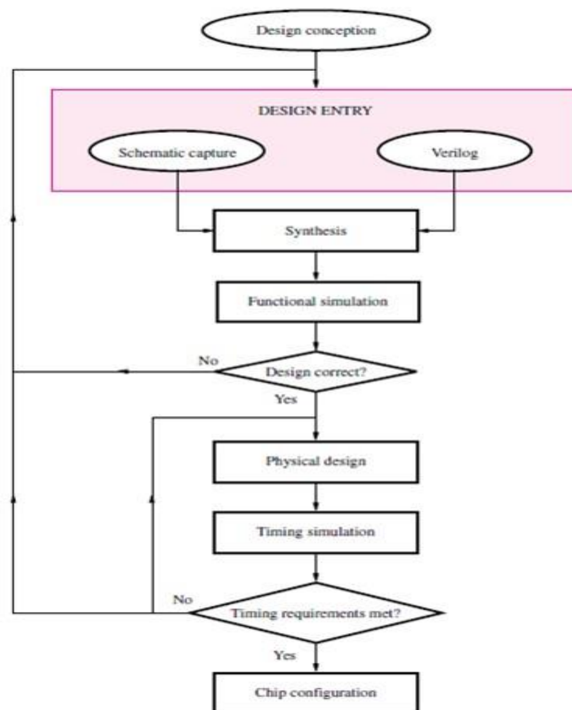


Fig 5.13 A Typical CAD system

Verilog is introduced in several stages throughout the book. Our general approach will be to introduce particular features only when they are relevant to the design topics covered in that part of the text. In Appendix A we provide a concise summary of the Verilog features covered in the book. The reader will find it convenient to refer to that material from time to time. In the remainder of this chapter we discuss the most basic concepts needed to write simple Verilog code. A typical CAD system is given in figure 5.13.

Representation of Digital Circuits in Verilog

When using CAD tools to synthesize a logic circuit, the designer can provide the initial description of the circuit in several different ways, as we explained in the previous section. One efficient way is to write this description in the form of Verilog source code. The Verilog compiler translates this code into a logic circuit. Verilog allows the designer to describe a desired circuit in a number of ways. One possibility is to use Verilog constructs that describe the structure of the circuit in terms of circuit elements, such as logic gates. A larger circuit is defined by writing code that connects such elements together. This approach is referred to as the *structural* representation of logic circuits. Another possibility is to describe a circuit more abstractly, by using logic expressions and Verilog programming constructs that define the desired behavior of the circuit, but not its actual structure in terms of gates. This is called the *behavioral* representation.

STRUCTURAL SPECIFICATION OF LOGIC CIRCUITS

Verilog includes a set of *gate-level primitives* that correspond to commonly-used logic gates. A gate is represented by indicating its functional name, output, and inputs. For example, a two-input AND gate, with output y and inputs x_1 and x_2, is denoted as

and (y, x1, x2);

A four-input OR gate is specified as

or (y, x1, x2, x3, x4);

The keywords **nand** and **nor** are used to define the NAND and NOR gates in the same way. The NOT gate given by **not** (y, x);

implements $y = \bar{x}$. The gate-level primitives can be used to specify larger circuits..

A logic circuit is specified in the form of a *module* that contains the statements that define the circuit. A module has inputs and outputs, which are referred to as its *ports*. The word port is a commonly-used term that refers to an input or output connection to an electronic circuit. Consider the multiplexer circuit from Figure 5.14. This circuit

can be represented by the Verilog code in Figure 5.14. The first statement gives the module a name, *example1*, and indicates that there are four port signals. The next two statements declare that x_1, x_2, and s are to be treated as **input** signals,

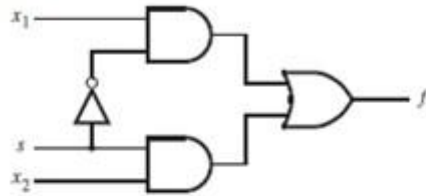


Fig 5.14 A logic circuit for multiplexer

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

Verilog code for the circuit in Figure 5.14.

while f is the **output**. The actual structure of the circuit is specified in the four statements that follow. The NOT gate gives $k = \overline{s}$. The AND gates produce $g = \overline{s}x_1$ and $h = sx_2$. The outputs of AND gates are combined in the OR gate to form

$f = g + h$

$= \overline{s}x_1 + sx_2$

The module ends with the **endmodule** statement.

A second example of Verilog code is given in Figure 5.15. It defines a circuit that has four input signals, x_1, x_2, x_3, and x_4, and three output signals, f, g, and h. It implements the logic functions

$g = x_1x_3 + x_2x_4$

$h = (x_1 + \overline{x}_3)(\overline{x}_2 + x_4)$

$f = g + h$

```
module example2 (x1, x2, x3, x4, f, g, h);
input x1, x2, x3, x4;
output f, g, h;
and (z1, x1, x3);
```

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

Figure 5.15 Verilog code for a four-input circuit.

Instead of using explicit NOT gates to define x_2 and x_3, we have used the Verilog operator "~" (tilde character on the keyboard) to denote complementation. Thus, x_2 is indicated as ~x2 in the code.

Verilog Syntax

The names of modules and signals in Verilog code follow two simple rules: the name must start with a letter, and it can contain any letter or number plus the "_" underscore and "$" characters. Verilog is case sensitive. Thus, the name *k* is not the same as *K* and *Example1* is not the same as *example1*. The Verilog syntax does not enforce a particular style of code. For example, multiple statements can appear on a single line. White space characters, such as SPACE and TAB, and blank lines are ignored. As a matter of good style, code should be formatted in such a way that it is easy to read.

BEHAVIORAL SPECIFICATION OF LOGIC CIRCUITS

Using gate-level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the behavior of a logic circuit. One possibility is to define the circuit using logic expressions. Figure 5.16 shows how the circuit in Figure 5.15 can be defined with the expression $f = sx_1 + sx_2$ –

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

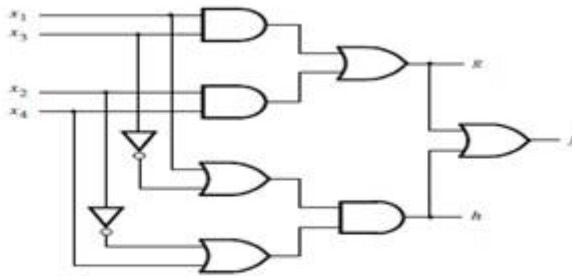


Fig 5.16. A logic circuit for the code given in fig 5.15

```
module example3 (x1, x2, s, f);
input x1, x2, s;
output f;
assign f = (~s & x1) | (s & x2);
endmodule
```

Figure 5.17 Using the continuous assignment

stems from the use of Verilog for simulation; whenever any signal on the right-hand side changes its state, the value of *f* will be re-evaluated. Using logic expressions makes it easier to write Verilog code. But even higher levels of abstraction can often be used to advantage. Consider again the multiplexer circuit of Figure 5.14. The circuit can be described in words by saying that $f = x_1$ if $s = 0$ and $f = x_2$ if $s = 1$. In Verilog, this behavior can be defined with the **if-else** statement

```
if (s == 0)
    f = x1;
else
f = x2;
```

```
module example4 (x1, x2, x3, x4, f, g, h);
input x1, x2, x3, x4;
output f, g, h;
assign g = (x1 & x3) | (x2 & x4);
assign h = (x1 | ~x3) & (~x2 | x4);
assign f = g | h;
endmodule
```

Figure 5.18 Using the continuous assignment to specify the circuit in Figure 5.16.

Behavioral specification

```
module example5 (x1, x2, s, f);
input x1, x2, s; output f;
reg f;
always @(x1 or x2 or s)
if (s == 0)
```

```
f = x1;
else
f = x2;
endmodule
```

Figure 5.19 Behavioral specification

The complete code is given in Figure 5.19. The first line illustrates how a comment can be inserted. The **if-else** statement is an example of a Verilog *procedural* statement. Verilog syntax requires that procedural statements be contained inside a construct called an **always** block, as shown in Figure 5.19. An **always** block can contain a single statement, as in this example, or it can contain multiple statements. A typical Verilog design module may include several **always** blocks, each representing a part of the circuit being modeled. An important property of the **always** block is that the statements it contains are evaluated in the order given in the code. This is in contrast to the continuous assignment statements, which are evaluated concurrently and hence have no meaningful order. The part of the **always** block after the @ symbol, in parentheses, is called the *sensitivity list*. This list has its roots in the use of Verilog for simulation. The statements inside an **always** block are executed by the simulator only when one or more of the signals in the sensitivity list changes value. In this way, the complexity of a imulation process is simplified, because it is not necessary to execute every statement in the code at all times. When Verilog is being employed for synthesis of circuits, as in this book, the sensitivity list simply tells the Verilog compiler which signals can directly affect the outputs produced by the **always** block. If a signal is assigned a value using procedural statements, then Verilog syntax requires that it be declared as a *variable*; this is accomplished by using the keyword **reg** in Figure 5.19. This term also derives from the simulation jargon: It means that, once a variable's value is assigned with a procedural statement, the simulator "registers" this value and it will not change until the **always** block is executed again. Instead of using a separate statement to declare that the variable *f* is of **reg** type in Figure 5.19, we can alternatively use the syntax **output reg** f;

which combines these two statements. Also, Verilog 2001 adds the ability to declare a signal's direction and type directly in the module's list of ports. This style of code is illustrated in Figure 5.20. In the sensitivity list of the **always** statement we can use commas instead of the word **or**, which is also illustrated in Figure 5.20. Moreover, instead of listing the relevant signals in the sensitivity list, it is possible to write simply **always** @(*)

or even more simply

always @∗

```
// Behavioral specification
module  example5 (input x1, x2, s, output reg f);

always @(x1, x2, s)
if (s == 0)
f = x1;
else
f = x2;
endmodule
```

Fig 5.20 A more compact version of the code in Figure 5.19.

HIERARCHICAL VERILOG CODE

The examples of Verilog code given so far include just a single module. For larger designs, it is often convenient to create a hierarchical structure in the Verilog code, in which there is a *top-level* module that includes multiple instances of *lower-level* modules. To see how hierarchical Verilog code can be written consider the circuit in Figure 5.21. The purpose of the circuit is to generate the arithmetic sum of the two inputs *x* and *y*, using the *adder* module, and then to show the resulting decimal value on the 7-segment display. Verilog code for the *adder* module from Figure 5.21 and the *display* module from Figure 5.21 is given in Figures 5.22 and 5.23, respectively. For the *adder* module con- tinuous assignment statements are used to specify the two-bit sum s_1s_0. The assignment statement for s_0 uses the Verilog XOR operator, which is specified as $s_0 = a$ ^ b. The code for the *display* module includes continuous assignment statements that correspond to the

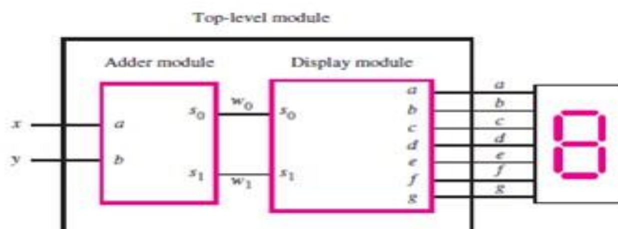


Fig 5.21 A logic circuit with two modules

An adder module

```
module adder (a, b, s1, s0);
input a, b; output s1, s0;
assign s1 = a & b;
assign s0 = a^b;
endmodule
```

Fig 5.22 Verilog specification of the circuit in Figure 5.21.

A module for driving a 7-segment display **module** display (s1, s0, a, b, c, d, e, f, g);

```
input s1, s0;
output a, b, c, d, e, f, g;
assign a = ~s0;
assign b = 1;
assign c = ~s1;
assign d =~s0;
assign e = ~s0;
assign f = ~s1 & ~s0;
assign g = s1 & ~s0;
endmodule
```

Fig 5.23 Verilog specification of the circuit in Figure 5.21

```
module adder_display (x, y, a, b, c, d, e, f, g);
input x, y;
output a, b, c, d, e, f, g;
wire w1, w0;
adder U1 (x, y, w1, w0);
display U2 (w1, w0, a, b, c, d, e, f, g);
endmodule
```

Figure 5.24 Hierarchical Verilog code for the circuit in Figure 5.21

The statement

assign b = 1;

assigns the output *b* of the display module to have the constant value 1. The top-level Verilog module, named *adder_display*, is given in Figure 5.21. This module has the inputs *x* and *y*, and the outputs *a, . . . , g*. The statement

wire w1, w0;

is needed because the signals w_1 and w_0 are neither inputs nor outputs of the circuit in Figure 5.24. Since these signals cannot be declared as input or output ports in the Verilog code, they have to be declared as (internal) *wires*. The statement

adder U1 (x, y, w1, w0);

instantiates the *adder* module from Figure 5.22 as a submodule. The submodule is given a name, U1, which can be any valid Verilog name. The order in which signals are listed in the instantiation statement determines which signal is connected to each port in the submodule. The instantiation statement also attaches the last two ports of the *adder* submodule, which are its outputs, to the wires w1 and w0 in the top-level module. The statement

display U2 (w1, w0, a, b, c, d, e, f, g);

instantiates the other submodule in our circuit. Here, the wires w1 and w0, which have already been connected to the outputs of the *adder* submodule, are attached to the corre-sponding input ports of the *display* submodule. The *display* submodule's output ports are attached to the *a, . . . , g* output ports of the top-level module.

Unit – V

Assignment-Cum-Tutorial Questions

Section-A

1. The inputs in the PLD is given through

   a) NAND gates b) OR gates c) NOR gates d) AND gates

2. PAL refers to

   a) Programmable Array Loaded b) Programmable Logic Array c) Programmable Array Logic d) None of the Mentioned

3. Outputs of the AND gate in PLD is known as

   a) Input lines b) Output lines c) Strobe lines d) None of the Mentioned

4. PLA contains

   a) AND and OR arrays b) NAND and OR arrays c) NOT and AND arrays d) NOR and OR

5. A PLA is similar to a ROM in concept except that

   a) It hasn't capability to read only b) It hasn't capability to read or write operation

   c) It doesn't provide full decoding to the variables d) It hasn't capability to write only

6. For programmable logic functions, which type of PLD should be used?

   a) PLA b) CPLD c) PAL d) SLD

7. The complex programmable logic device contains several PLD blocks and ______________________

   a) A language compiler b) AND/OR arrays c) Global interconnection matrix d) Field-programmable switches

8. PALs tend to execute _________ logic. a) SAP b) SOP c) PLA d) SPD

9. Which type of device FPGA are?

   a) SLD b) SROM c) EPROM d) PLD

10. For designing a 4-variable combinational circuit, a designer must use a a) ROM with atleast 16 locations b) PLA with atleast 32 product terms

    c) PLA with atleast 16 product terms d) PLA with atleast 16 product terms and 16 input OR gate

11. A 32x10 ROM contains a decoder of size a) 5x32 b) 32x32 c) 32x10 d) 10x32

12. Once a PAL has been programmed:

    a) it cannot be reprogrammed. b) its outputs are only active HIGHs c) its outputs are only active LOWs d) its logic capacity is lost

13. Combinational Programmable Logic Devices (PLDs) circuits comprise of --------- a) Only gates b) Only flip flops c) Both a and b d) None of the above

14. Which among the following statement/s is/are not an/the advantage/s of Programmable Logic Devices (PLDs)?

a) Short design cycle b) increased space requirement c)

Increased switching speed d) All of the above

15. The difference between a PAL & a PLA is a)
    PALs and PLAs are the same thing
    b) The PAL has a programmable OR plane and a programmable AND plane, while the PLA only has a programmable AND plane
    c) The PLA has a programmable OR plane and a programmable AND plane, while the PAL only has a programmable AND plane
    d) The PAL has more possible product terms than the PLA

16. The FPGA refers to

    a) First programmable Gate Array   b) Field Programmable Gate Array c) First Program Gate Array   d) Field Program Gate Array

17. In FPGA, vertical and horizontal directions are separated by

    a) A line b) A channel c) A strobe d) A flip-flop

18. Most FPGA logic modules utilize a(n) _________ approach to create the desired logic functions.

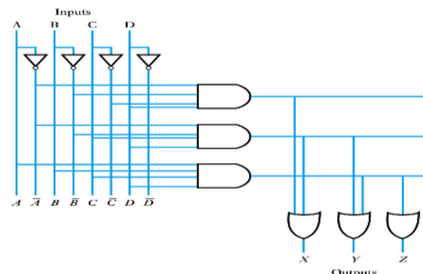
    a) AND array b. Look-up table c. OR array d. AND and OR array

19. PROM stands for

    a) Permanent Read Only Memory b) Portable Read Only Memory c)

    Programmable Read Only Memory d) Plugin Read Only Memory

20. In the following PLA, which output implements the logic function ABCD?



a) X b) Y c) Z d) all of the above

21. Simple Programmable Logic Devices (SPLDs) are also regarded as _______________. a)
    Programmable Array Logic (PAL) b) Generic Array Logic (GAL)
    c) Programmable Logic Array (PLA) d) All of the above

22. The content of a simple programmable logic device (PLD) consists of:

    a) advanced sequential logic functions b) thousands of basic logic gates

    c) thousands of basic logic gates and advanced sequential logic functions d) none

23. The complex programmable logic device (CPLD) contains -------------------

24. State whether the following statements are TRUE or FALSE:

a. Verilog is case sensitive.

b. "beginmodule" and "endmodule" are reserved words in Verilog.

c. The semantics of an "&" operator depends on the number of operands. d. An "if" statement must always be inside of an "always" block.

e. Verilog may be written at the Behavioral, Structural, Gate, Switch, and Transistor levels.

25. How many logic values defined in Verilog with their strength's

a) One b) Two c) Three d) Four

Section-B

1) Give the classification of PLDs with respect to their programmability.

2) Explain the internal structure of PROM.

3) What is programmable logic array? How it differs from PROM?

4) Give the comparison between PROM, PLA and PAL.

5) Show how these functions can be implemented on a PLA having an 8*8 AND array and a 4X8 OR array.

F1(A,B, C,D) = ∑m (2, 3, 6, 7, 11, 15); F2(A,B, C,D) = ∑m (0, 4, 8, 9, 11, 15) F3(A,B,C,D) = ∑m (1,3,5,7,10,11); F4(A,B,C,D) = ∑m (0, 2, 4, 6, 8, 9, 11, 12, 13, 15)

6) Implement 3-bit binary to gray code converter using PROM.

7) Using PAL, implement full adder digital circuit.

8) Explain levels of design description in Verilog HDL.

9) Design full adder using gate level modeling Verilog HDL.

10) Design 8X1 multiplexer using behavioral flow modeling?

11) Design full adder using half adder using hierarchical flow modeling?

12) Explain design at behavioral levels in HDL.

13) Explain the basic structures of CPLD and FPGA.

14) Realize the following functions using PLA and PAL, and give programming table for both.

F1(A,B, C,D) = ∑m (2 ,3, 6, 7, 10, 14, 15); F2(A,B, C,D) = ∑m (3, 5, 7, 10, 12, 14, 15)

F3(A,B,C,D) = ∑m (2, 3, 7, 8, 9, 12, 13, 14, 15).

15) With neat steps explain about CAD design flow using Verilog HDL.

16) Give advantages of FPGAs over PLDs.

17) Write short notes on important features of Veriog HDL.

Section-C

1) Choose the correct statement from the following. GATE-1992

a) PROM contains a programmable AND array and a fixed OR array. b) PLA

contains a fixed AND array and a programmable OR array.

c) PLA contains a programmable AND array and a programmable OR array d) PROM

contains a fixed AND array and a programmable OR array.

2) A PLA can be used GATE-1994

a) As a microprocessor b) as a dynamic memory
c) to realize a sequential logic d) to realize a combinational logic

3) Which one of the following statements is correct? IES-2013

a) PROM contains a programmable AND array and a fixed OR array b) PLA
contains a fixed AND array and a programmable OR array
c) PROM contains a fixed AND array and programmable OR array
d) PLA contains a programmable AND array and a programmable OR array

4) What is the minimum size of ROM required to implement the given Boolean

function. GATE-1996

$F1 = ABCD + \bar{A}\bar{B}\bar{C}\bar{D}$ $F2 = (A + B)(\bar{A} + \bar{B} + C)$

$F3 = \sum 13, 15$

UNIT – IV

Finite State Machines

Objectives:

- To familiarize with the concepts of Finite state machines.

Syllabus:

Types of FSM, Capabilities and limitations of FSM, State assignment, Realization of FSM using flip-flops, Mealy to Moore conversion and vice-versa, Reduction of state tables using partition technique

Outcomes:

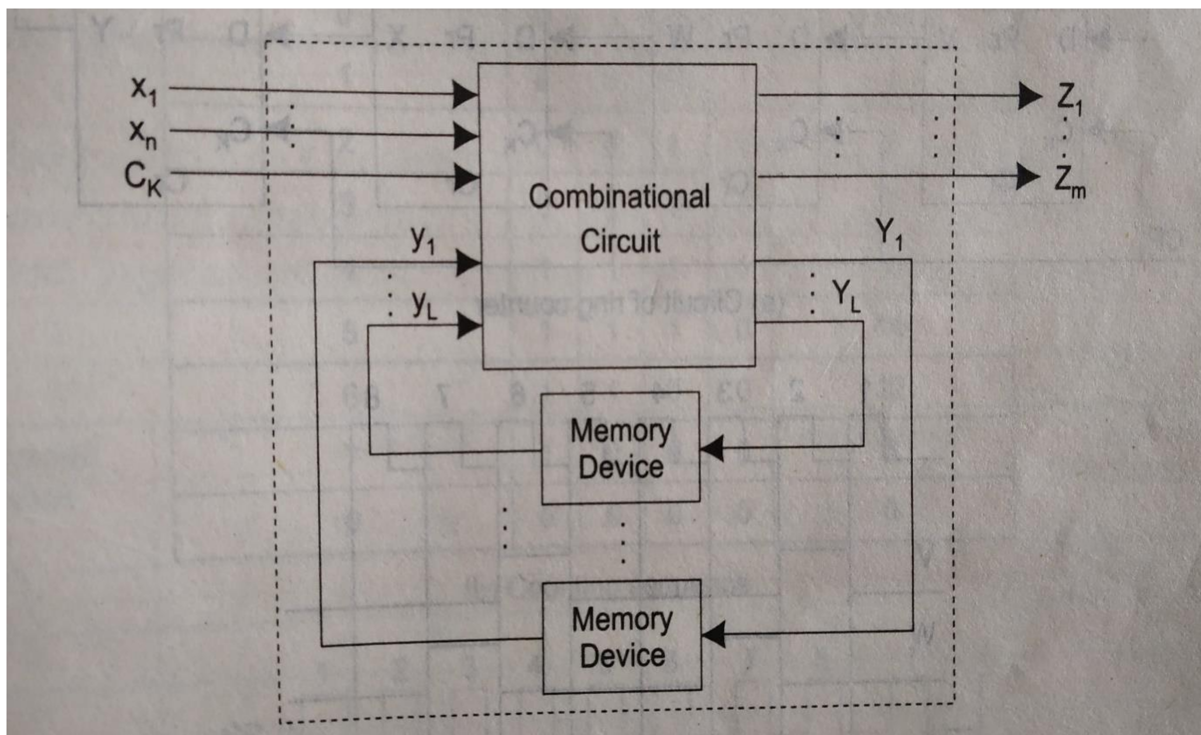
Students will be able to

- design FSM charts using flip flops.
- understand the melay machines and moore machines.
- Reduction methods of state tables.
- partition technique.

Model Of A Finite State Machine (FSM)

It is a Finite State Machine (FSM) also called Finite Automation in the literature pertaining to automata theory. FSM comprises an input set (I), output set (Z),a set of states (S), state transition function (δ), and output function (λ). Thus, the finite state machine M is a quintuple given by M = (I,Z,S, δ, λ),where δ is a function of present state resulting in the next state and λ is a function which enables us to compute the output depending on the present inputs and present state.The previous statement refers to what is generally called the Mealy Machines.

The clock pulses control all timing in the machine . If the clock is removed, the model represents an asynchronous sequential machine with mere delays replacing flip-flops.



Limitations Of Finite State Machines

- No finite state machine can be produced for an infinite sequence.
- No finite state machine can multiply two arbitrary large binary numbers.

No finite state machine can be designed to produce such a non-periodic infinite sequence for a periodic input.

Mealy And Moore Models

A sequential machine M is a quintuple comprising a set of inputs I,a set of outputs Z, a set of states S, a transition function δ which enables finding the next state depending on the present state and present input and finally an output function λ.This is symbolically expressed as M=(I,Z,S, δ, λ).

If the output function depends on the present state and present inputs,it is called the Mealy model,named after G.H.Mealy, a pioneer in the field. If the output is associated only with the present state,it is called the Moore model, named after another pioneer E.F.Moore.The counters are clearly Moore machines as the output depends only on the states of the flip-flops.Likewise,

a sequence detector is also a Moore machine.Serial adder is an example of a Mealy machine as each one of its states is reached producing a 0 or 1 output depending on the starting state and the value of the inputs.

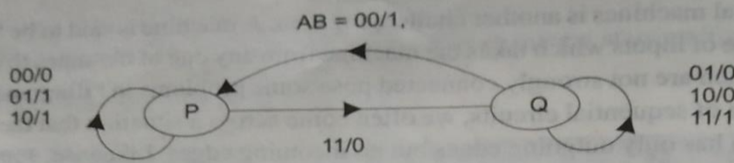
Mealy To Moore Conversion

Let us learn how to convert a Mealy machine into a Moore machine .The state diagram and the state table of the synchronous serial adder are given below.Notice that the state P is reached from the state Q on the application of the inputs AB=00 and ,in the process, the machine produces an output Z=1 indicated on the arc as 00/1.Also notice that the machine produces Z=0 in another transition to P.This transition is indicated as a self loop around P on inputs AB=00/0.For AB=01 or 10 while in P, the machine produces an output 1 and remains in the same state P.

The two important observations are

1.If the Mealy machine has K states, the equivalent Moore machine will have at most 2K states if the output is a binary variable.

2.There is no power-on state,unless specifically defined.A special state may be introduced if the user wants one.



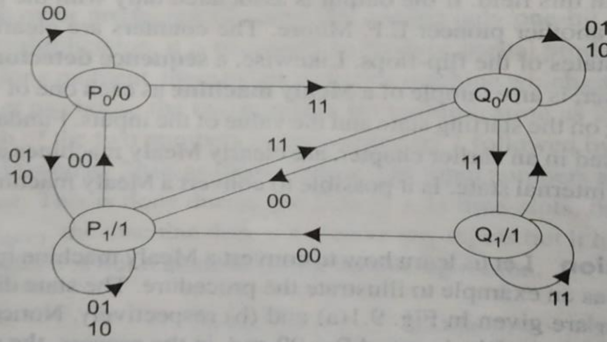
(a) State diagram of a serial adder as a Mealy machine

PS	NS, Z Inputs A B			
	00	01	11	10
P	P, 0	P, 1	Q, 0	P, 1
Q	P, 1	Q, 0	Q, 1	Q, 0

(b) Mealy state table

PS	NS, AB				Z
	00	01	11	10	
P_0	P_0	P_1	Q_0	P_1	0
P_1	P_0	P_1	Q_0	P_1	1
Q_0	P_1	Q_0	Q_1	Q_0	0
Q_1	P_1	Q_0	Q_1	Q_0	1

(c) Moore state table



(d) State diagram of a serial adder as a Moore machine

Moore To Mealy Conversion

It is amazingly simple to convert a Moore to Mealy machine.Let some state si given Moore machine be associated with an output Zi.What we need to do is simply associate the output Zi where Si occurs as the next state by scanning all the input columns.All states of Moore machine are Z homogeneous.

Reduction Of State Tables Using Partition Technique:

Clearly, the 0 partition P0 contains all the states of the machine in one group indicated by brackets, because by applying 0 inputs, that is ,no inputs at all, it is not possible to distinguish between the states.If you apply any one input, either x=0 or x=1, observe that the outputs ,A,B,F cause the corresponding output pattern to be 00 while the states C,D,E cause the outputs to be 01 in the two input columns of the corresponding rows. Thus, by merely observing the outputs , we may form the 1-partition P1 as (ABF),(CDE).

If each successor pair is within one bracket , we retain the pair intact; otherwise we split the pair. Suppose in P2 ,we consider the transition from pair AB in the x=0 column and x=1 column. This means that “equivalence of A and B implies equivalence of A and B”- a strange partition which is to be ignored.

Continuing the process, we find that neither C and E nor D and E can be equivalent. Hence E parts company from CD in the next partition P3 .Continuing further , we find that P4 is identical to P3 and hence we stop here and conclude that no experiment exists to distinguish between the states AB and CD. Hence we taken them as equivalence classes.

Table 9.1 *Illustrative Example*

(a) Machine M_1

PS	NS, Z	
	X = 0	X = 1
A	F, 0	B, 0
B	F, 0	A, 0
C	E, 0	B, 1
D	E, 0	A, 1
E	C, 0	F, 1
F	B, 0	C, 0

(b) K Equivalence Partitions

P_0 = (ABCDEF)

P_1 = $(\widehat{ABF})$ (CDE)

P_2 = (AB) (F) $(\widehat{CDE})$

P_3 = (AB) (F) (CD) (E)

P_4 = (AB) (F) (CD) (E) $\equiv P_3$

Hence A $\equiv$ B, C $\equiv$ D.

Table 9.2 *Reduced Machine M_2*

(a) Machine M_2

PS	NS, Z	
	X = 0	X = 1
A	F, 0	A, 0
F	A, 0	C, 0
C	E, 0	A, 1
E	C, 0	F, 1

(b) Row-wise Occurrence of States

AFA
FAC
CEA
ECF

Set $A = S_1$, $F = S_2$
$C = S_3$, $E = S_4$
and rewrite the table.

(c) Transformed Table

PS	NS, Z	
	X = 0	X = 1
S_1	S_2, 0	S_1, 0
S_2	S_1, 0	S_3, 0
S_3	S_4, 0	S_1, 1
S_4	S_3, 0	S_2, 1

(d) RSFST

PS	NS, Z	
	X = 0	X = 1
A	B, 0	A, 0
B	A, 0	C, 0
C	D, 0	A, 1
D	C, 0	B, 1

Derivation Of Flip-Flop Input Equations

After the number of states in a state table has been reduced, the following procedure can be used to derive the flip-flop input equations:

1. Assign flip-flop state values to correspond to the states in the reduced table.
2. Construct a transition table which gives the next states of the flip-flops as a function of the present states and inputs.
3. Derive the next state maps from the transition table.

We could make a straight binary state assignment for which S0 is represented by flip-flops states ABC=000, S1 by ABC =001, S2 by ABC=010, etc. However because the correspondence between flip-flops states and the state names is arbitrary , we could use many different state assignments. Using a different assignment may lead to simpler or more complex flip-flops input equations.

S0=000, S1=110, S2=001 , S3=111, S4=011, S5=101, S6=010

-6

(a) State table

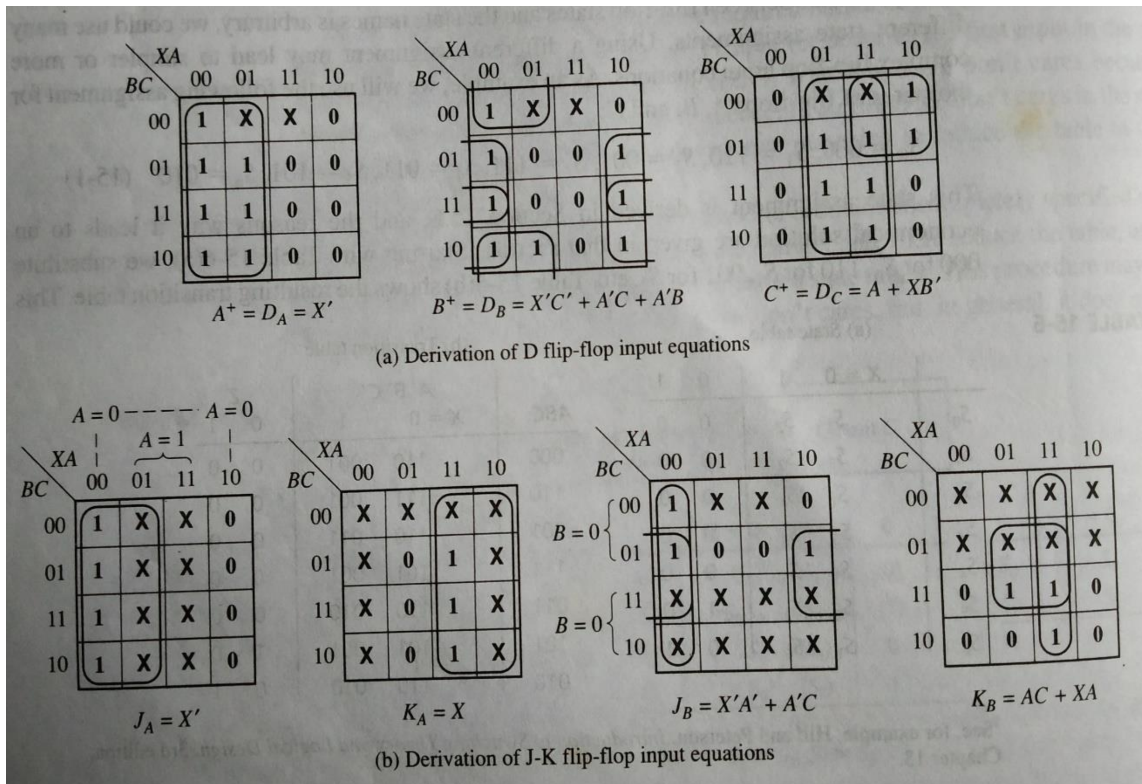
	X = 0	1	0	1
S_0	S_1	S_2	0	0
S_1	S_3	S_2	0	0
S_2	S_1	S_4	0	0
S_3	S_5	S_2	0	0
S_4	S_1	S_6	0	0
S_5	S_5	S_2	1	0
S_6	S_1	S_6	0	1

(b) Transition table

	$A^+B^+C^+$		Z	
ABC	X = 0	1	0	1
000	110	001	0	0
110	111	001	0	0
001	110	011	0	0
111	101	001	0	0
011	110	010	0	0
101	101	001	1	0
010	110	010	0	1

For XABC=0000 the next state entry is 110, so we fill in A+ =1 , B+ =1, C+=0.The below figure shows the D flipflop input equations can be derived directly from the next state maps because DA=A+ , DB=B+, DC=C+. If J-K flip-flops are used, the J and K input equations can be derived from the next state maps as shown below.

A sequential circuit with two inputs (X1 and X2) and two outputs (Z1 and Z2).Note that the column headings are listed in Karnaugh map order because this will facilitate derivation of the flip-flops input equations. Because the table has four states, two flip-flops (A and B) are required to realize the table.



(a) Derivation of D flip-flop input equations

(b) Derivation of J-K flip-flop input equations

(a) State table

P.S.	Next State $X_1X_2 =$ 00	01	11	10	Outputs (Z_1Z_2) $X_1X_2 =$ 00	01	11	10
S_0	S_0	S_0	S_1	S_1	00	00	01	01
S_1	S_1	S_3	S_2	S_1	00	10	10	00
S_2	S_3	S_3	S_2	S_2	11	11	00	00
S_3	S_0	S_3	S_2	S_0	00	00	00	00

(b) Transition table

AB	A^+B^+ $X_1X_2 =$ 00	01	11	10	Outputs (Z_1Z_2) $X_1X_2 =$ 00	01	11	10
00	00	00	01	01	00	00	01	01
01	01	10	11	01	00	10	10	00
11	10	10	11	11	11	11	00	00
10	00	10	11	00	00	00	00	00

If J-K, T, or S-R flip-flops are used, the flip-flops input maps can be derived from the next state maps.

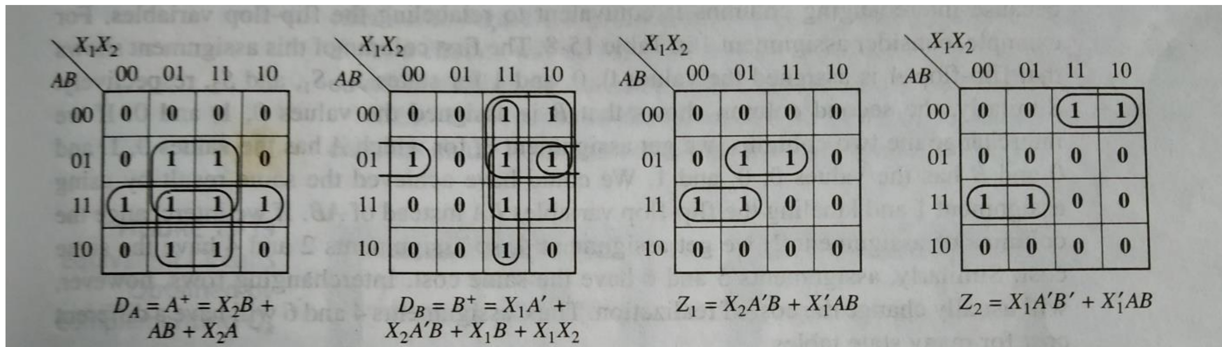
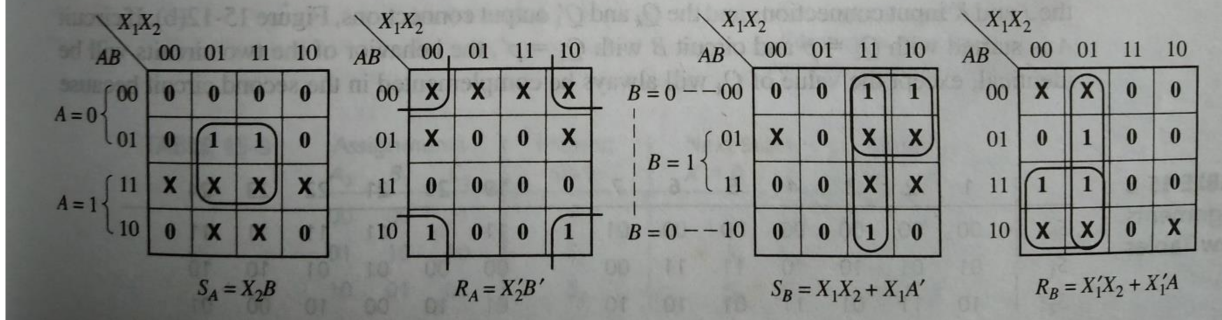


FIGURE 15-11 Derivation of S-R Equations for Table 15-7

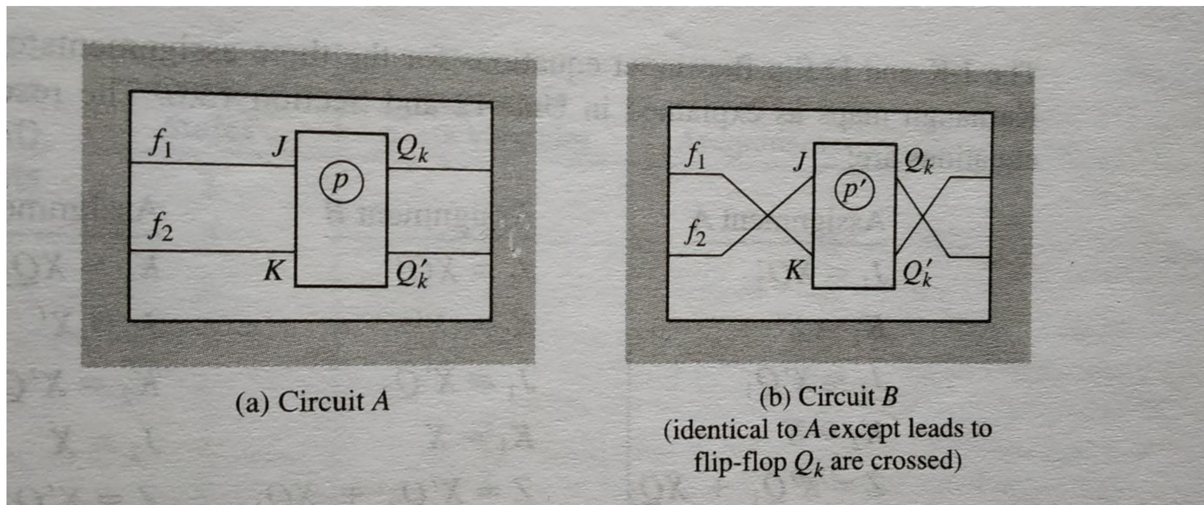
Equivalent State Assignments

After the number of states in a state table has been reduced, the next step in realizing the table is to assign flip-flop states to correspond to the states in the table. The trail-and-error method described next is useful for only a small number of states. If the number of states is small, it may be feasible to try all possible state assignments, evaluate the cost of the realization for each assignment, and choose the assignment with low cost.

If symmetrical flip-flops such as T, J-K, S-R are used, complementing one or more columns of the state assignment will have no effect on the cost of realization. Consider a J-K flip-flip imbedded in a circuit. Leave the circuit unchanged and interchange the J and K input connections.



	1	2	3	4	5	6	7		19	20	21	22	23	24
S_0	00	00	00	00	00	00	01	. . .	11	11	11	11	11	11
S_1	01	01	10	10	11	11	00		00	00	01	01	10	10
S_2	10	11	01	11	01	10	10		01	10	00	10	00	01

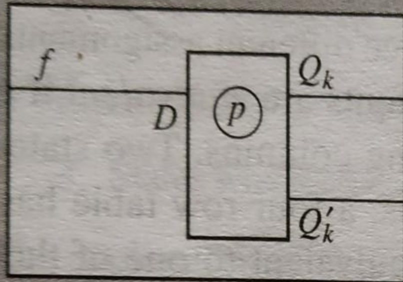


(a) Circuit A

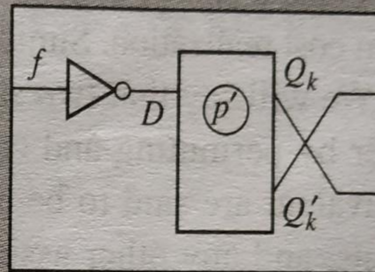
(b) Circuit B
(identical to A except leads to flip-flop Q_k are crossed)

If unsymmetrical flip-flops are used such as D flip-flop , it is still true that permuting columns in the state assignment will not affect the cost; however complementing a column may require adding an inverter to the circuit .

If different types of gates are available the circuit can generally be redesigned to eliminate the inverter and use the same number of gates as the original.



(a) Circuit A



(b) Circuit B
(identical to A except for connections to flip-flop Q_k)

Assignments A_3	B_3	C_3	Present State	Next State $X = 0$	1	Output 0	1
00	00	11	S_1	S_1	S_3	0	0
01	10	10	S_2	S_2	S_1	0	1
10	01	01	S_3	S_2	S_3	1	0

The J-K and D flip-flops input equations for the three assignments can be derived using Karnaugh maps.

Assignment A	Assignment B	Assignment C
$J_1 = XQ_2'$	$J_2 = XQ_1'$	$K_1 = XQ_2$
$K_1 = X'$	$K_2 = X'$	$J_1 = X'$
$J_2 = X'Q_1$	$J_1 = X'Q_2$	$K_2 = X'Q_1'$
$K_2 = X$	$K_1 = X$	$J_2 = X$
$Z = X'Q_1 + XQ_2$	$Z = X'Q_2 + XQ_1$	$Z = X'Q_1' + XQ_2'$
$D_1 = XQ_2'$	$D_2 = XQ_1'$	$D_1 = X' + Q_2'$
$D_2 = X'(Q_1 + Q_2)$	$D_1 = X'(Q_2 + Q_1)$	$D_2 = X + Q_1Q_2$

We will say that two state assignments are equivalent if one can be derived from the other by permuting and complementing columns.Two state assignments which are not equivalent are said to be distinct. Hand solution is feasible for two, three,or four states ; computer solution is feasible for five through eight states; but more than nine states it is not practically to try all assignments even if high-speed computer is used.

Number of States	Minimum Number of State Variables	Number of Distinct Assignments
2	1	1
3	2	3
4	2	3
5	3	140
6	3	420
7	3	840
8	3	840
9	4	10,810,800
⋮	⋮	⋮
16	4	$\approx 5.5 \times 10^{10}$

IMPORTANT TERMS:

Terminal state:

A terminal state is a state with no incoming arcs which start from other states and terminate on it.

Strongly connected machine:

A sequential machine M is said to be strongly connected, if for every pair of states s_i, sj of the sequential machine,there exists an input sequence which takes the machine M from s_i to sj.

Redundant states:

Redundant states are states whose functions can be accomplished by other states.

Equivalent states:

Two states are said to be equivalent if for every possible set of inputs they generate exactly the same output and the same next state.

When equivalent states are there, one of them can be retained and all others can be removed without altering the input-output relationship because they are redundant. This results in reduction of states which in turn reduces the number of required flip-flops and logic gates reducing the cost of final circuit.

UNIT-VI
Digital Design using HDL's

Objective:

- To give a model of combinational and sequential circuits using HDL's.

Syllabus:

Verilog for combinational circuits- conditional operator, if-else statement, case statement, for loop; using storage elements with CAD tools-using Verilog constructs for storage elements, blocking and non-blocking assignments, non-blocking assignments for combinational circuits, flip-flop with clear capability, using Verilog constructs for registers and counters.

Outcome:

Students will be able to

- Develop digital circuits using HDL
- Differentiate blocking and non-blocking assignments in Verilog.

6.1 Verilog for Combinational circuits:

Rather than using logic gates or logic expressions, combinational circuits can be specified in terms of their behavior. To describe the building blocks efficiently, several Verilog constructs have been used. In many cases a given circuit can be described in various ways, using different constructs.

A circuit can be described using if-else statement can also described using a case statement or perhaps a for loop. In general there is no restrict rules that dictate when one style should be preferred over another.

Various constructs which are useful to design combinational circuits are discussed in following sections.

6.1.1 Conditional Operator:

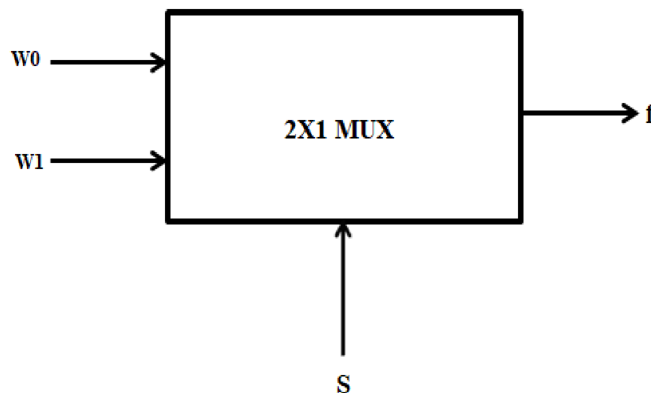
In a logic circuit it is often necessary to choose between several possible signals or values based on the state of some condition.

Example: Multiplexer

In Multiplexer the output is equal to the data input signal chosen by the valuation of the select inputs. For simple implementation of such choices Verilog provides a *conditional* operator (? :) which assigns one of two values depending on a conditional expression. It involves three operands used in the syntax.

conditional_expression ? true_expression : false_expression

Example: 2X1 Multiplexer



The 2X1 Multiplexer has the inputs w0,w1 and s, and the output f. The signal s is used for the selection. The output f is equal to w1 if the select input s has the value 1; otherwise f is equal to w0.

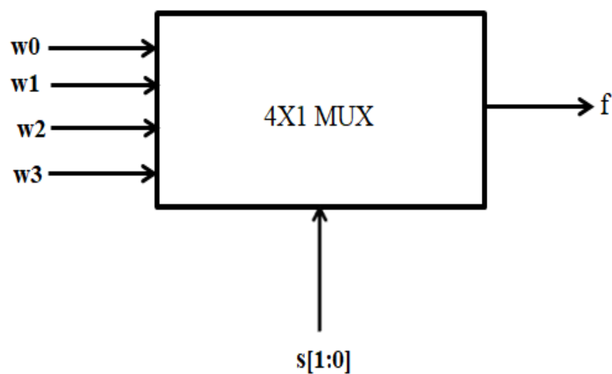
The following module shows 2X1 multiplexer code using conditional operator in an assignment statement.

```
module mux2x1 (w0,w1,s,f);
input w0,w1,s;
output f;
assign f = s ? w1 : w0;
endmodule
```

The conditional operator can be used in always block. The following module shows 2X1 multiplexer code using conditional operator in always block.

```
module mux2x1 (w0,w1,s,f);
input w0,w1,s;
output  reg f;
always @( w0,w1,s)
   f = s ? w1 : w0;
endmodule
```

Example: 4X1 Multiplexer



s[1]	s[0]	f
0	0	w0
0	1	w1
1	0	w2
1	1	w3

The 4X1 Multiplexer has 2 select line s1 and s0, which are represented by the two-bit vector S.

The first conditional expression tests the value of bit s1. If s1=1, the s0 is tested and f is set to w3 if s0=1 and f is set to w2 if s0=0.This corresponds to third and fourth rows of the truth table.

Similarly if s1=0 the conditional operator on the right chooses f=w1 if s0=1 and f=w0 if s0=0, thus realizing the first two rows of the truth table.

```
module mux4x1 ( w0,w1,w2,w3,S,f);
input w0,w1,w2,w3;
input [1:0] S;
output f;
assign f = S[1] ? (S[0] ? w3 :w2) : (S[0] ? w1 : w0);
endmodule
```

6.1.2 The if-else Statement:

Syntax: **if** (conditional_expression) statement;
else statement;

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

2X1 multiplexer code using if-else:

```
module mux2x1 (w0,w1,s,f);
input w0,w1,s;
output  reg f;
always @( w0,w1,s)
     if (s == 0)
         f=w0;
    else
        f=w1;
endmodule
```

4X1 multiplexer code using if-else:

```
module mux4x1 ( w0,w1,w2,w3,S,f);
input w0,w1,w2,w3;
input [1:0] S;
output reg f;
always @ (*)
 if (s == 2’b00)
    f = w0;
else if( s == 2’b01)
    f=w1;
else if (s == 2’b10)
    f=w2;
else
   f = w3;
endmodule
```

6.1.3 The case statement

The **if-else** statement provides the means for choosing an alternative based on the value of an expression. Instead, it is often possible to use the Verilog **case** statement which is defined as

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

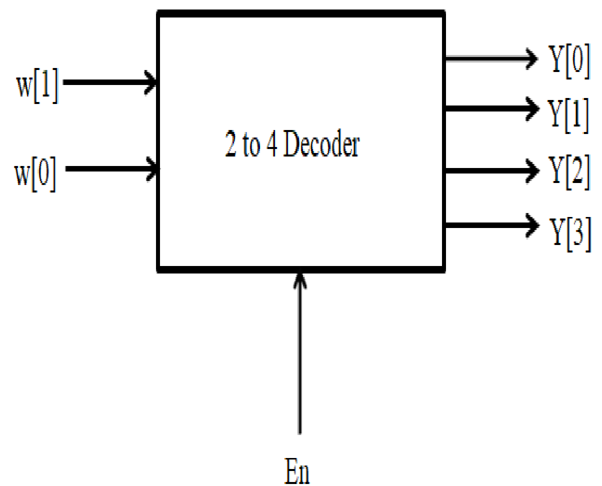
The value of the controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included. Otherwise, the Verilog compiler will synthesize memory elements to deal with the unspecified possibilities.

Example 1 : 4X1 multiplexer code using case

```
module mux4x1 ( w, S, f);
input [3:0]w;
input [1:0] S;
output reg f;
  always @ (*)
     case (s)
        0: f = w[0];
        1: f = w[1];
        2 : f = w  2];
        3 : f = w[3];
         default: f=1'b0;
   endcase
    endmodule
```

Example 2 : 2 to 4 decoder

```
 module dec2to4 (W, En, Y);
input [1:0]W;
input En;
output reg [0:3] Y;
always @(W, En)
case ({En,W})
3'b100: Y = 4'b1000;
3'b101: Y = 4'b0100;
3'b110: Y = 4'b0010;
3'b111: Y = 4'b0001;
default: Y = 4'b0000;
endcase
endmodule
```



6.1.3.1 casex and casez statements

In the **case** statement it is possible to use the logic values 0, 1, *z*, and *x* in the **case** alternatives. A bit-by-bit comparison is used to determine the match between the expression and one of the alternatives.

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

Example : 4X2 Priority encoder

```
module priority (W, Y);
  input [3:0]W;
  output reg [1:0] Y;
  always @(W)
        begin
           casex (W)
           4'b1xxx: Y = 3;
           4'b01xx: Y = 2;
           4'b001x: Y = 1;
           4'b0001: Y = 0;
           default: begin
           Y = 2'bx;
          endcase
       end
endmodule
```

// The first alternative specifies that the output is set to $y1y0 = 3$ if the input $w3$ is 1.
// This assignment does not depend on the values of inputs $w2$, $w1$, or $w0$; hence their values do not matter.
// The other alternatives in the **casex** // statement are evaluated only if $w3 = 0$.
// The second alternative states that if $w2$ is 1, then $y1y0 = 2$.
// If $w2 = 0$, then the next alternative results in $y1y0 = 1$ if $w1 = 1$.
// If $w3 = w2 = w1 = 0$ and $w0 = 1$, then the fourth alternative results in $y1y0 = 0$.

6.1.4 The for loop

If the structure of a desired circuit exhibits a certain regularity, it may be convenient to define the circuit using a **for** loop. The **for** loop has the syntax

for (initial_index; terminal_index; increment) statement;

A loop control variable, which has to be of type **integer**, is set to the value given as the initial index. It is used in the statement or a block of statements delineated by **begin** and **end** keywords. After each iteration, the control variable is changed as defined in the increment. The iterations end after the control variable has reached the terminal index.

Unlike **for** loops in high-level programming languages, the Verilog **for** loop does not specify changes that take place in time through successive loop iterations. Instead, during each iteration it specifies a different sub circuit.

Example 1: 2 to 4 decoder

```
module dec2to4 (W, En, Y);
input [1:0]W;
input En;
output reg [0:3] Y;
integer k;
   always @(W, En)
       for (k = 0; k< = 3; k = k+1)
           if ((W == k) && (En == 1))
                     Y[k] = 1;
          else
                    Y[k] = 0;
Endmodule
```

// Similarly, the other three iterations set the values of $y1$, $y2$, and $y3$ according to the values of *W* and *En*.

// if one or more of the four inputs $w3, \ldots, w0$ is equal to 1, the **for** loop will set
the valuation of $y1y0$ to match the index of the highest priority input that has the value 1.
// Note that each successive iteration through the loop corresponds to a higher priority. // Verilog semantics specify that a signal that receives multiple assignments in an **always** block retains the last assignment.
// Thus the iteration that corresponds to the highest priority input that is equal to 1 will override any setting of *Y* established during the previous iterations.

Example 2: 4X2 priority encoder

```
module priority (W, Y);
input [3:0]W;
output reg [1:0] Y;
integer k;
         always @(W)
           begin
             Y = 2'bx;
                  for (k = 0; k < 4; k = k+1)
                           if (W[k])
                           Y= k;

           end
endmodule
```

// The effect of the loop is to repeat the **if-else** statement four times, for $k = 0, \ldots, 3$.
// The first loop iteration sets $y0 = 1$ if $W = 0$ and $En = 1$.

6.2 Using Storage elements with CAD Tools

Circuits with storage elements can be designed using either schematic capture or Verilog code.

6.2.1 Using Verilog Constructs for Storage Elements

A simple way of specifying a storage element is by using the **if-else** statement to describe the desired behavior responding to changes in the levels of data and clock inputs.

Consider the always block

```
always @(Control, B)
   if (Control)
      A= B;
```

where *A* is a variable of reg type. This code specifies that the value of *A* should be made equal to the value of *B* when *Control* = 1. But the statement does not indicate an action that should occur when *Control* = 0. In the absence of an assigned value, the Verilog compiler assumes that the value of *A* caused by the **if** statement must be maintained when *Control* is not equal to 1. This notion of *implied memory* is both of these signals can cause a change in the value of the Q output. realized by instantiating a latch in the circuit.

Example 1: Gated D latch

The following module named *D_latch*, which has the inputs *D* and *Clk* and the output Q. The **if** clause defines that the output must take the value of *D* when *Clk* = 1. Since no **else** clause is given, a latch will be synthesized to maintain the value of Q when *Clk* = 0. Therefore, the code describes a gated D latch. The sensitivity list includes *Clk* and *D* because both of these signals can cause a change in the value of the Q output.

```
module D_latch (D, Clk, Q);
input D, Clk;
output reg Q;
  always @ (D, Clk)
     if (Clk)
        Q = D;
endmodule
```

Example 2: D Flip-Flop

The following module named *flip flop*, which is a positive- edge-triggered D flip-flop. The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output. The keyword posedge specifies that a change may occur only on the positive edge of *Clock*. At this time the output Q is set to the value of the input *D*. Since posedge appears in the sensitivity list, Q will be implemented as the output of a flip-flop.

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

6.2.2 Blocking and Non-Blocking Assignments

Two types of procedural assignment statements:

a) Blocking (denoted by "=")

b) Non-blocking (denoted by "<=")

6.2.2.1: Blocking Assignments

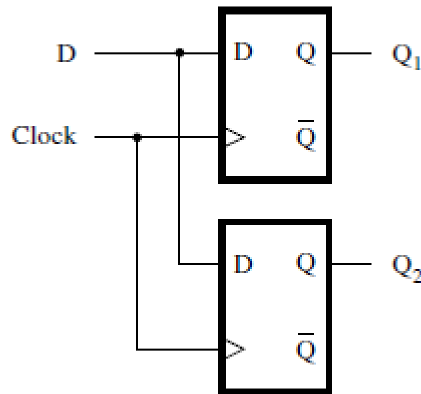
Blocking assignment statements are executed in the order they are specified in a procedural block. The target of an assignments gets updated before the next sequential statement in the procedural block is executed. They do not block execution of statements in other procedural blocks. This is the recommended style for modeling combinational logic.

Example:

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

In the above code the always block is sensitive to the positive clock edge, both Q1 and Q2 will be implemented as the outputs of D flip-flops. However, because blocking assignments are

involved, these two flip-flops will not be connected in cascade The first statement Q1 = D; sets Q1 to the value of *D*. This new value is used in evaluating the subsequent statement Q2 = Q1; which results in Q2 = Q1 = D. The synthesized circuit has two parallel flip-flops, as illustrated in following figure.



6.2.2.2: Non- Blocking Assignments

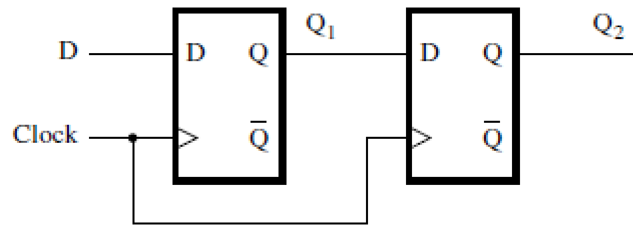
The "<=" operator is used to specify non-blocking assignment. Non-blocking assignment statements allow scheduling of assignments without blocking execution of statements that follow within the procedural block. The assignment to the target gets scheduled for the end of the simulation cycle (at the end of the procedural block). Statements subsequent to the instruction under consideration are not blocked by the assignment. These assignments allow concurrent procedural assignment, suitable for sequential logic.

Example:

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

The variables Q1 and Q2 have some value at the start of evaluating the **always** block, and then they change to a new value concurrently at the end of the **always** block. This code generates a

cascaded connection between flip-flops, which implements the shift register as shown in following figure.



6.3 Non-blocking assignments for combinational circuits

Non-blocking assignments can be used in most situations, but when subsequent assignments in an always block depend on the results of previous assignments, the non-blocking assignments can generate nonsensical circuits.

Example: Assume that we have a three-bit vector $A = a_2a_1a_0$, and we wish to generate a combinational function f that is equal to 1 when there are two adjacent bits in A that have the value 1.

With blocking assignments the function f is specified as

```
always @(A)
begin
f = A[1] & A[0];
f = f | (A[2] & A[1]);
end
```

These statements produce the desired logic function, which is $f = a_1a_0 + a_2a_1$.

With Non- blocking assignments the function f is specified as

```
f <= A[1] & A[0];
f <= f | (A[2] & A[1]);
```

Here f has an unspecified initial value when we enter the **always** block. The first statement assigns $f = a_1a_0$, but this result is not visible to the second statement. It still sees the original unspecified value of f. The second assignment overrides (deletes!) the first assignment and produces the logic function $f = f + a_2a_1$. This expression does not correspond to a combinational circuit, because it represents an AND-OR circuit in which the OR-gate is fed back to itself.

So, It is best to use blocking assignments when describing combinational circuits, so as to avoid accidentally creating a sequential circuit.

6.4: Flip-flops with clear Capability

Reset (clear) is a signal that is used to initialize the hardware, as the design does not have a way to do self-initialization. That means, reset forces the design to a known state. In simulation, usually it is activated at the beginning, but in real hardware, reset is usually activated to power up the circuits.

By using a particular sensitivity list and a specific style of **if-else** statement, it is possible to include clear (or preset) signals on flip-flops.

There are two types of resets used in hardware designs. They are synchronous and asynchronous resets.

6.4.1 Synchronous Reset

Synchronous reset means reset is sampled with respect to clock. In other words, when reset is enabled, it will not be effective till the next active clock edge.

Example:

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

In the above code, the reset signal is acted upon only when a positive clock edge arrives.

6.4.2: Asynchronous Reset

In asynchronous reset, reset is sampled independent of clock. That means, when reset is enabled it will be effective immediately and will not check or wait for the clock edges.

Example:

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

When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0. Note that the sensitivity list specifies the negative edge of *Resetn* as an event trigger along with the positive edge of the clock. We cannot omit the keyword **negedge** because the sensitivity list cannot have both edge-triggered and level sensitive signals.

6.5 Using Verilog Constructs for Registers and Counters

6.5.1: n-Bit Register with Asynchronous Clear:

Registers of different sizes are often needed in logic circuits, it is advantageous to define a register module for which the number of flip-flops can be easily changed. n-bit Register code can be written with parameter construct

Parameter: Verilog allows constants to be defined in a module by the keyword **parameter.** Parameters cannot be used as variables. Parameter values for each module instance can be overridden individually at compile time. This allows the module instances to be customized. Use of parameters make the module definition flexible. Module behavior can be altered simply by changing the value of a parameter.

Code for n-bit register:

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

The parameter *n* specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size.

6.5.2: Four Bit Shift Register:

A four bit shift register can be written using hierarchical code that uses four D flip-flops. Instead of using sub circuits, the shift register can also be written using behavioral style.

In the following behavioral code, all actions take place at the positive edge of the clock. If $L = 1$, the register is loaded in parallel with the four bits of input *R*. If $L = 0$, the contents of the register are shifted to the right and the value of the input *w* is loaded into the most-significant bit Q3.

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

6.5.3: N- Bit Shift Register:

The following code shows the code that can be used to represent shift registers of any size. The parameter *n*, which has the default value 16, sets the number of flip-flops.

```
module shiftn (R, L, w, Clock, Q);
parameter n = 16;
input [n –1:0] R;
input L, w, Clock;
output reg [n –1:0] Q;
integer k;
always @(posedge Clock)
if (L)
Q <= R;
else
begin
```

```
for (k = 0; k < n –1; k = k+1)
Q[k] < = Q[k+1];
Q[n –1] <= w;
end
endmodule
```

6.5.4: Up-Counter

A four-bit up-counter with a reset input, *Resetn*, and an enable input, *E*. The outputs of the flip-flops in the counter are represented by the vector named Q. The **if** statement specifies an asynchronous reset of the counter if *Resetn* = 0. The **else if** clause specifies that if *E* = 1 the count is incremented on the positive clock edge.

```
module upcount (Resetn, Clock, E, Q);
input Resetn, Clock, E;
output reg [3:0] Q;
always @(negedge Resetn, posedge Clock)
if (!Resetn)
Q <= 0;
else if (E)
Q <= Q + 1;
Endmodule
```

6.5.5: Up-Counter with parallel load

The following code defines an up-counter that has a parallel-load input in addition to a reset input. The parallel data is provided as the input vector *R*. The first **if** statement provides the same asynchronous reset. The **else if** clause specifies that if *L* = 1 the flip-flops in the counter are loaded in parallel from the *R* inputs on the positive clock edge. If *L* = 0, the count is incremented, under control of the enable input *E*.

```
module upcount (R, Resetn, Clock, E, L, Q);
input [3:0] R;
input Resetn, Clock, E, L;
output reg [3:0] Q;
always @(negedge Resetn, posedge Clock)
if (!Resetn)
Q <= 0;
else if (L)
Q <= R;
else if (E)
Q <= Q + 1;
Endmodule
```

6.5.6: Down Counter with parallel load

The following figure shows the code for a down-counter named *downcount*. A down-counter is normally used by loading it with some starting count and then decrementing its contents. The starting count is represented in the code by the vector *R*. On the positive clock edge, if $L = 1$ the counter is loaded with the input *R*, and if $L = 0$ the count is decremented, under control of the enable input *E*.

```
module downcount (R, Clock, E, L, Q);
parameter n = 8;
input [n –1:0] R;
input Clock, L, E;
output reg [n –1:0] Q;
always @(posedge Clock)
if (L)
Q <= R;
else if (E)
Q <= Q – 1
Endmodule
```

6.5.7: Up/Down Counter

Verilog code for an up/down counter is given in following Figure. This module combines the capabilities of up and down counters. It includes a control signal *up_down* that governs the direction of counting.

```
module updowncount (R, Clock, L, E, up_down, Q);
parameter n = 8;
input [n– 1:0] R;
input Clock, L, E, up_down;
output reg [n– 1:0] Q;
always @(posedge Clock)
if (L)
Q <= R;
else if (E)
Q < = Q + (up_down ? 1 : –1);
Endmodule
```

Assignment-Cum-Tutorial Questions
Section-A

1. Which of the following statement is true for Verilog modules?
   a. A module can contain definitions of other modules.
   b. When a module X is called multiple numbers of times from some other module, only one copy of module X is included in the hardware after synthesis.
   c. More than one module can be instantiated within another module.
   d. None of the above
2. What does the statement “assign f = (a & b) | (a ^ b)” signify?
   a. In module declaration f is declared as reg.
   b. A dataflow description of the function *f*.

c. A structural description of the function *f*.

d. All of the above

3. Which of the following is not true for register type variables?

a. It will always map to a hardware register after synthesis.

b. It can be used in an expression on the RHS of an "assign" statement.

c. Once a value is assigned, it will hold the value.

d. None of the above.

4. If "clk" and "clear" are two inputs of a module that defines a register, which of the following event expressions must be used if we want to implement asynchronous clear (assuming "clear" is active low)?

a. always @(posedge clk)

b. always @(negedge clear)

c. always @(posedge clk or negedge clear)

d. None of the above

5. What will the following code segment do?

```
always @(posedge clock)
begin
red = blue;
blue = red;
end
```

a. Exchange the values of the variables "red" and "blue".

b. Both variables will get the value previously stored in "red".

c. Both variables will get the value previously stored in "blue".

d. None of the above.

6. What will the following code segment generate on synthesis, assuming that the four variables y0, y1, y2 and y3 map into four latches / flip-flops?

```
always @(posedge clock)
begin
y3 = in;
y2 = y3;
y1 = y2;
y0 = y1;
```

end

a. A 4-bit shift register. b. A 4-bit parallel-in parallel-out register.

c. Four D flip-flops all fed with the data "in". d. None of the above.

7. What will the following code segment generate on synthesis?

```
always @(posedge clock)
begin
y3 <= in;
y2 <= y3;
y1 <= y2;
y0 <= y1;
end
```

a. A 4-bit shift register. b. A 4-bit parallel-in parallel-out register.

c. Four D flip-flops all fed with the data "in". d. None of the above.

8. An event is triggered by symbol

a. => b . ---> c . @ d. None

9. Which of the following is true about the always block?

a. There can be exactly one always block in a design.

b. There can be exactly one always block in a module.

c. Execution of an always block occurs exactly once per simulation run.

d. An always block may be used to generate a periodic signal.

10. For describing circuits like flip flops ______________ statement is used

a. Always b. Entity c. Component d. Process

11. In non- blocking assignment

a . Evaluates all RHS for current time unit and assign to LHS at current time

b . Evaluates all RHS for current time unit and assign to LHS at the end of time unit

c . Whole statement is done before control passes to next statement

d. None

12. If a variable is not assigned in all possible executions of an always statement then:

a. A don't care is inferred

b. A latch is inferred

c. The variable is set to 0

d. The synthesis process will fail

Section-B

1.Write a verilog code to swap contents of two registers with and without a temporary register?

2. Write a Verilog code that represents a T flip-flop with asynchronous clear input.

3. Differentiate blocking and non blocking assignments with examples.

4. Why non-blocking assignments are not preferable in combinational circuits.

5. Using casex statement, write Verilog code for an 8-to-3 priority encoder.

6. What is the difference between synchronous Reset and Asynchronous reset.

7. Write Verilog code for 3-to-8 decoder using for loop.

Section-C

1. Consider the Verilog code. What type of circuit does the code represent?

```
module example (W, En, y0, y1, y2, y3);
input [1:0]W;
input En;
output reg y0, y1, y2, y3;
always @(W, En)
begin
y0 = 0;
y1 = 0;
y2 = 0;
y3 = 0;
if (En)
if (W == 0) y0 = 1;
else if (W == 1) y1 = 1;
else if (W == 2) y2 = 1;
else y3 = 1;
end
endmodule
```

2. Consider the following Verilog module.

```
module guess (data, cond, result);
input [7:0] data;
input [1:0] cond;
output reg result;
always @(data)
begin
if (cond == 2'b00) result = |data;
else result = ~^data;
end
endmodule
```

Which of the following are true when the module is synthesized?

a. A combinational circuit will be generated.

b. A sequential circuit with a storage element for result will be generated.

c. The synthesize system will generate a wire for result.

d. None of the above.

3. Design four-bit Synchronous counter with parallel load. Use T flip-flops.

4. An SR flip-flop is a flip-flop that has set and reset inputs like a gated SR latch. Show how an SR flip-flop can be constructed using a D flip-flop and other logic gates.

5. The circuit in Figure looks like a counter. What is the counting sequence of this circuit?