MODULE-1 PRINCIPLES OF COMBINATION LOGIC

Definition of combinational logic, Canonical forms, Generation of switching equations from truth tables, Karnaugh maps-2,3,4variables, Quine-McCluskey Minimization Technique, Quine-McCluskey using don't care terms.

1.1 COMBINATIONAL LOGIC

Introduction Logic circuit may be classified into two categories

1. Combinational logic circuits

2. Sequential logic circuits

A combinational logic circuit contains logic gates only but does not contain storage elements. A sequential logic circuit contains storage elements in addition to logic gates. When logic gates are connected together to give a specified output for certain specified combination of input variables, with no storage involved, the resulting network is known as combinational logic circuit.

In combinational logic circuit the output level is at all times dependent on the combination of input level. The block diagram is shown

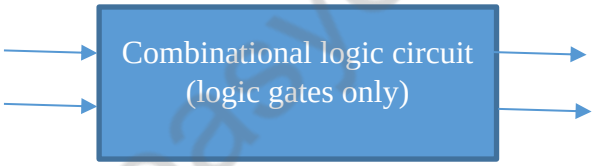


Fig : Block diagram of Combinational circuit

The combinational logic circuit with memory elements(s) is called sequential logic circuit. It consists of a combinational circuit to which memory elements are connected to form a feedback path. The memory elements are devices, capable of storing binary information within them. The block diagram is shown.

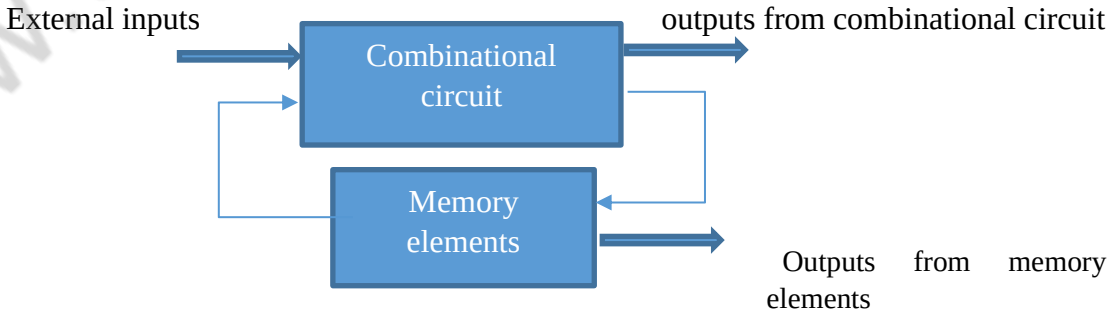


Fig : Block diagram of Sequential circuit

By block diagram, it can be said that the output(s) of sequential logic circuit is (are) dependent not only on external input(s) but also on the present state of the memory element(s). The next state of the memory element(s) is also dependent on external input and the present state. Applications Logic gates find wide applications in Calculators and computers, digital measuring techniques, digital processing of communications, musical instruments, games and domestic appliances etc, for decision making in automatic control of machines and various industrial processes and for building more complex devices such as binary counters etc.

Laws and Rules of Boolean Algebra

- Laws of Boolean Algebra

The basic laws of Boolean algebra-the commutative laws for addition and multiplication, the associative laws for addition and multiplication, and the distributive law-are the same as in ordinary algebra.

The commutative law $A+B = B+A$

$A.B = B.A$

The associative law $A + (B + C) = (A + B) + C$

$A(BC) = (AB)C$

Distributive Law $A(B + C) = AB + AC$

- Rules of Boolean Algebra

1. $A + 0 = A$	**7.** $A \cdot A = A$
2. $A + 1 = 1$	**8.** $A \cdot \overline{A} = 0$
3. $A \cdot 0 = 0$	**9.** $\overline{\overline{A}} = A$
4. $A \cdot 1 = A$	**10.** $A + AB = A$
5. $A + A = A$	**11.** $A + \overline{A}B = A + B$
6. $A + \overline{A} = 1$	**12.** $(A + B)(A + C) = A + BC$

A, B, or *C* can represent a single variable or a combination of variables.

(Referring to the table above)

Proof Rule 10: A + AB = A

This rule can be proved by applying the distributive law, rule 2, and rule 4 as follows:

A + AB = A (1 + B) Factoring (distributive law)

= A. l Rule 2: (1 + B) = 1

= A Rule 4: A. 1 = A

Rule 11.

A + AB = A + B

This rule can be proved as follows:

A + AB = (A + AB) + AB Rule 10: A = A + AB

= (AA + AB) + AB Rule 7: A = AA
=AA +AB +AA +AB Rule 8: adding AA = 0
= (A + A) (A + B) Factoring
= 1. (A + B) Rule 6: A + A = 1
=A + B Rule 4: drop the 1

Rule 12. (A + B) (A + C) = A + BC
This rule can be proved as follows:
(A + B) (A + C) = AA + AC + AB + BC Distributive law
= A + AC + AB + BC Rule 7: AA = A
= A (1 + C) + AB + BC Rule 2: 1 + C = 1
= A. 1 + AB + BC Factoring (distributive law)
= A (1 + B) + BC Rule 2: 1 + B = 1
= A. 1 + BC Rule 4: A. 1 = A
= A + BC

DEMORGAN'S THEOREMS

The complement of a product of variables is equal to the sum of the individual complements of the variables.

$$\overline{X.Y} = \bar{X} + \bar{Y}$$

The complement of a sum of variables is equal to the product of the individual complements of the variables.

$$\overline{X + Y} = \bar{X}.\bar{Y}$$

1.2. CANONICAL FORMS AND NORMAL FORMS

We will get four Boolean product terms by combining two variables x and y with logical AND operation. These Boolean product terms are called as **min terms** or **standard product terms**. The min terms are x'y', x'y, xy' and xy.

Similarly, we will get four Boolean sum terms by combining two variables x and y with logical OR operation. These Boolean sum terms are called as **Max terms** or **standard sum terms**. The Max terms are x+y, x+y', x'+y and x'+y'.

The following table shows the representation of min terms and MAX terms for 2 variables.

x	y	Min terms	Max terms
0	0	$m_0=x'y'$	$M_0=x+y$
0	1	$m_1=x'y$	$M_1=x+y'$
1	0	$m_2=xy'$	$M_2=x'+y$

1	1	$m_3=xy$	$M_3=x'+y'$

If the binary variable is '0', then it is represented as complement of variable in min term and as the variable itself in Max term. Similarly, if the binary variable is '1', then it is represented as complement of variable in Max term and as the variable itself in min term.

From the above table, we can easily notice that min terms and Max terms are complement of each other. If there are 'n' Boolean variables, then there will be 2^n min terms and 2^n Max terms.

1.3 GENERATION OF SWITCHING EQUATION FROM TRUTH TABLE

Canonical SoP and PoS forms

A truth table consists of a set of inputs and output(s). If there are 'n' input variables, then there will be 2^n possible combinations with zeros and ones. So the value of each output variable depends on the combination of input variables. So, each output variable will have '1' for some combination of input variables and '0' for some other combination of input variables.

Therefore, we can express each output variable in following two ways.

- Canonical SoP form
- Canonical PoS form

Canonical SoP form (Minterm canonical form)

Canonical SoP form means Canonical Sum of Products form. In this form, each product term contains all literals. So, these product terms are nothing but the min terms. Hence, canonical SoP form is also called as **sum of min terms** form.

First, identify the min terms for which, the output variable is one and then do the logical OR of those min terms in order to get the Boolean expression (function) corresponding to that output variable. This Boolean function will be in the form of sum of min terms.

Follow the same procedure for other output variables also, if there is more than one output variable.

Example:

Consider the following **truth table**.

Inputs			**Output**
p	**q**	**r**	**F**
0	0	0	0

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

Here, the output (f) is '1' for four combinations of inputs. The corresponding min terms are p'qr, pq'r, pqr', pqr. By doing logical OR of these four min terms, we will get the Boolean function of output (f).

Therefore, the Boolean function of output is, f=p'qr + pq'r + pqr' + pqr. This is the **canonical SoP form** of output, f. We can also represent this function in following two notations.

f=m3+m5+m6+m7

f=∑m(3,5,6,7)

In one equation, we represented the function as sum of respective min terms. In other equation, we used the symbol for summation of those min terms.

Canonical PoS form (Maxterm canonical form)

Canonical PoS form means Canonical Product of Sums form. In this form, each sum term contains all literals. So, these sum terms are nothing but the Max terms. Hence, canonical PoS form is also called as **product of Max terms** form.

First, identify the Max terms for which, the output variable is zero and then do the logical AND of those Max terms in order to get the Boolean expression (function) corresponding to that output variable. This Boolean function will be in the form of product of Max terms.

Follow the same procedure for other output variables also, if there is more than one output variable.

Example

Consider the same truth table of previous example. Here, the output (f) is '0' for four combinations of inputs. The corresponding Max terms are p+q+r, p+q+r', p+q'+r, p'+q+r. By doing logical AND of these four Max terms, we will get the Boolean function of output (f).

Therefore, the Boolean function of output is, f=(p+q+r).(p+q+r').(p+q'+r).(p'+q+r). This is the **canonical PoS form** of output, f. We can also represent this function in following two notations.

$$f=M0.M1.M2.M4$$

$$f=\prod M(0,1,2,4)$$

In one equation, we represented the function as product of respective Max terms. In other equation, we used the symbol for multiplication of those Max terms.

The Boolean function, f=(p+q+r).(p+q+r').(p+q'+r).(p'+q+r) is the dual of the Boolean function, f=p'qr + pq'r + pqr' + pqr.

Therefore, both canonical SoP and canonical PoS forms are **Dual** to each other. Functionally, these two forms are same. Based on the requirement, we can use one of these two forms.

Standard SoP and PoS forms

We discussed two canonical forms of representing the Boolean output(s). Similarly, there are two standard forms of representing the Boolean output(s). These are the simplified version of canonical forms.

- Standard SoP form
- Standard PoS form

We will discuss about Logic gates in later chapters. The main **advantage** of standard forms is that the number of inputs applied to logic gates can be minimized. Sometimes, there will be reduction in the total number of logic gates required.

Standard SoP form

Standard SoP form means **Standard Sum of Products** form. In this form, each product term need not contain all literals. So, the product terms may or may not be the min terms. Therefore, the Standard SoP form is the simplified form of canonical SoP form.

We will get Standard SoP form of output variable in two steps.

- Get the canonical SoP form of output variable
- Simplify the above Boolean function, which is in canonical SoP form.

Follow the same procedure for other output variables also, if there is more than one output variable. Sometimes, it may not possible to simplify the canonical SoP form. In that case, both canonical and standard SoP forms are same.

Example

Convert the following Boolean function into Standard SoP form.

$$f=p'qr + pq'r + pqr' + pqr$$

The given Boolean function is in canonical SoP form. Now, we have to simplify this Boolean function in order to get standard SoP form.

Step 1 − Use the **Boolean postulate**, x + x = x. That means, the Logical OR operation with any Boolean variable 'n' times will be equal to the same variable. So, we can write the last term pqr two more times.

$$\Rightarrow f= p'qr + pq'r + pqr' + pqr + pqr + pqr$$

Step 2 − Use **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rdand 6th terms.

$$\Rightarrow f= qr(p' + p) + pr(q'+q) + pq(r'+r)$$

Step 3 − Use **Boolean postulate**, x + x' = 1 for simplifying the terms present in each parenthesis.

$$\Rightarrow f= qr(1) + pr(1) + pq(1)$$

Step 4 − Use **Boolean postulate**, x.1 = x for simplifying above three terms.

$$\Rightarrow f= qr + pr + pq$$

$$\Rightarrow f= pq + qr + pr$$

This is the simplified Boolean function. Therefore, the **standard SoP form**corresponding to given canonical SoP form is **f= pq + qr + pr**

Standard PoS form

Standard PoS form means **Standard Product of Sums** form. In this form, each sum term need not contain all literals. So, the sum terms may or may not be the Max terms. Therefore, the Standard PoS form is the simplified form of canonical PoS form.

We will get Standard PoS form of output variable in two steps.

- Get the canonical PoS form of output variable
- Simplify the above Boolean function, which is in canonical PoS form.

Follow the same procedure for other output variables also, if there is more than one output variable. Sometimes, it may not possible to simplify the canonical PoS form. In that case, both canonical and standard PoS forms are same.

Example

Convert the following Boolean function into Standard PoS form.

$$f=(p+q+r).(p+q+r').(p+q'+r).(p'+q+r)$$

The given Boolean function is in canonical PoS form. Now, we have to simplify this Boolean function in order to get standard PoS form.

Step 1 − Use the **Boolean postulate**, x.x=x. That means, the Logical AND operation with any Boolean variable 'n' times will be equal to the same variable. So, we can write the first term p+q+r two more times.

$$\Rightarrow f=(p+q+r).(p+q+r).(p+q+r).(p+q+r').(p+q'+r).(p'+q+r)$$

Step 2 − Use **Distributive law,** x + (y.z) = (x+y).(x+z) for 1st and 4thparenthesis, 2nd and 5th parenthesis, 3rd and 6th parenthesis.

$$\Rightarrow f=(p+q+rr').(p+r+qq').(q+r+pp')$$

Step 3 − Use **Boolean postulate**, x.x'=0 for simplifying the terms present in each parenthesis.

$$\Rightarrow f=(p+q+0).(p+r+0).(q+r+0)$$

Step 4 − Use **Boolean postulate**, x+0=x for simplifying the terms present in each parenthesis

$$\Rightarrow f=(p+q).(p+r).(q+r)$$

$$\Rightarrow f=(p+q).(q+r).(p+r)$$

This is the simplified Boolean function. Therefore, the **standard PoS form**corresponding to given canonical PoS form is **f=(p+q).(q+r).(p+r)**. This is the **dual** of the Boolean function, f=pq+qr+pr.

Therefore, both Standard SoP and Standard PoS forms are Dual to each other.

1.4. K-MAPS FOR 2 TO 5 VARIABLES

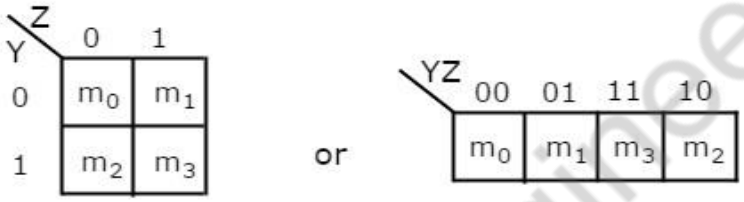
We have simplified the Boolean functions using Boolean postulates and theorems. It is a time-consuming process and we have to re-write the simplified expressions after each step.

To overcome this difficulty, **Karnaugh** introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2^n cells for 'n' variables. The adjacent cells are differed only in single bit position.

K-Map method is most suitable for minimizing Boolean functions of 2 variables to 5 variables. Now, let us discuss about the K-Maps for 2 to 5 variables one by one.

2 Variable K-Map

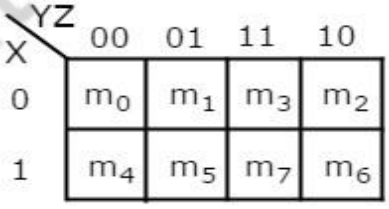
The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.



- There is only one possibility of grouping 4 adjacent min terms.
- The possible combinations of grouping 2 adjacent min terms are {(m_0, m_1), (m_2, m_3), (m_0, m_2) and (m_1, m_3)}.

3 Variable K-Map

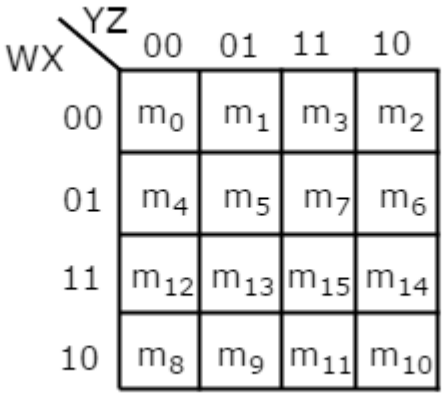
The number of cells in 3 variable K-map is eight, since the number of variables is three. The following figure shows **3 variable K-Map**.



- There is only one possibility of grouping 8 adjacent min terms.
- The possible combinations of grouping 4 adjacent min terms are {(m_0, m_1, m_3, m_2), (m_4, m_5, m_7, m_6), (m_0, m_1, m_4, m_5), (m_1, m_3, m_5, m_7), (m_3, m_2, m_7, m_6) and (m_2, m_0, m_6, m_4)}.
- The possible combinations of grouping 2 adjacent min terms are {(m_0, m_1), (m_1, m_3), (m_3, m_2), (m_2, m_0), (m_4, m_5), (m_5, m_7), (m_7, m_6), (m_6, m_4), (m_0, m_4), (m_1, m_5), (m_3, m_7) and (m_2, m_6)}.
- If x=0, then 3 variable K-map becomes 2 variable K-map.

4 Variable K-Map

The number of cells in 4 variable K-map is sixteen, since the number of variables is four. The following figure shows **4 variable K-Map**.



- There is only one possibility of grouping 16 adjacent min terms.
- Let R_1, R_2, R_3 and R_4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C_1, C_2, C_3 and C_4represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are {(R_1, R_2), (R_2, R_3), (R_3, R_4), (R_4, R_1), (C_1, C_2), (C_2, C_3), (C_3, C_4), (C_4, C_1)}.
- If w=0, then 4 variable K-map becomes 3 variable K-map.

Minimization of Boolean Functions using K-Maps

If we consider the combination of inputs for which the Boolean function is '1', then we will get the Boolean function, which is in **standard sum of products** form after simplifying the K-map.

Similarly, if we consider the combination of inputs for which the Boolean function is '0', then we will get the Boolean function, which is in **standard product of sums** form after simplifying the K-map.

Follow these **rules for simplifying K-maps** in order to get standard sum of products form.

- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as sum of min terms form, then place the ones at respective min term cells in the K-map. If the Boolean function is given as sum of products form, then place the ones in all possible cells of K-map for which the given product terms are valid.

- Check for the possibilities of grouping maximum number of adjacent ones. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.

- Each grouping will give either a literal or one product term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single '1' is not covered with any other groupings but only that grouping covers.

- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

Note 1 − If outputs are not defined for some combination of inputs, then those output values will be represented with **don't care symbol 'x'**. That means, we can consider them as either '0' or '1'. **Note 2** − If don't care terms also present, then place don't cares 'x' in the respective cells of K-map. Consider only the don't cares 'x' that are helpful for grouping maximum number of adjacent ones. In those cases, treat the don't care value as '1'.

1.5. THE TABULATION METHOD (QUINE-MC CLUSKEY ALGORITHM)

For function of five or more variables, it is difficult to be sure that the best selection is made. In such case, the tabulation method can be used to overcome such difficulty. The tabulation method was first formulated by Quine and later improved by McCluskey. It is also known as Quine-McCluskey method.

The Quine–McCluskey algorithm (or the method of prime implicants) is a method used for minimization of boolean functions. It is functionally identical to Karnaugh mapping, but the tabular form makes it more efficient for use in computer algorithms, and it also gives a deterministic way to check that the minimal form of a Boolean function has been reached.

The method involves two steps:

- Finding all prime implicants of the function.
- Use those prime implicants in a prime implicant chart to find the essential prime implicants of the function, as well as other prime implicants that are necessary to cover the function.

Finding prime implicants : Minimizing an arbitrary function:

	A	B	C	D	f
m0	0	0	0	0	0
m1	0	0	0	1	0
m2	0	0	1	0	0
m3	0	0	1	1	0
m4	0	1	0	0	1
m5	0	1	0	1	0
m6	0	1	1	0	0
m7	0	1	1	1	0
m8	1	0	0	0	1
m9	1	0	0	1	x
m10	1	0	1	0	1
m11	1	0	1	1	1
m12	1	1	0	0	1
m13	1	1	0	1	0
m14	1	1	1	0	x
m15	1	1	1	1	1

One can easily form the canonical sum of products expression from this table, simply by summing the minterms (leaving out don't-care terms) where the function evaluates to one:

F(A,B,C,D) = A′BC′D′ + AB′C′D′ + AB′CD′ + AB′CD + ABC′D′ + ABCD

Of course, that's certainly not minimal. So to optimize, all minterms that evaluate to one are first placed in a minterm table. Don't-care terms are also added into this table, so they can be combined with minterms:

Number of 1s	Minterm	Binary Representation
1	m4	0100
	m8	1000
2	m9	1001
	m10	1010
	m12	1100
3	m11	1011
	m14	1110
4	m15	1111

At this point, one can start combining minterms with other minterms. If two terms vary by only a single digit changing, that digit can be replaced with a dash indicating that the digit doesn't matter. Terms that can't be combined any more are marked with a "*". When going from Size 2 to Size 4, treat '-' as a third bit value. Ex: -110 and -100 or -11- can be combined, but not -110 and 011-. (Trick: Match up the '-' first.)

Number of 1s	Minterm	0-Cube	Size 2 Implicants	Size 4 Implicants
1	m4	0100	m(4,12) -100*	m(8,9,10,11) 10--*
	m8	1000	m(8,9) 100-	m(8,10,12,14) 1--0*
2	m9	1001	m(8,10) 10-0	m(10,11,14,15) 1-1-*
	m10	1010	m(8,12) 1-00	
	m12	1100	m(9,11) 10-1	
3	m11	1011	m(10,11) 101-	
	m14	1110	m(10,14) 1-10	
			m(12,14) 11-0	
4	m15	1111	m(11,15) 1-11	
			m(14,15) 111-	

At this point, the terms marked with * can be seen as a solution. That is the solution is

F=AB′+AD′+AC+BC′D′

If the karnaugh map was used, we should have obtain an expression simplier than this.

Prime implicant chart

None of the terms can be combined any further than this, so at this point we construct an essential prime implicant table. Along the side goes the prime implicants that have just been generated, and along the top go the minterms specified earlier. The don't care terms are not placed on top - they are omitted from this section because they are not necessary inputs.

	4	8	10	11	12	15	
m(4,12)	X				X		-100 (BC′D′)
m(8,9,10,11)		X	X	X			10--(AB′)
m(8,10,12,14)		X	X		X		1--0 (AD′)
m(10,11,14,15)			X	X		X	1-1- (AC)

MODULE-2 LOGIC DESIGN WITH MSI COMPONENTS AND PROGRAMMABLE LOGIC DEVICES

Binary Adders and Subtractors, Comparators, Decoders, Encoders, Multiplexers, Programmable Logic Devices (PLD's).

DESIGN OF COMBINATIONAL CIRCUITS

The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram. The procedure involves the following steps:

1. State the given problem completely and exactly
2. Interpret the problem, and determine the available input variables and required output variables.
3. Assign a letter symbol to each input and output variables.
4. Design the truth table, which defines the required relations between inputs and outputs.
5. Obtain the simplified Boolean expression for each output using k-maps.
6. Draw the logic circuit diagram to implement the Boolean expression.

ARITHMETIC CIRCUITS

One essential function of most computers and calculators is the performance of arithmetic operations. The logic gates designed so far can be used to perform arithmetic operations such as addition, subtraction, multiplication and division in electronic calculators and digital instruments. Since these circuits are electronic, they are very fast. Typically an addition operation takes less than 1 μs.

HALF – ADDER

A Logic circuit used for the addition of two one bit numbers is referred to as a half-adder. From the verbal explanation of a half adder, we find that this 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. We assign the symbols A and B to the two inputs and S (for sum) and C (for carry) to the outputs. The truth table for the half-adder is shown below.

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Here the C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum. The logic expression for the sum output can be obtained from the truth table. It can be written as a SOP expression by summing up the input combinations for which the sum is equal to 1.

In the truth table, the sum output is 1 for **A′B and AB′.** Therefore, the expression for the sum is

$$\mathbf{S = A'B + AB' = A \oplus B.}$$

Similarly, the logic expression for carry output can be written as a SOP expression by summing up the input combinations for which the carry is equal to 1. In the truth table, the carry is 1 for AB. Therefore **C = AB** This expression for C cannot be simplified. The sum output corresponds to a logic Ex-OR function while the carry output corresponds to an AND function. So the half-adder circuit can be implemented using Ex-OR and AND gate as shown below.

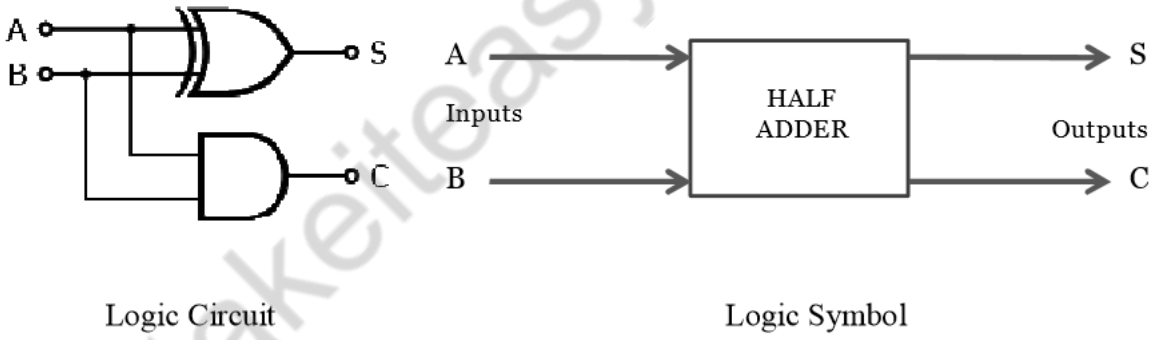


Fig 1: Half Adder Logic circuit

This circuit is called Half-Adder, because it cannot accept a CARRY-IN from previous additions. This is the reason that half – adder circuit can be used for binary addition of lower most bits only. For higher order columns we use a 3-input adder called full-adder

FULL – ADDER

A combinational logic circuit for adding three bits. As seen, a half-adder has only two inputs and there is no provision to add carry coming from the lower bit order when multi bit addition is performed. For this purpose we use a logic circuit that can add three bits, the third bit is the carry

from the lower column. This implies that we need a logic circuit with 3 inputs and 2 outputs. Such a circuit is called a full – adder. The truth table for the full-adder is as shown below.

Inputs			Outputs	
A	B	C_{in}	S	C_{out}
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

As shown there are 8 possible input combinations for the three inputs and for each case the S and Cout values are listed. From the truth table, the logic expression for S can be written by summing up the input combinations for which the sum output is 1 as:

S = A′B′Cin + A′BC′in + AB′C′in + ABCin

= A′(B′Cin + BC′in) + A(B′C′in + BCin)

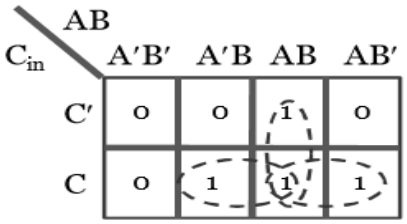
= A′(B⊕Cin)+ A(B⊕Cin)′ Let B⊕Cin = X

Now, S = A′X + AX′ = A ⊕ X Replacing X in the above expression we get

S = A ⊕ B ⊕ Cin

Similarly the logic expression for Cout can be written as

Cout = A′BCin + AB′Cin + ABC′in + ABCin



Cout = BCin + ACin + AB (using the map shown)

From the simplified expressions of S and C the full adder Circuit can be implemented using two 2-input XOR gates, Three 2 –input AND gates and one 3-input OR gate a shown below fig (a). The logic symbol is also shown as fig (b).

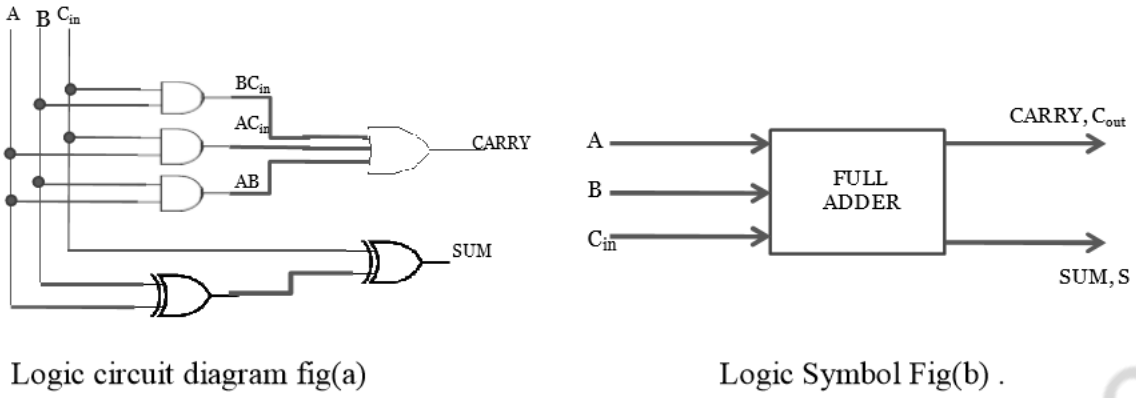


Fig 2: Full Adder Logic Circuit

The logic symbol has two inputs A and B plus a third input Cin called the Carry-in and two outputs SUM and the Carry called Carry out, Cout going to the next higher column.. A full adder can be made by using two half adders and an OR gate as shown below.

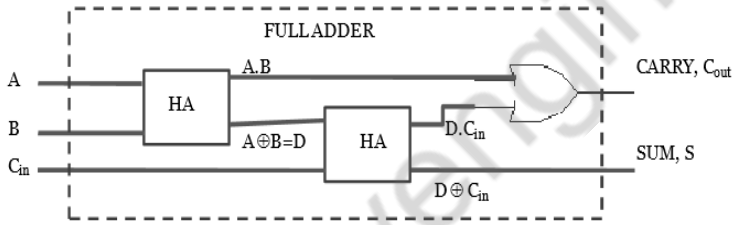
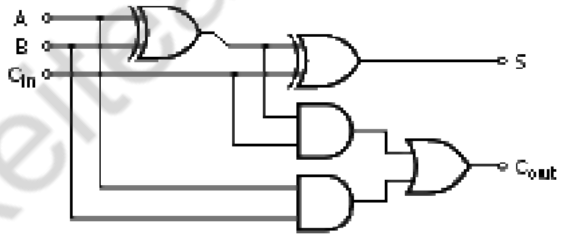


Fig 3: Full adder circuit using two half adders



HALF – SUBTRACTOR

A logic circuit that subtracts Y (subtrahend) from X(minuend), where X and Y are 1-bit numbers, is known as a half-subtractor. It has two inputs X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow), as shown in the block diagram.

The operation of this logic circuit is based on the rules of binary subtraction given in the truth table reproduced on the basis of the subtraction process.

Inputs		Outputs	
X	Y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

The difference output in the third column has the same logic pattern as when X is XORed with Y (same as in the case of sum). Hence an Ex-Or gate can be used to give difference of two bits. The borrow output in the 4th column can be obtained by using a NOT gate and AND gate, as shown in the circuit diagram below.

The logical equations for the difference D and borrow B are given as

D = X′Y + XY′ = X ⊕ Y.

B = X′Y

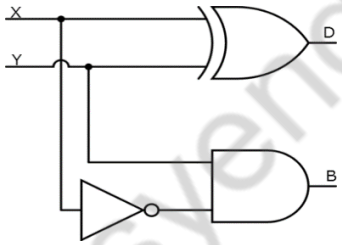
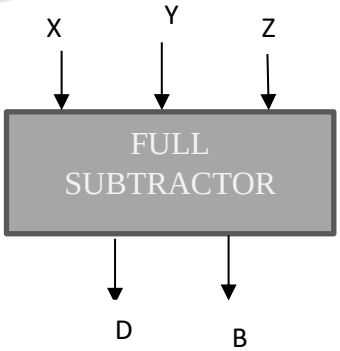


Fig 4: Half Subtractor Logic circuit

FULL – SUBTRACTOR

The full-subtractor is a combinational circuit which is used to perform subtraction of three single bits.



The truth table for the full-subtractor is as shown below.

INPUT			OUTPUT	
X	**Y**	**Z**	**D**	**B**
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

As shown there are 8 possible input combinations for the three inputs and for each case the D and B values are listed. From the truth table, the logic expression for D can be written by summing up the input combinations for which the Difference output is 1 as:

D = X'Y'Z + X'YZ' + XY'Z'+ XYZ

=X' (Y'Z + YZ') + X (Y'Z'+ YZ)

= X'(Y ⊕Z) + X(Y ⊕Z)'

D=X ⊕Y ⊕ Z

B = X'Y'Z + X'YZ' + X'YZ + XYZ

= Z (X'Y + XY') + X'Y (Z' + Z)

B = Z (X⊕ Y) + X'Y

The circuit diagram for full subtractor is constructed from half subtractor and the extension to it , as shown below.

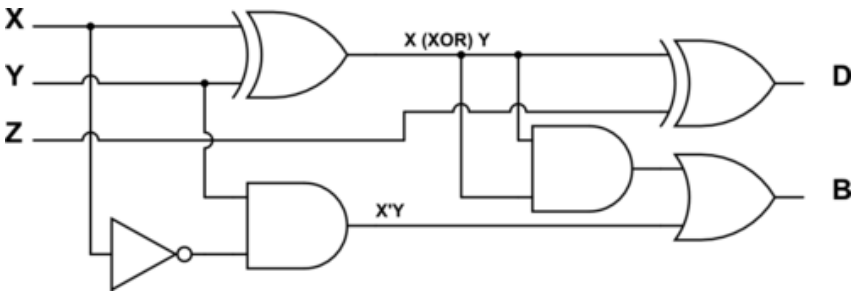


Fig 5: Full Subtractor Logic circuit

4-BIT PARALLEL ADDER

The 4-bit binary adder performs the **addition of two 4-bit numbers**. Let the 4-bit binary numbers, A=A3 A2 A1 A0 and B=B3 B2 B1 B0. We can implement 4-bit binary adder in one of the two following ways.

- Use one Half adder for doing the addition of two Least significant bits and three Full adders for doing the addition of three higher significant bits.
- Use four Full adders for uniformity. Since, initial carry C_0 is zero, the Full adder which is used for adding the least significant bits becomes Half adder.

For the time being, we considered second approach. The **block diagram** of 4-bit binary adder is shown in the following figure.

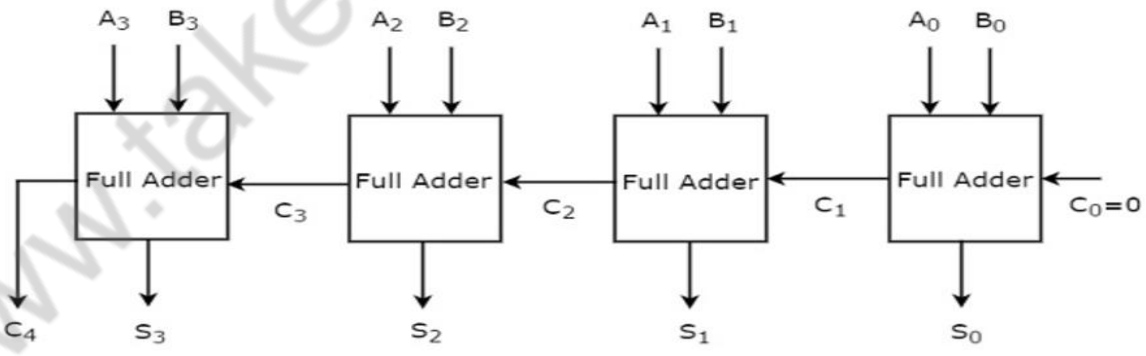


Fig 6: 4-Bit Binary Adder Circuit

Here, the 4 Full adders are cascaded. Each Full adder is getting the respective bits of two parallel inputs A & B. The carry output of one Full adder will be the carry input of subsequent higher order Full adder. This 4-bit binary adder produces the resultant sum having at most 5 bits. So, carry out of last stage Full adder will be the MSB.

In this way, we can implement any higher order binary adder just by cascading the required number of Full adders. This binary adder is also called as **ripple carry (binary) adder** because the carry propagates (ripples) from one stage to the next stage.

BINARY SUBTRACTOR

The circuit, which performs the subtraction of two binary numbers is known as **Binary subtractor**. We can implement Binary subtractor in following two methods.

- Cascade Full subtractors
- 2's complement method

In first method, we will get an n-bit binary subtractor by cascading 'n' Full subtractors. So, first you can implement Half subtractor and Full subtractor, similar to Half adder & Full adder. Then, you can implement an n-bit binary subtractor, by cascading 'n' Full subtractors. So, we will be having two separate circuits for binary addition and subtraction of two binary numbers.

In second method, we can use same binary adder for subtracting two binary numbers just by doing some modifications in the second input. So, internally binary addition operation takes place but, the output is resultant subtraction.

We know that the subtraction of two binary numbers A & B can be written as,

$A-B = A+(2's \text{ complement of } B)$

$A-B = A+(2's \text{ complement of } B)$

$\Rightarrow A-B = A + (1's \text{ complement of } B) + 1$

4-bit Binary Subtractor

The 4-bit binary subtractor produces the **subtraction of two 4-bit numbers**. Let the 4bit binary numbers, A = A3 A2 A1 A0 and B = B3 B2 B1 B0 . Internally, the operation of 4-bit Binary subtractor is similar to that of 4-bit Binary adder. If the normal bits of binary number A, complemented bits of binary number B and initial carry (borrow), C_{in} as one are applied to 4-bit Binary adder, then it becomes 4-bit Binary subtractor. The **block diagram** of 4-bit binary subtractor is shown in the following figure.

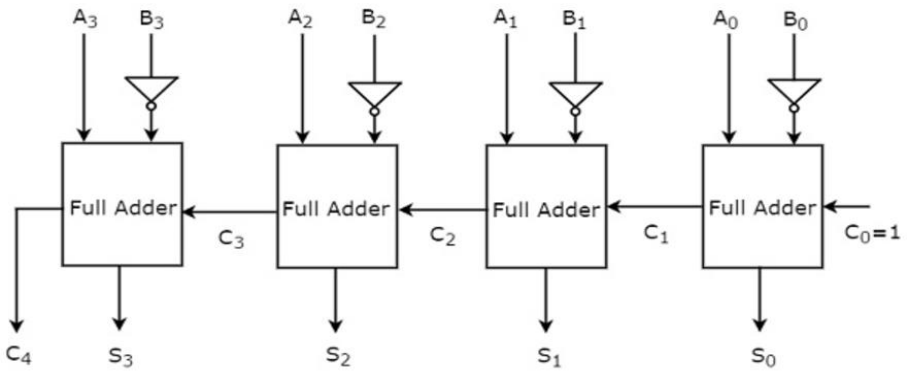


Fig 7: 4-Bit Binary Subtractor Circuit

This 4-bit binary subtractor produces an output, which is having at most 5 bits. If Binary number A is greater than Binary number B, then MSB of the output is zero and the remaining bits hold the magnitude of A-B. If Binary number A is less than Binary number B, then MSB of the output is one. So, take the 2's complement of output in order to get the magnitude of A-B.

In this way, we can implement any higher order binary subtractor just by cascading the required number of Full adders with necessary modifications.

4-bit Binary Adder / Subtractor

The 4-bit binary adder / subtractor produces either the addition or the subtraction of two 4-bit numbers based on the value of initial carry or borrow, C_0. Let the 4-bit binary numbers, A = A3 A2 A1 A0 and B = B3 B2 B1 B0. The operation of 4-bit Binary adder / subtractor is similar to that of 4-bit Binary adder and 4-bit Binary subtractor.

Apply the normal bits of binary numbers A and B & initial carry or borrow, C_0 from externally to a 4-bit binary adder. The **block diagram** of 4-bit binary adder / subtractor is shown in the following figure.

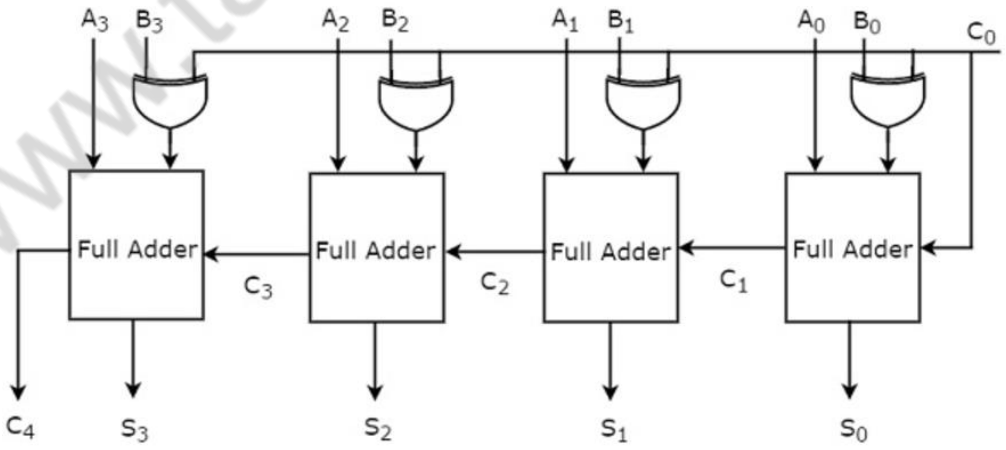


Fig 8: 4-Bit Binary Adder /Subtractor Circuit

If initial carry, C_0 is zero, then each full adder gets the normal bits of binary numbers A & B. So, the 4-bit binary adder / subtractor produces an output, which is the **addition of two binary numbers** A & B.

If initial borrow, C_0 is one, then each full adder gets the normal bits of binary number A & complemented bits of binary number B. So, the 4-bit binary adder / subtractor produces an output, which is the **subtraction of two binary numbers** A & B.

Therefore, with the help of additional Ex-OR gates, the same circuit can be used for both addition and subtraction of two binary numbers.

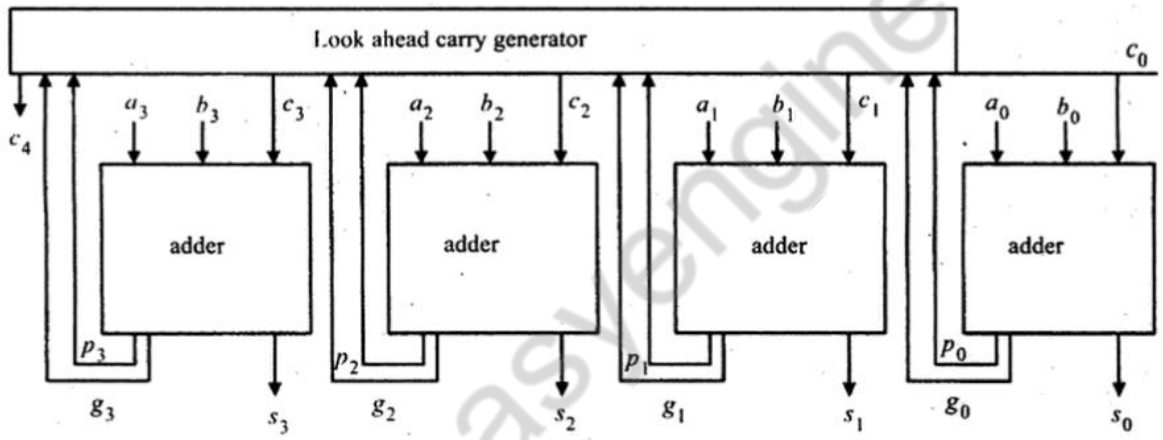


Fig 9: 4-Bit Carry Look Ahead Adder Block Diagram

Pi = Ai ⊕ Bi Carry propagate

Gi = AiBB i Carry generate

For i=0

$c_1 = g_0 + p_0c_0$

For i=1

$c_2 = g_1 + p_1c_1$

$= g_1 + p_1(g_0 + p_0c_0)$

$= g_1 + p_1g_0 + p_1p_0c_0$

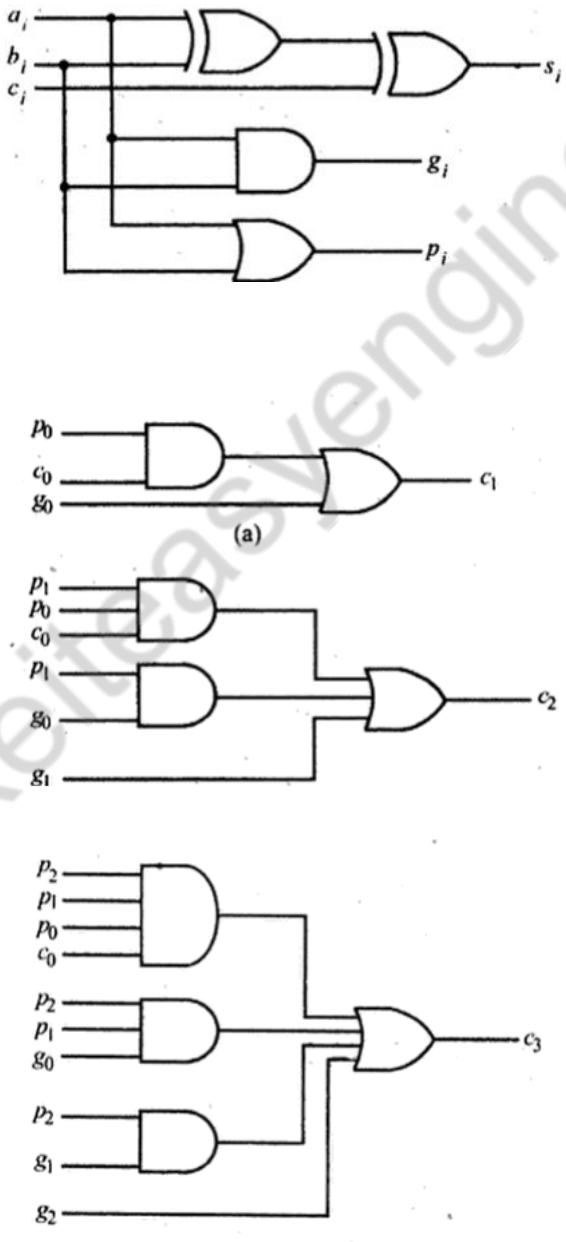
For i=2

$c_3 = g_2 + p_2c_2$

$= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

For i=3

$c_4 = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$



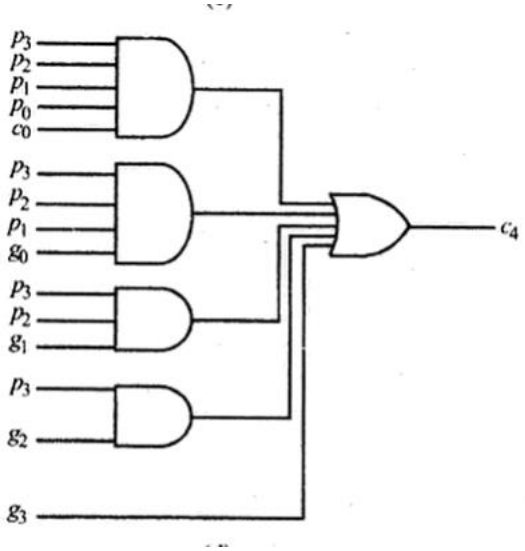


Fig 10: 4-Bit Carry Look Ahead Adder Logic circuit

Complex Programmable Logic Devices (CPLD's)

- As number of Boolean expression increases, designing a digital circuit using PLD's becomes difficult.
- To overcome this problem we can use complex programmable logic devices.
- Using CPLDs we can implement more than 20 Boolean expression in a digital circuit.

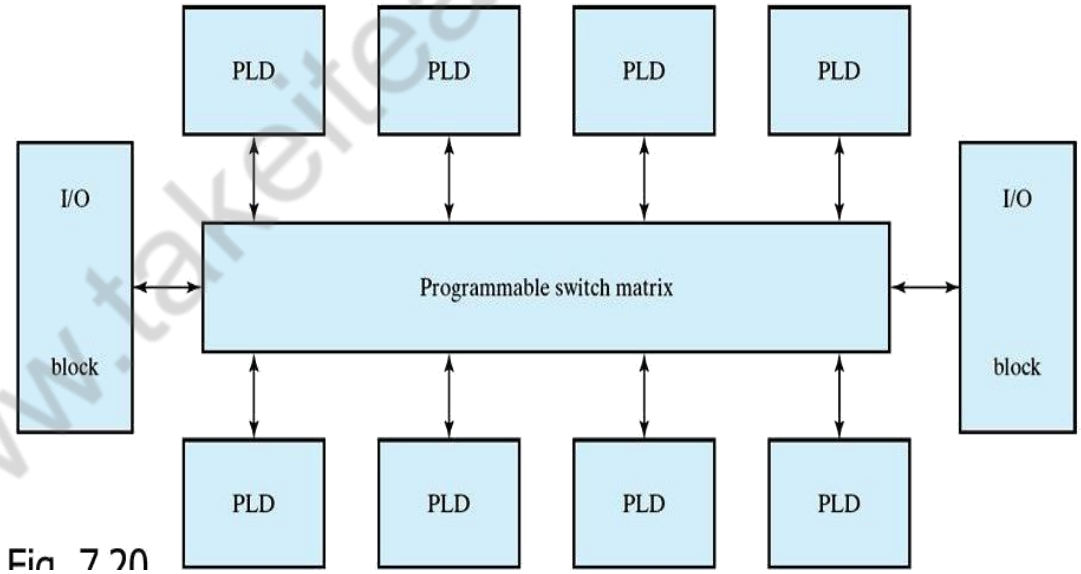


Fig 11: Block Diagram of CPLD

Features of CPLD

- High Performance
- Fast connection of Switch matrix
- Programmable switching mode

Applications of CPLD

- It is used in a digital circuit where Number of input and output are > 32.
- Television and Automation Industries.
- Implementation of large digital circuits.

Field Programmable Gate Arrays (FPGA)

- A **Field-Programmable Gate Array** is an integrated circuit silicon chip which has array of logic gates and this array can be programmed in the field i.e. the user can overwrite the existing configurations with its new defined configurations and can create their own digital circuit on field. The FPGAs can be considered as blank slate. FPGAs do nothing by itself whereas it is up to designers to create a configuration file often called a **bit file for the FPGA**. The FPGA will behave like the digital circuit once it is loaded with a bit file.
- Field Programmable Gate Arrays (**FPGAs**) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects.
- FPGAs are particularly useful for prototyping application-specific integrated circuits (ASICs) or processors.
- An FPGA can be reprogrammed until the processor design is final and bug-free.

FPGA Architecture

- An FPGA has a regular structure of logic cells or modules and interlinks which is under the developers and designers complete control. The FPGA is built with mainly three major blocks such as Configurable Logic Block (CLB), I/O Blocks or Pads and Switch Matrix/ Interconnection Wires. Each block will be discussed below in brief.
- CLB (Configurable Logic Block): These are the basic cells of FPGA. It consists of one 8-bit function generator, two 16-bit function generators, two registers (flip-flops or

latches), and reprogrammable routing controls (multiplexers). The CLBs are applied to implement other designed function and macros. Each CLBs have inputs on each side which makes them flexile for the mapping and partitioning of logic.

- I/O Pads or Blocks: The Input/Output pads are used for the outside peripherals to access the functions of FPGA and using the I/O pads it can also communicate with FPGA for different applications using different peripherals.
- Switch Matrix/ Interconnection Wires: Switch Matrix is used in FPGA to connect the long and short interconnection wires together in flexible combination. It also contains the transistors to turn on/off connections between different lines.
-

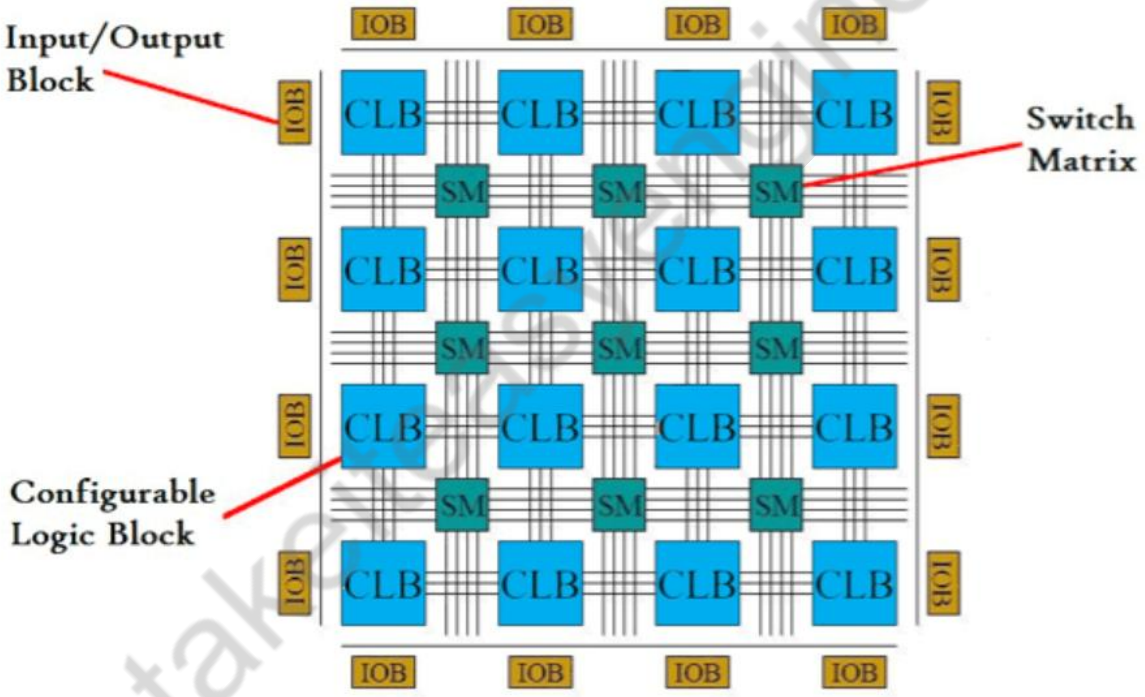


Fig 12: Block Diagram of FPGA

Configurable logic blocks:

Each Configurable logic block can generate a Logic function with many inputs.

Interconnection Switches:

They are used to interconnect various block with input/output blocks.

Application:

FPGA most commonly used in Digital systems such as smart phones, Computer systems etc.

CPLD vs FPGA comparison summary

	CPLD	FPGA
1.	Instant-on. CPLDs start working as soon as they are powered up	Since FPGA has to load configuration data from external ROM and setup the fabric before it can start functioning, there is a time delay between power ON and FPGA starts working. The time delay can be as large as several tens of milliseconds.
2.	Non-volatile. CPLDs remain programmed, and retain their circuit after powering down. FPGAs go blank as soon as powered-off.	FPGAs uses SRAM based configuration storage. The contents of the memory is lost as soon as power is disconnected.
3.	Deterministic Timing Analysis. Since CPLDs are comparatively simpler to FPGAs, and the number of interconnects are less, the timing analysis can be done much more easily.	Size and complexity of FPGA logic can be humongous compared to CPLDs. This opens up the possibility less deterministic signal routing and thus causing complicated timing scenarios. Thankfully implementation tools provided by FPGA vendors have mechanisms to assist achieving deterministic timing. But additional steps by the user is usually necessary to achieve this.
4.	Lower idle power consumption. Newer CPLDs such as CoolRunner-II use around 50 uA in idle conditions.	Relatively higher idle power consumption.
5.	Might be cheaper for implementing simpler circuits	FPGAs are much more capable compared to CPLDs but can be more expensive as well.
6.	More "secure" due to design storage within built in non-volatile memory.	FPGAs that use external memory can expose the IP externally. Many FPGA vendors offer mechanisms such as encryption to combat this. Design specific protection mechanisms also can be implemented.

7.	Very small amount of logic resources.	Massive amount logic and storage elements, with which incredibly complex circuits can be designed. FPGAs have thousands times more resources! This point alone makes FPGAs more popular than CPLDs.
8.	No on-die hard IPs available to offload processing from the logic fabric.	Variety of on-die dedicated hardware such as Block RAM, DSP blocks, PLL, DCMs, Memory Controllers, Multi-Gigabit Transceivers etc give immense flexibility. This is not even thinkable with CPLDs.
9.	Power down and reprogramming is always required in order to modify design functionality.	FPGAs can change their circuit even while running! (Since it is just a matter of updating LUTs with different content) This is called Partial Reconfiguration, and is very useful when FPGAs need to keep running a design and at the same time update the it with different design as per requirement. This feature is widely used in Accelerated Computing.

MODULE-3 FLIP-FLOPS AND ITS APPLICATIONS

The Master-Slave Flip-flops (Pulse-Triggered flip-flops): SR flip-flops, JK flip flops, Characteristic equations, Registers, Binary Ripple Counters, Synchronous Binary Counters, Counters based on Shift Registers, Design of Synchronous mod-n Counter using clocked T, JK, D and SR flip-flops.

Combinational Circuit and Sequential Circuit

Key	Combinational Circuit	Sequential Circuit
Definition	A Combinational Circuit is a type of circuit in which the output is independent of time and only relies on the input present at that particular instant.	A Sequential circuit is a type of circuit where output not only relies on the current input but also depends on the previous output.
Feedback	Since output does not depend on the time instant, no feedback is required for its next output generation.	The output relies on its previous feedback so output of previous input is being transferred as feedback used with input for next output generation.
Performance	As the input of current instant is only required in case of Combinational circuit, it is faster and better in performance as compared to that of Sequential circuit.	Sequential circuits are comparatively slower and has low performance as compared to that of Combinational circuit.
Complexity	No implementation of feedback makes the combinational circuit less complex as compared to sequential circuit.	The implementation of feedback makes sequential circuit more complex as compared to combinational circuit.
Elementary Blocks	The elementary building blocks of a combinational circuit are its logic gates.	The building blocks of a sequential circuit are the logic gates along with flip flops.
Operation	Combinational circuits are mainly used for arithmetic as well as Boolean operations.	Sequential circuits are mainly used for storing data.

SEQUENTIAL CIRCUITS- I

Digital logic can be classified into two major sections

1. Combinational logic
2. Sequential logic

Combinational Logic: Combinational logic deals with the technique of "combining" logic gates into circuits that perform some desired function.

Ex: Adders, Subtractors, Decoders, Encoders, Multiplier, Divider.

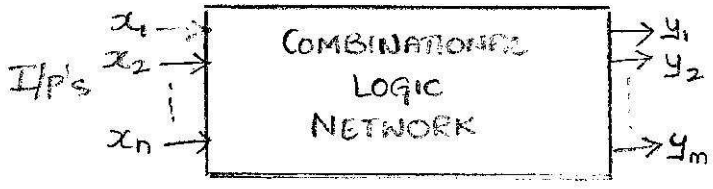


Fig: Block Diagram of Combinational N/W.

- A Combinational network is a two-valued network in which the outputs at any instant of time are **dependent only upon the inputs** present at that instant. Thus the output of a combinational network can be described by a single algebraic expression whose variables are inputs to the network.
- A Combinational network contains no closed loops or feedback paths and hence they are said to be acyclic.
- Combinational circuits are easy to design.
- Truth table is a convenient representation of the operation of a combinational circuits

Sequential Circuits: Many applications in digital world require circuits in which digital outputs are required to be generated in accordance with the sequence in which the input signals are received. Such applications require output to be generated that are not only **dependent on the present input condition but also upon the past history** of these inputs.

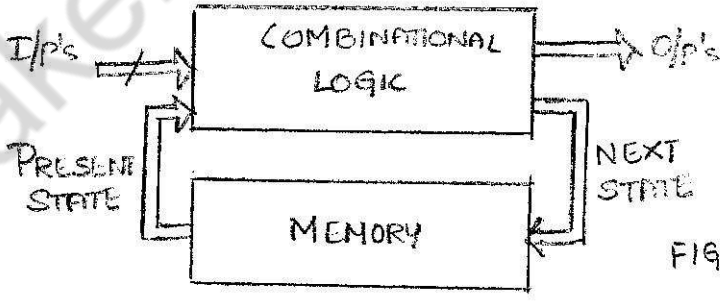


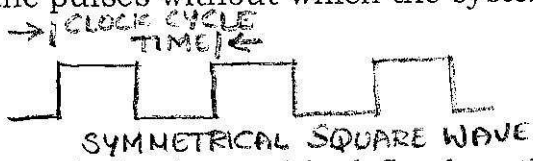
Fig: General Model of Sequential Network

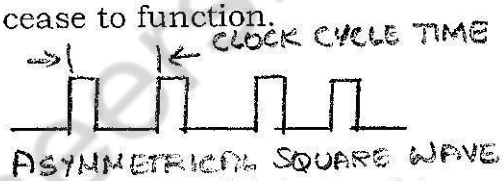
- A Sequential network is defined as a two-valued network in which the outputs at any instant are dependent on only upon the past history or past sequence of inputs.
- The past history of inputs is provided by feedback from output to the input.

- Since the past history of inputs must be preserved by the network, sequential circuits are said to have memory.
- The operation of sequential circuits is expressed in terms of their states. The state or internal state is a set of signals at a given point in the sequential circuit.
- Output of a sequential circuit or network is a function of the present external inputs and its present state i.e. the two determine the next state.
- The basic memory (ability to store information) element in sequential circuits is **Flip flop**.

  A Flip flop is a sequential circuit which can store a 1 or a 0 indefinitely. Thus a flip flop is said to possess two stable states. It continues indefinitely in one of these two stable states until a change is prompted by the change in its input.

Clock: The heart of every digital system is the system clock. The system clock provides the pulses without which the system would cease to function.





- The basic timing interval is defined as the clock cycle time and it is equal to one period of the clock waveform; it defines the timing interval during which logic operations must be performed.
- There are two basic types of sequential networks.

a. **Synchronous** sequential network: - In such systems, a change of state occurs in synchronism with the system clock. A change of state will either occur as the clock transitions from low to high or as it transition from high to low.

Low to high transition ---> Positive transition (PT)
High to low transition ---> Negative transition (NT)

A circuit that changes state during PT is called a **positive edge triggered** circuit and that for NT is called as **negative edge triggered** circuit.

b. **Asynchronous** sequential network: - There is no synchronization control and hence change in input itself can cause a change in output and state.

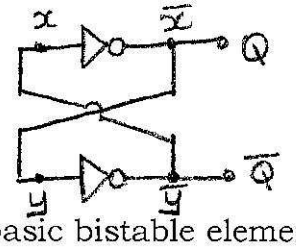
These networks are capable of high speed performance since it is no longer necessary to wait for the sample times occur before the network responds to the input changes. Since they do not have a synchronizing clock, the memory portion of asynchronous sequential networks consists of unclocked flip flops or time delay elements. A time delay device provides memory by the fact that it takes a finite amount of time for a signal to propagate through it and correspondingly serves to store information for that time duration.

	COMBINATIONAL CIRCUITS	SEQUENTIAL CIRCUTS
1.	Output at any instant of time are dependent only upon the inputs present at that instant	Output at any instant are dependent not only on the present input but also on the past history of these input variables
2.	Memory unit is not required	Memory unit is required to store the past history of inputs
3.	Faster in speed because the delay between input and output is due to propagation delay of gates	Slower in speed
4.	Easy to design Ex: Parallel adder	Comparatively not easier Ex: Serial adder

	SYNCHRONOUS SEQUENTIAL CIRCUITS	ASYNCHRONOUS SEQUENTIAL CIRCUITS
1.	Memory elements are clocked flip flops	Memory elements are either unclocked flip flops or time delay elements.
2.	Change in input signals can cause a change in output or state only on activation of clock signal.	Change in input itself can cause change in output or state.
3.	Maximum operating speed of clock depends on time delays involved.	Due to absence of clock they can operate faster than synchronous circuits
4.	Easier to deign	Difficult to design

Propagation time delay is the time required for a signal to pass from the input of a circuit to its output.

Basic Bistable Element:



- Central to all flip flops is the basic bistable element as shown above. It consists of two cross-coupled not gates i.e. output of first not gate serves as input to the second not gate & output of second serves as input to the first not gate
- A basic bistable element is a circuit having two stable states

  Let x=0 then $Q = \bar{x} = 1$

  So y=1 then $\bar{Q} = \bar{y} = 0$

  Thus $\bar{Q} = x = \bar{y} = 0$ & $Q = \bar{x} = y = 1$ (circuit is stable with these values)

  Similarly let x=1 then $Q = \bar{x} = 0$

So y=0 then $\bar{Q} = \bar{y} = 1$

Thus $\bar{Q}$= x = $\bar{y}$ = 1 & Q = $\bar{x}$ = y =0 (2nd stable state)

- Since it has two stable states, the basic bistable element is used to store binary symbols. In case of positive logic, when Q=1 the element is said to store a 1 and when Q=0, the element is said to store a 0.
- The binary symbol that is stored in the basic bistable element is referred to as the content or state of the element. The state is given by the signal value at the Q output terminal. Thus Q output is called normal output & $\bar{Q}$ is referred to as Complementary output.
- When the device is storing a 1 it is said to be in **1-state or set state** and when the device is storing a 0 it is said to be in **0-state or reset state.**
- A Third equilibrium state can exist when the two output signals are about halfway between logic-0 and logic-1. Output is not a valid logic signal. This is known as the metastable state.
- When power is applied, it becomes stable in one of its two stable states. It remains in this state until power is removed.

— * — * — * — * —

- **A flip flop** is a bistable device, with inputs, that remain in a given state as long as power is applied and until input signals are applied to cause change in its output. The process of storing a 1 into a flip flop is called **setting or presetting** the flip flop; while the process of storing a 0 into a flip flop is called **resetting or clearing** the flip flop
- Inputs to a flip flop are two types
  Asynchronous input
  Synchronous input

Asynchronous input: is one in which a signal change of sufficient magnitude and the duration essentially produces an immediate change in the state of the flip flop.

Synchronous input: is one that does not immediately affect the state of the flip flop but rather affects the state of the flip flop only when some control signal, usually called as enable or clock input also occurs.

Latches: - Latches and flip flops are the basic building blocks of the most sequential circuits; the difference is in the method used to change their states.

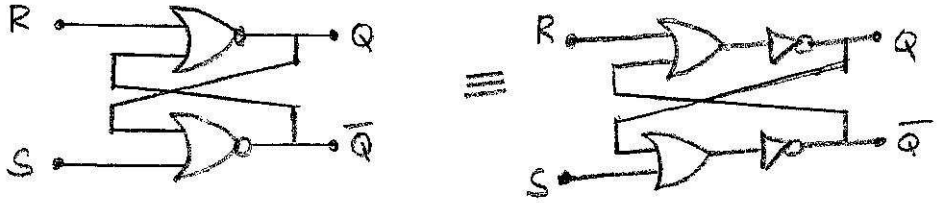
- Flip flop changes state only at times determined by the clocking state.
- Latches changes state corresponding to changes in input independent of the clocking signal. A special control signal called as enable or clock is provided with the latch, when enable signal is active output changes occur as input changes.

- But when enable signal is not activated input changes do not affect output, such latches are called as **gated latch**.

1. SR Latch (Set-Reset):-

- Simplest type of latch that consists of two cross-coupled NOR gates

LOGIC DIAGRAM



- It has two inputs namely S (set) and R (Reset) and two outputs Q & $\bar{Q}$.

FUNCTION TABLE

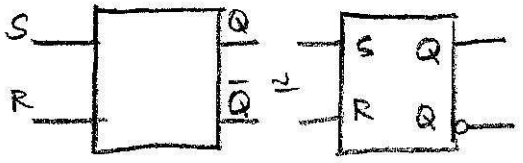
INPUTS		OUTPUTS	
S	R	Q^+	$\bar{Q}^+$
0	0	Q	$\bar{Q}$
0	1	0	1
1	0	1	0
1	1	0*	0*

* UNPREDICTABLE STATE

- **Case1:** When S = R = 0, Circuits becomes similar to basic bistable element i.e, cross-coupling of two NOT-gates thus latch is in one of the two stable states.
  In the Truth table Q & $\bar{Q}$ represents the state of the latch when the inputs are applied i.e. Q is the present state, Q^+& $\bar{Q}^+$ represents the response of the latch to the inputs applied. i.e. $\bar{Q}^+$ is the next state of the latch.
  Thus the outputs do not change and the next state equals the present state.

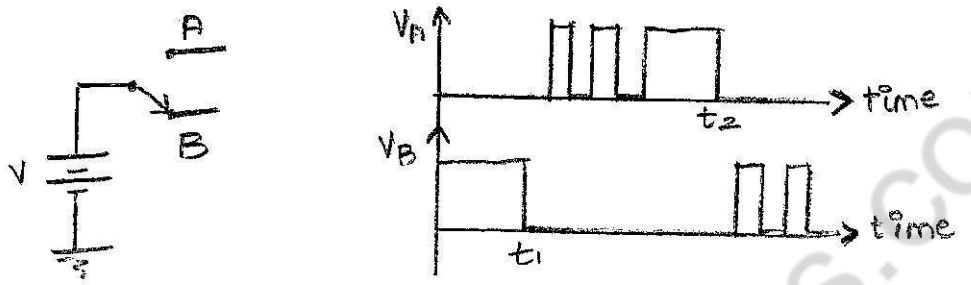
  i.e. $Q^+ = Q$
- **Case 2:** When S=0, R=1, Regardless of the second input to the upper NOR gate, the output Q must become 0 since R=1. This in turn is fed to the lower NOR gate. Thus Q=0 & $\bar{Q}$ = 1.(reset state)
- **Case 3:** When S=1, R=0, The latch becomes set regardless of its present state because one of the input to the lower NOR gate is a 1.
  Thus Q=1 & $\bar{Q}$ = 0.(set state)
- **Case 4:** When S=1, R=1, Outputs of both NOR gates becomes '0' and hence it is a forbidden state. When both S & R transit to zero simultaneously, then the latch could get into the metastable state which is undesirable.

SYMBOL



An application of SR latch: A switch debouncer

- When mechanical switches such as toggle switches or push buttons are switched from one position to the other, several make and break operations occur at the second position called **Switch Bounce.**



- At time t_1, the centre contact is moved from B to A first the voltage at B becomes zero as soon as contact leaves B. However, when it reaches A, several make and break operations takes place due to the spring like nature of the contacts.

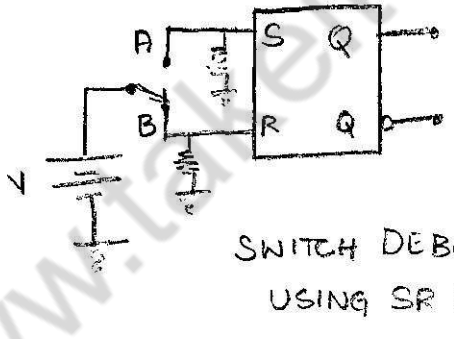
➢ During contact bounce, the center contact of the switch does not return all the way to terminal B. similar occurrence can be seen at B, when center contact is moved from A to B at t_2. This effect is called **switch bounce** and is normally undesirable.

Ex: Push-button keys on a keyboard, contact bounce causes a system to respond as through a key was pressed several times in succession.

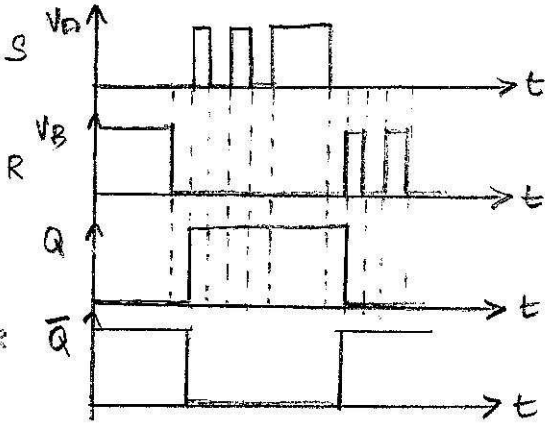
➢ This effect is undesirable even in digital circuits because between t_1 & t_2, the logic level read by a gate connected to A could be a 0 or a 1 depending on the instance when the switch is read, leading to unpredictable results.

Such a switch bounce can be eliminated by the use of a SR latch.

A Switch Debouncer:

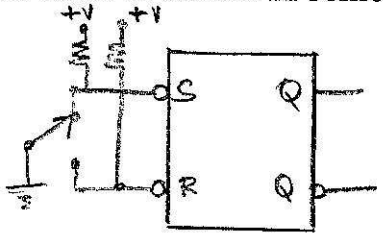


SWITCH DEBOUNCER USING SR LATCH



- Let center contact of the switch be initially at position B, i.e. S=0, R=1
  Therefore Q=0, $\overline{Q}$=1 for time $t < t_1$
- At time t_1, switch moves from position B to A, after the center contact leaves B 7 till it reaches A, S = R = 0, thus output remains Q=0, $\overline{Q}$=1
- As soon as contact ouches A, he inputs to the latch become S=1, R=0, therefore Q=1, $\overline{Q}$ =0

- Now, when the center contact temporarily leaves A due to contact bounce, the inputs becomes S=R=0, thereby Q remains 1 & $\overline{Q}$=0. Similar action takes place when center contact is switched back to B at t = t_2
- Thus SR latch is useful in removing the effect of contact bounce as glitches.



NOTE: WAVEFORMS REMAIN SAME AS OF SR LATCH SWITCH DEBOUNCER

FIG: SWITCH DEBOUNCER USING $\overline{S}\overline{R}$ LATCH

$\overline{S}\,\overline{R}$ Latch:

- The $\overline{S}\,\overline{R}$ latch is constructed by cross-coupling two NAND gates

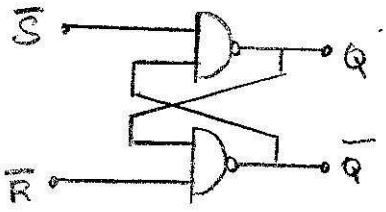


FIG: LOGIC DIAGRAM

INPUTS		OUTPUTS	
$\overline{S}$	$\overline{R}$	Q^+	$\overline{Q}^+$
0	0	1	1
0	1	1	0
1	0	0	1
1	1	Q	$\overline{Q}$

FORBIDDEN STATE

FIG: FUNCTION TABLE

- It has two inputs, namely $\bar{S}$ & $\bar{R}$ and two outputs Q & $\bar{Q}$
- When $\overline{S}$ = 1; $\overline{R}$ = 1, Circuits becomes similar to basic bistable element i.e. Cross-coupling of two NOT gates. Thus the device has 2 stable states.
- When $\overline{S}$ =0, $\overline{R}$ = 1, (set state) The latch becomes set regardless of the second input to the upper NAND gate, therefore S = 0 thus Q=1 & $\bar{Q}$ =0
- When $\overline{S}$ =1 $\overline{R}$ = 0, (reset state) The latch becomes reset and hence Q = 0 & $\bar{Q}$ =1
- When $\overline{S}$ = $\overline{R}$ =0, (forbidden state) Outputs of both NAND gates becomes '1' and hence forbidden state
  - If both S & R transit to 1 simultaneously, then the latch could enter into the metastable state which is undesirable.

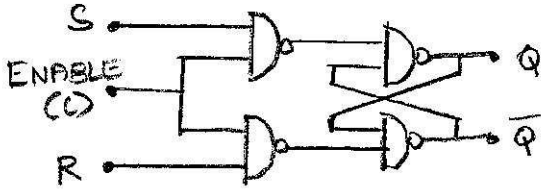
Ex: Switch debouncer using $\overline{SR}$ latch

LOGIC DIAGRAM AND WAVEFORMS AS GIVEN ABOVE

FIG: SYMBOL OF $\overline{SR}$ LATCH

Gated SR Latch

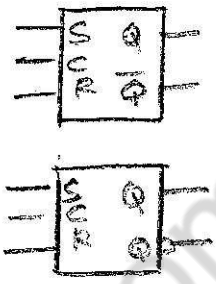
- Gated SR latch is also called as SR latch **with enable.**



LOGIC DIAGRAM

S	R	C	Q^+	$\overline{Q}^+$
0	0	1	Q	$\overline{Q}$
0	1	1	0	1
1	0	1	1	0
1	1	1	1*	1*
x	x	0	Q	$\overline{Q}$

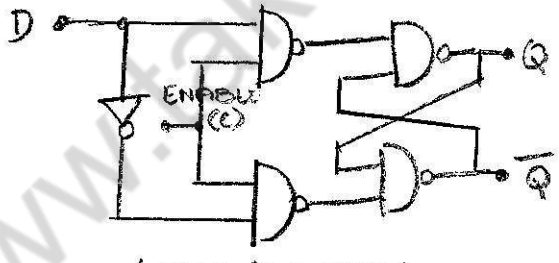
SYMBOL



- A gated SR latch is as shown above. It consists of the $\bar{S}\ \bar{R}$ latch along with two additional NAND gates and a control input, C referred to as the enable, gate or clock input.
  The enable input decides when the inputs S & R are effective i.e. it decides if a change in input causes corresponding changes in output of flip flop
- As long as the enable input is zero, the output of the two NAND gates (X & Y) is 1. This output is in turn fed as input to the NAND gates (A & B) and hence according to the function table of $\bar{S}\ \bar{R}$ latch, $\bar{S}\bar{R}$ latch remains in its current stable state. The latch is set to be disabled.
- When the enable input is made high, the latch is said to be enabled.
- The NAND gate inverts the signals on the S and R input lines. When the latch is enabled.
- Since the effects of the S and R inputs are dependent upon the presence of an enable signal, these inputs are classified as synchronous inputs.

Gated D Latch (Data Latch)

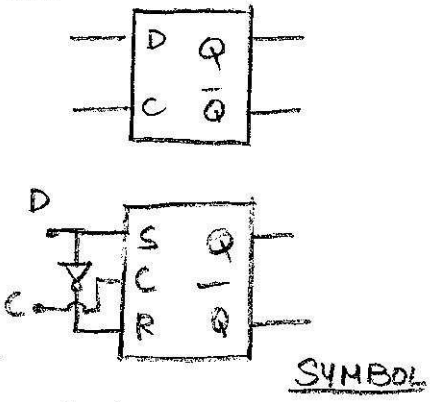
- SR, $\bar{S}\bar{R}$ & gated $\bar{S}\bar{R}$ latch had an inputs combination that was forbidden, in particular when the inputs S & R or $\bar{S}$ & $\bar{R}$ are simultaneously high.
  Hence a gated D-latch was designed to overcome this problem.
- The logic diagram of a gated D-latch is as shown



LOGIC DIAGRAM

D	C	Q^+	$\overline{Q}^+$
0	1	0	1
1	1	1	0
x	0	Q	$\overline{Q}$

FUNCTION TABLE



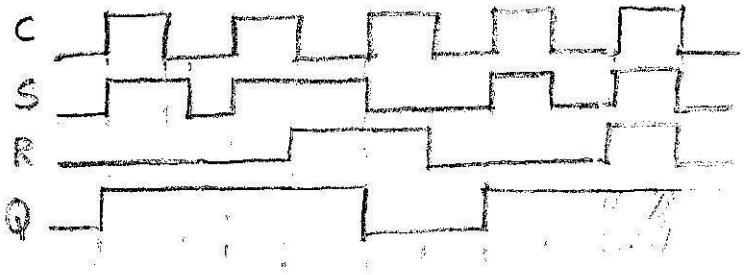
SYMBOL

- A NOT gate is connected between the S and R terminals thus the latch consists of a single input 'D' that determines the next sate.
  It also consists of a control signal, i.e. enable or clock that determines when the 'D' input is effective.

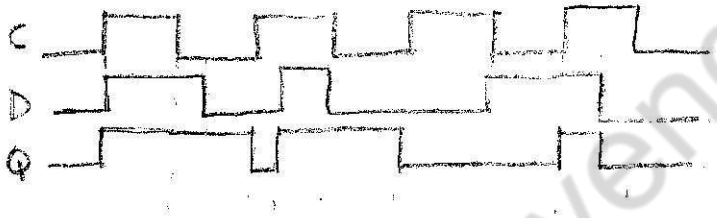
- When the latch is enable C=1, the output of the latch follows the D-input. i.e. if D is 0, then output switches to or remains in the 0-state and vice-versa.
- If C=0 i.e. the latch is disabled, then the latch remains in the previous state itself.
- D-flip flop (Transparent latch):- A flip flop whose output follows its data input when the clock is active.

Timing diagram is a graph that depicts the input and output transactions of a flip-flop as a function of a time.

Timing diagram for an SR latch [GATED]



Timing diagram for an gated D latch



- Applying a '1' simultaneously to both the S and R input terminals when C=1 should be avoided to prevent the possibility of an unpredictable state when the signals are removed subsequently or if C is changed to 0 while both signals are '1'.

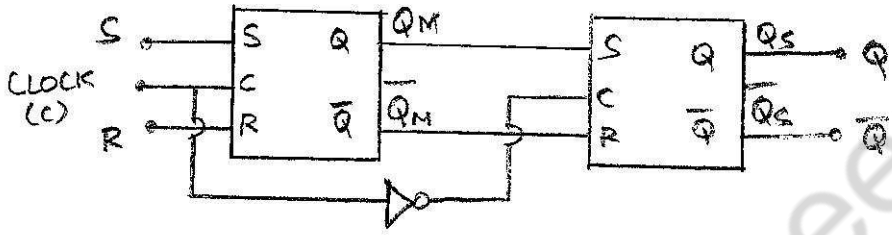
Master-Slave Flip Flops (Pulse Trigged Flip Flop):

- Flip flops can be categorized
  - Pulse triggered flip flops
  - Edge triggered flip flops
- Latches respond immediately to changes in input. The enable or clock i.e. the control signal might be present wherein changes in input cause changes in output only when the enable is high. This property is referred as **transparency**.
- In some applications, this property is undesirable wherein it is necessary that he output changes occurs only with changes in a control input line like the clock.
- The purpose of master-slave flip flops is to protect the flip flop's output changing or responding to glitches on the input. Master-slave flip flops are used in applications where glitches are prevalent on inputs.

- A master-slave flip flop consists of two cascaded sections, 1st section is referred to as master and 2nd section as the slave.
- Information is entered into the master on one edge or level of the control signal and the information is transferred to the slave on the next edge or level of the control signal.
  Each section is a Latch.

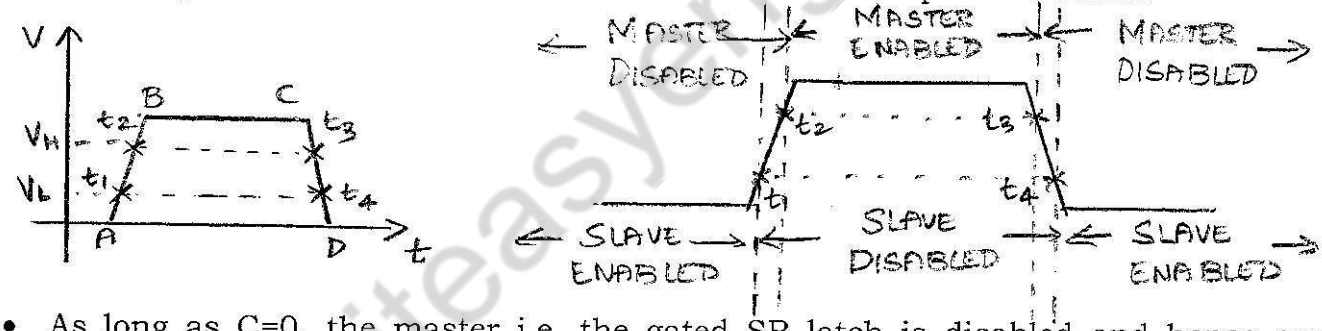
Master Slave SR Flip Flops:

- The master – slave SR flip-flop is constructed using two gated SR latches and an inverter as shown below



MASTER - SLAVE SR FLIP FLOP

- The S & R inputs are used to set and reset the flip-flop. The clock acts as a control input. Operation of master-slave with reference to clock pulse is shown



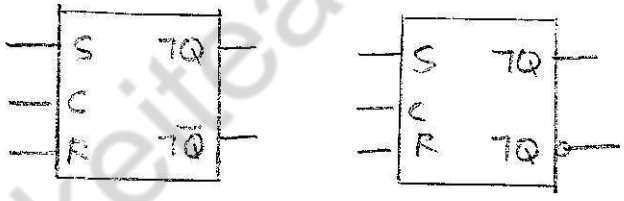
- As long as C=0, the master i.e. the gated SR latch is disabled and hence any changes on the S & R input lines are ignored. But the slave is enabled due to the preference of an inverter.
- Since the slave is enabled, it is in the same state as that of the master because the outputs of the master Q_M & $\overline{Q}_M$ are connected to S & R inputs respectively. i.e. State of master is transferred to the slave when clock is at logic 0
- At time t_1, the slave is disabled but retains the state of master .
- The control signal continues to rise and at time t_2, the master is enabled and it responds to the inputs S & R.
- Changes in the state of the master are not reflected to the slave since the slave is disabled.
- At time t_3, the control signals is returned to its low level and hence the master gets disabled.
- At time t_4, the slave is enabled and hence it takes the state of the master.

- For very short instants of time, during rising and falling edges of the control signal both the master and the slave latches are disabled.
- The master can change its state anytime when the clock input is logic 1, whereas the slave can change state only along the falling edge of the clock input at time t_4.
- Thus, it can be observed that the output change of the master-slave flip flop is synchronized to the falling edge o the control signal.

Function table of master-slave SR-flip flop:

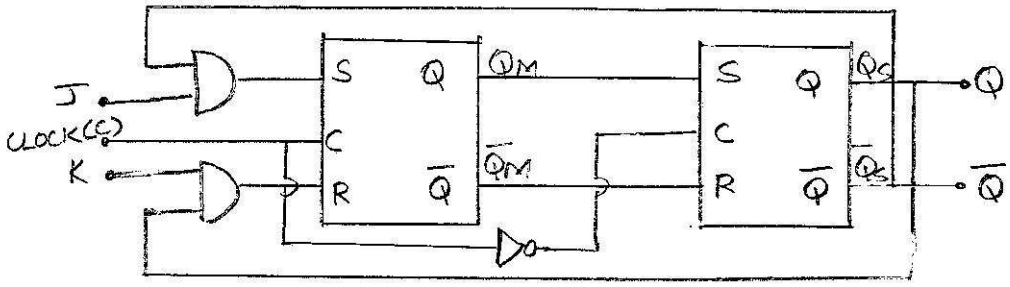
S	R	C	Q^+	$\overline{Q}^+$
0	0	⊓	Q	$\overline{Q}$
0	↓	⊓	0	1
↓	0	⊓	1	0
1	1	⊓	Undefined	
X	X	0	Q	$\overline{Q}$

- The clock pulse shown in the table represent the fact that the master is enabled as long as the clock is at 1 and the state of the master is transferred to the slave when the clock begins to go to '0'.
- When S=R=1, when control signals goes from high to low. The master enters into an unpredictable state; this state is transferred to the slave. Thus, the output of the master-slave flip flops itself becomes unpredictable. Hence such a condition is undesirable and should be avoided.
- Since, the state of the master depends on the period of time for which the clock is at logic 1 and the state of the slave changes at the falling edge these flip flops are also called as pulse triggered flip flops.
- The two logic symbols of the master-slave SR flip flop



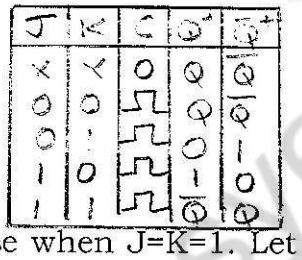
- The ' ⊓ 'symbol represents the fact that the output changes only at the falling edge of the pulse and is called as the postponed output indicator. Since the output is postponed till the end of the pulse period.

Master-Slave JK Flip Flop(MS- JK)



MASTER - SLAVE JK FLIP FLOP

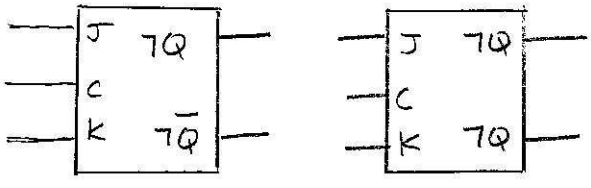
- A master slave JK flip flop is constructed using a MS SR flip flop and two AND gates as shown above.
- The main purpose of a MS JK flip flop is to overcome the undefined output state of MS SR flip flop when S=R=1 during ON pulse of the clock.
- It has two inputs namely, J & K that are analogous to S & R inputs. Thus J & K inputs are used to set and reset the flip flop respectively.
- The truth table of a master-slave JK flip flop is as shown



- Consider the case when J=K=1. Let Q=0 & $\bar{Q}$=1, thus the output of the lower, i.e. K-input AND gate is 0 and the output of the J-input AND gate is a 1, as a result of feedback. Thus R=0, S=1 to the master flipflop.
- At the instant the clock goes high, master sets i.e. Q_M=1 and $\bar{Q}_M$=0.
- On the next falling edge of the clock, the state of the master is transferred to the slave thus Q_S=1 & $\bar{Q}_S$=0.
- Similar condition can be observed when Q=1 & $\bar{Q}$=0 wherein the output changes to Q_S = 0 & $\bar{Q}_S$ =1.
- When J=K=0, both the AND gate outputs are zero and this corresponds to S=R=0. Thus the next state is the same as the present state.
- Race – around toggling condition of JK FF when J=K=1.

'⊓' -> postponed output indicator

SYMBOLS

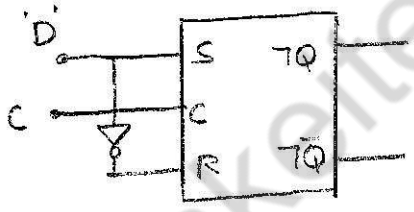


0's catching and 1's catching:

- A Master-slave JK flip flop is subject to 0's & 1's catching
- If the slave latch is in its 1-state then if first IP's to flip flop are J=0, K=1 causes master to reset.
  The slave resets during falling edge of the clock.
- Once the master is reset, any changes in J & K inputs are not reflected by master since the slave does not change its state until C returns to 0 (negative edge)and the feedback from slave keeps master in the same state. This is 0's catching; similarly we have 1's catching.

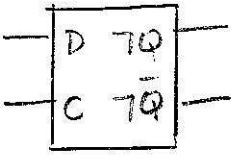
Additional type of Master – Slave flip flops:

MASTER-SLAVE 'D' FF: - By placing an inverter between the S & R inputs, of a MS SR FF a MSD-FF can be realized.



FUNCTION TABLE

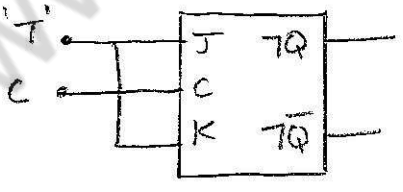
D	C	Q^+	$\bar{Q}^+$
X	0	Q	$\bar{Q}$
0	↓	0	1
1	↓	1	0



SYMBOL

MASTER – SLAVE 'T' FF: - By shorting the inputs J&K of a MS JK FF we get a MS T-FF.

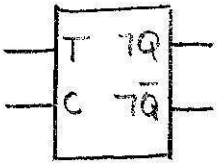
- The flip flop toggles when T=1 & does not change its state when T=0



LOGIC DIAGRAM

T	C	Q^+	$\bar{Q}^+$
X	0	Q	$\bar{Q}$
0	↓	Q	$\bar{Q}$
1	↓	$\bar{Q}$	Q

FUNCTION TABLE



SYMBOL

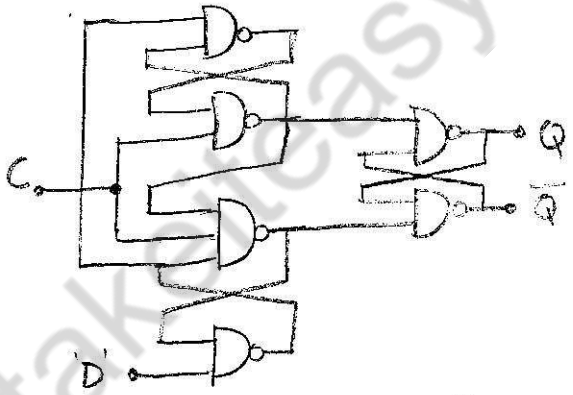
Edge triggered flip flops:

- In a master slave flip flop, the master is enabled as long as clock is 1, this leads to 0's catching & 1's catching
- To avoid the 0's & 1's catching problem, it should be seen that the J & K inputs are held constant when the clock input is 1, i.e. the master is enabled.
- In this way, the state of the master can be established during the positive edge of the clock and then transferred to the slave during the negative edge of the clock.
- Such flip flops are called as edge triggered flip flops. These flip flops respond only at the edges of the clock and ignore any changes in the inputs in between triggering edges

- Edge-triggered flip flops can be classified as
  - Positive edge triggered flip flops
  - Negative edge triggered flip flops

Positive edge triggered D flip flop:

- Logic diagram of positive edge triggered D flip flop is as shown below.
- D is the input & C is the control or the clock input.



LOGIC DIAGRAM

D	C	Q^+	$\overline{Q}^+$
0	↑	0	1
1	↑	1	0
X	0	Q	$\overline{Q}$
X	1	Q	Q

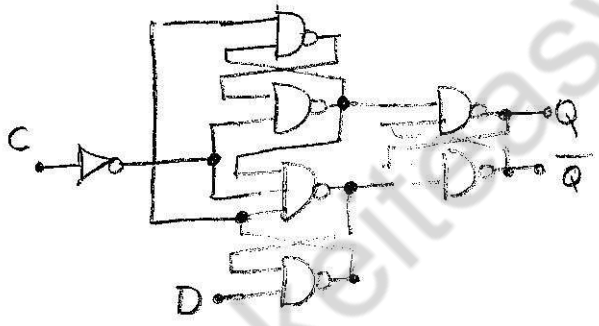
SYMBOL

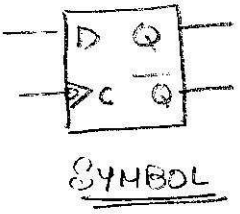
- Positive edge-triggered means the setting or resetting of the flip flop is established during the rising or positive edge of the clock.
- The '↑' symbol in the column indicates that D appears at the output during the positive edge of the clock.
- Let C=0, thus regardless of the 'D' input, the output of NAND gates 2 & 3 is a '1', thus input to the $\bar{S}\,\bar{R}$ latch is $\bar{S}=1=\bar{R}$ and the output remains in the same state
- If D=0, output of gate 4 is '1' & output of gate 1=0

- Let the clock, now transit from 0 to 1, i.e. during the positive edge of the clock, all the three inputs to gate 3 becomes zero and hence output of gate 3 is 0, the output of gate 2 remains a 1.
  Now the input to $\bar{S}\,\bar{R}$ latch is $\bar{S}$=1 & $\bar{R}$=0, thus Q=0 & $\bar{Q}$=1
- Thus during positive edge of the clock the 'D' input is transferred to the output.
- Also the output of the gate 3 is fed back to the input of gate 4. This makes the output of the gate 4 to be '1' and subsequent changes in 'D' input will have no effect.
- Thus, after the occurrence of the positive edge of the clock when D=0 the flip flop is in its 0-state and any changes in 'D' input are not allowed even though clock is 1
- Hence, only on the occurrence of the positive edge of the clock signal does the flip flop respond to 'D' input and changes in 'D' when C=1 is ineffective.

Negative-Edge triggered flip flops:

- Here the occurrence of the negative edge of clock signal causes the output changes with respect to changes in input.
- It can be realized by placing an inverter at the control input of the flip flop
  Logic diagram of positive edge triggered D-flip flop





SYMBOL

D	↓	Q^+	$\bar{Q}^+$
0	↓	0	1
1	↓	1	0
X	0	Q	$\bar{Q}$
X	1	Q	$\bar{Q}$

FUNCTION TABLE

(EXPLAIN ACCORDINGLY)

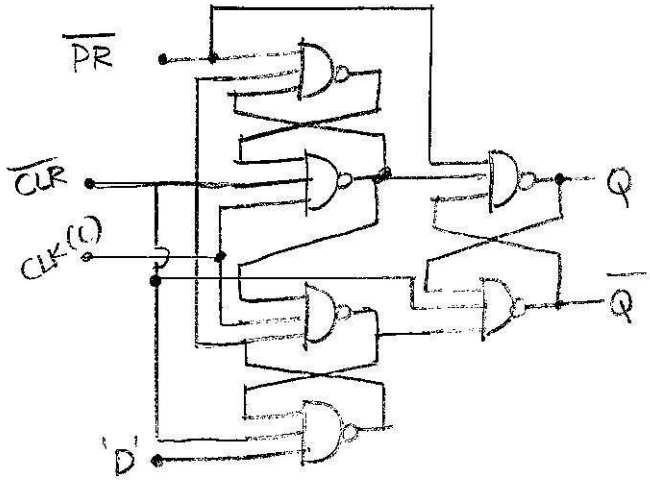
Asynchronous Inputs:

- To provide greater flexibility many flip flops have both synchronous and asynchronous inputs.

The two main asynchronous inputs are

- **PRESET (PR)**
- **CLEAR (CLR)**

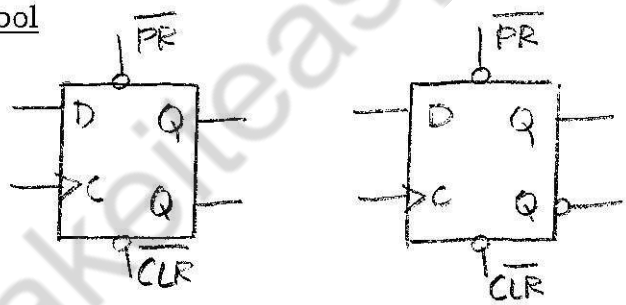
- The asynchronous inputs PR & CLR are used to forcibly set and reset the flip flop respectively independent of the control input.
- The logic diagram of a positive edge triggered D-flip flop with asynchronous preset & clear inputs is shown



$\overline{PR}$	$\overline{CLR}$	D	C	Q^+	$\overline{Q}^+$
0	1	X	X	1	0
1	0	X	X	0	1
0	0	X	X	1*	1*
1	1	0	↑	0	1
1	1	1	↑	1	0
1	1	X	0	Q	$\overline{Q}$
1	1	X	1	Q	$\overline{Q}$

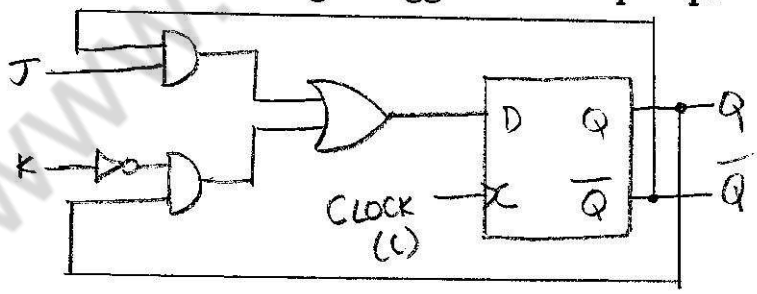
- A logic-0 on the $\overline{PR}$ input line causes the flip flop to enter into 1-state (set) asynchronously
- A logic-0 on the $\overline{CLR}$ input line causes the flip flop to enter into 0-state (reset) asynchronously.
- If $\overline{PR}$=0 & $\overline{CLR}$=1 while C=0, then output of NAND gate becomes 1 & gate 6 becomes 0. Thus $\bar{S}\ \bar{R}$ latch is forced to set, similarly if $\overline{PR}$=1 & $\overline{CLR}$ =0, the $\bar{S}\ \bar{R}$ latch portion of flip flop is forced to be reset.

Logic symbol

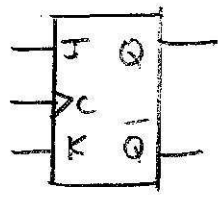


Additional types of edge triggered flip flops

1. Positive Edge Triggered JK flip flops

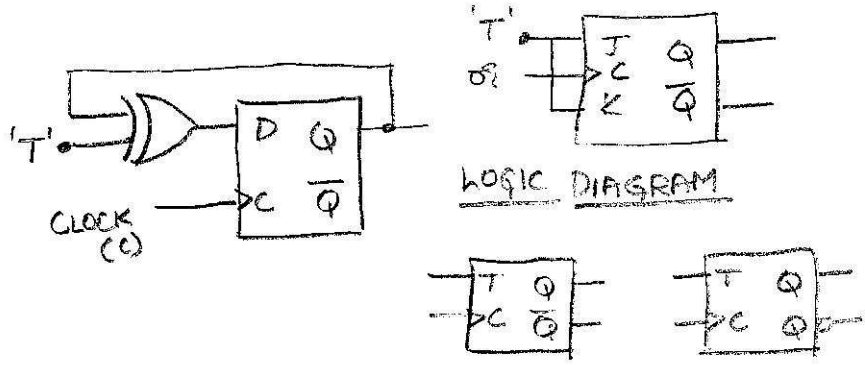


J	K	C	Q^+	$\overline{Q}^+$
0	0	↑	Q	$\overline{Q}$
0	1	↑	0	1
1	0	↑	1	0
1	1	↑	$\overline{Q}$	Q
X	X	0	Q	$\overline{Q}$
X	X	1	Q	$\overline{Q}$

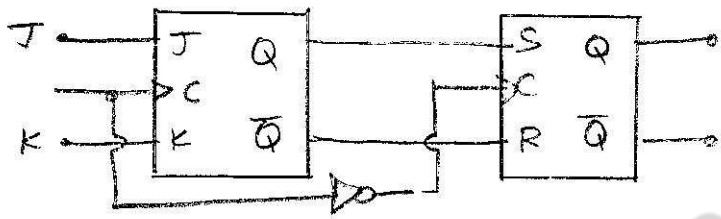


SYMBOL

2. Positive Edge Triggered T flip flop



Master-Slave flip flop with Data lockout:



- To have delayed outputs from flip flops, a master slave flip flop is appropriate. But it is subjected to 0's & 1's catching.
- However, to avoid 0's & 1's catching; the master should respond to the information lines only on one edge of the control signal and to transfer to the slave on the next opposite edge.

- Master – slave flip-flops having this property are said to have data-locked.
- In the above diagram, master is a positive edge triggered JK flip flop and SR latch is used for the slave.
- Here, information is entered into the master during the positive edge of the clock and hat is transferred to the slave during the negative edge.
- This flip-flop is not subjected to 0's and 1's catching.

Characteristic equations:

- A variation in the function table gives the next state table. The next state table shows the value of the next state of the flip flops for each combination of values to the present state of the flip flop.
- The algebraic description of the next state table of a flip flop is called as the characteristic equation of the flip flop and is obtained by constructing a K-Map for Q^+.

Characteristic equation of an SR flip-flop:

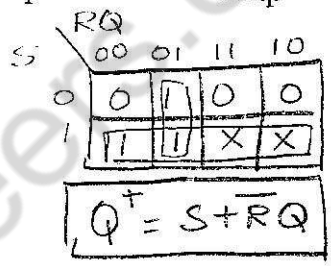
Function table of S-R flip flop

S	R	Q^+
0	0	Q
0	1	0
1	0	1
1	1	X

Next state table of S-R flip flop

S	R	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

K-Map



Characteristic equation of a JK flip-flop:

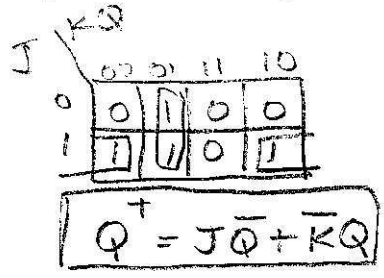
Function table of J-K flip flop

J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	$\bar{Q}$

Next state table of J-K flip flop

J	K	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

K-Map



Characteristic equation of a D flip-flop:

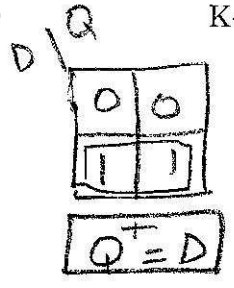
Function table of D flip flop

D	Q^+
0	0
1	1

Next state table of D flip flop

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

K-Map



Characteristic equation of a T flip-flop:

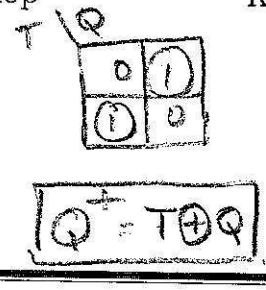
Function table of D flip flop

T	Q^+
0	Q
1	$\bar{Q}$

Next state table of D flip flop

T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

K-Map

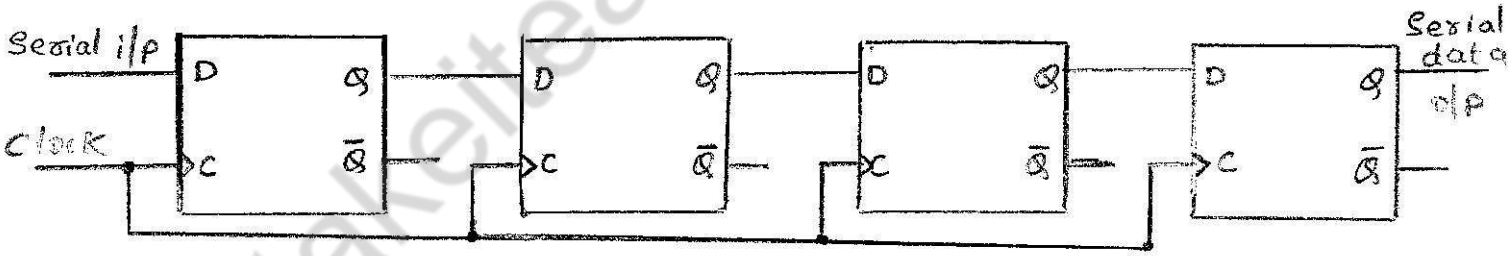


REGISTERS:

- Registers and counters are important applications of clocked flip flops.
- A collection of flip flop in cascade is called as a REGISTER.
- Registers are used to store data in a digital system so as to make it available to the logic elements during the computing process.
- A register can consist of only a finite number of flip flops and each flip flop can store either a '0' or a '1'. Each combination of 1's and 0's stored in a register is referred to as the state or content.
  Thus a cascade of 4 flip flops configured as a register can store 4 bits, i.e. a nibble of data. A 4-bit register can store binary bits from '0000' to'1111'. These are called states. Thus a 4-bit register has 16 possible states.
- Registers that are capable of moving data stored in their flip flops is referred to as a **shift-register.**
- Shift registers that can shift data in both the directions are called as **Bi-directional shift register** and the shift registers that can shift data in only one direction are called as **unidirectional shift register.**
- Shift registers can also be classified based on whether information is entered into and taken out from a register either serially or parallel.
  Accordingly there are four possible registers:
  - Serial in Serial Out (SISO)
  - Serial in Parallel Out (SIPO)
  - Parallel in Parallel Out (PIPO)
  - Parallel in Serial Out (PISO)

4- Bit Serial In Serial Out Unidirectional Register:

4-bit Serial in Serial out (SISO) unidirectional shift register configured using **positive edge triggered D-flip flop** is as shown.

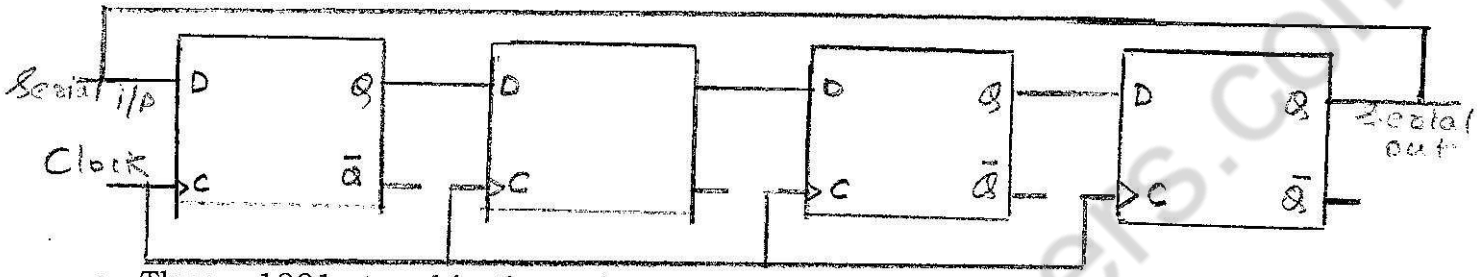


- The Q output of each flip flop is connected to the D-input of the next flip flop.
  The control input, i.e. the clock input of each flip flop is connected together to a common synchronizing signal called the clock.
  Thus, on the occurrence of the positive edge of the clock signal, the content of each flip flop is shifted one position to the right.
- The output from the shift register occurs at the rightmost flip flop on the serial-data-out line.
- The serial data input applied, appears at the serial data output after 4 clock pulses since it consists of 4- flip flops.

- Ex: assume a logic 1 to be applied at serial data in line and before the positive edge of the clock let the content of the register be 0011 thus after the positive edge of the clock signal , the register content is 1001.
- The content of the last flip flop is lost after every clock pulse.

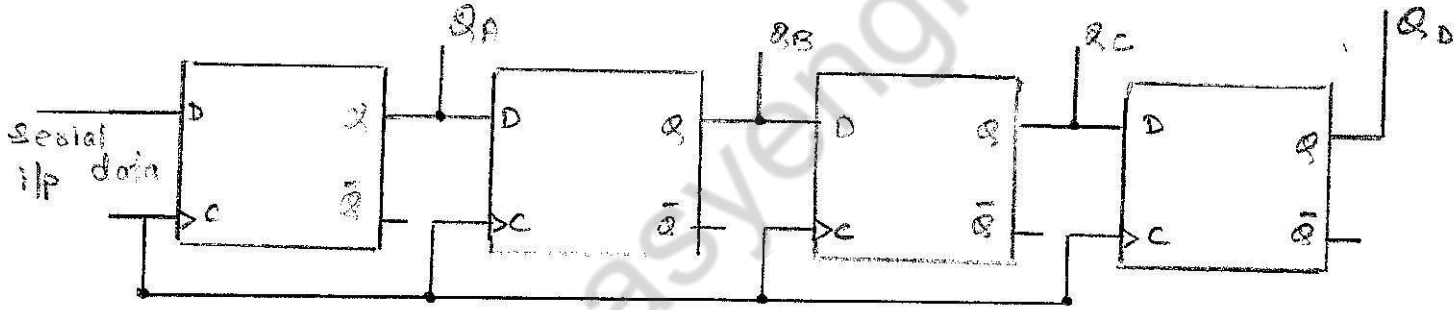
- To store the information within a register, the serial-data out line is connected to the serial-data in line. Such registers are called as **Circular Shift Registers.**

4-bit Circular Shift Register



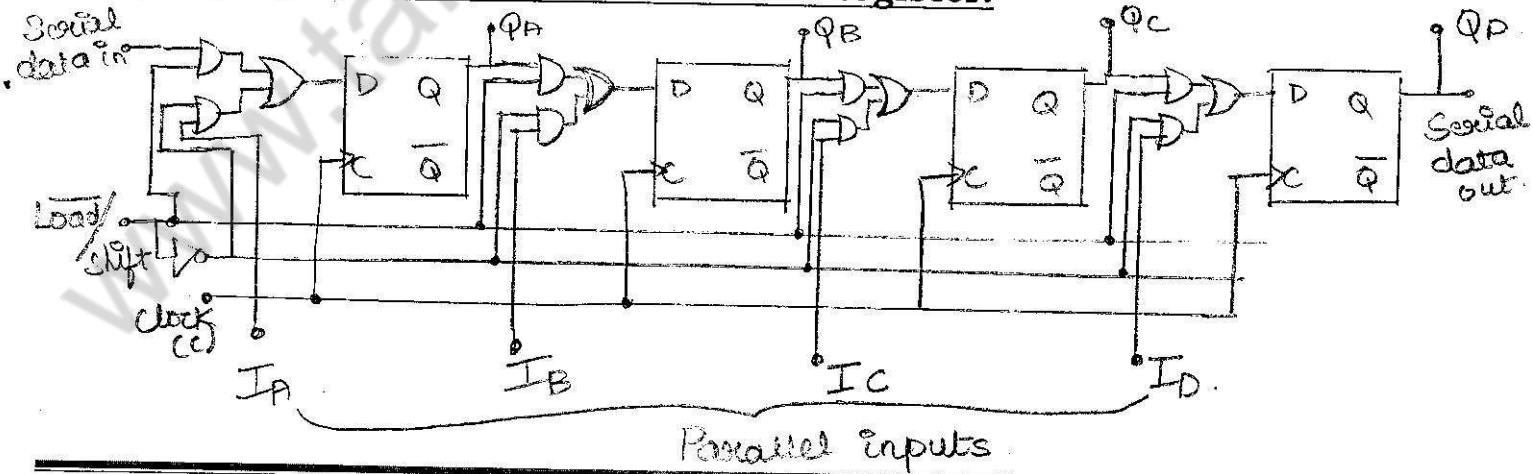
- Thus a 1001 stored in the register becomes 1100 on one shift to the right caused by the application of one clock pulse.

4-Bit Serial In Parallel Out Unidirectional Shift Register:



- A 4-bit SIPO unidirectional shift register is as shown above. Outputs are obtained from each flip-flop.
- Data is entered into the register serially and the output is available as a single entity i.e. parallel output at the flip flop output terminal.

4-bit Parallel-In unidirectional shift register:

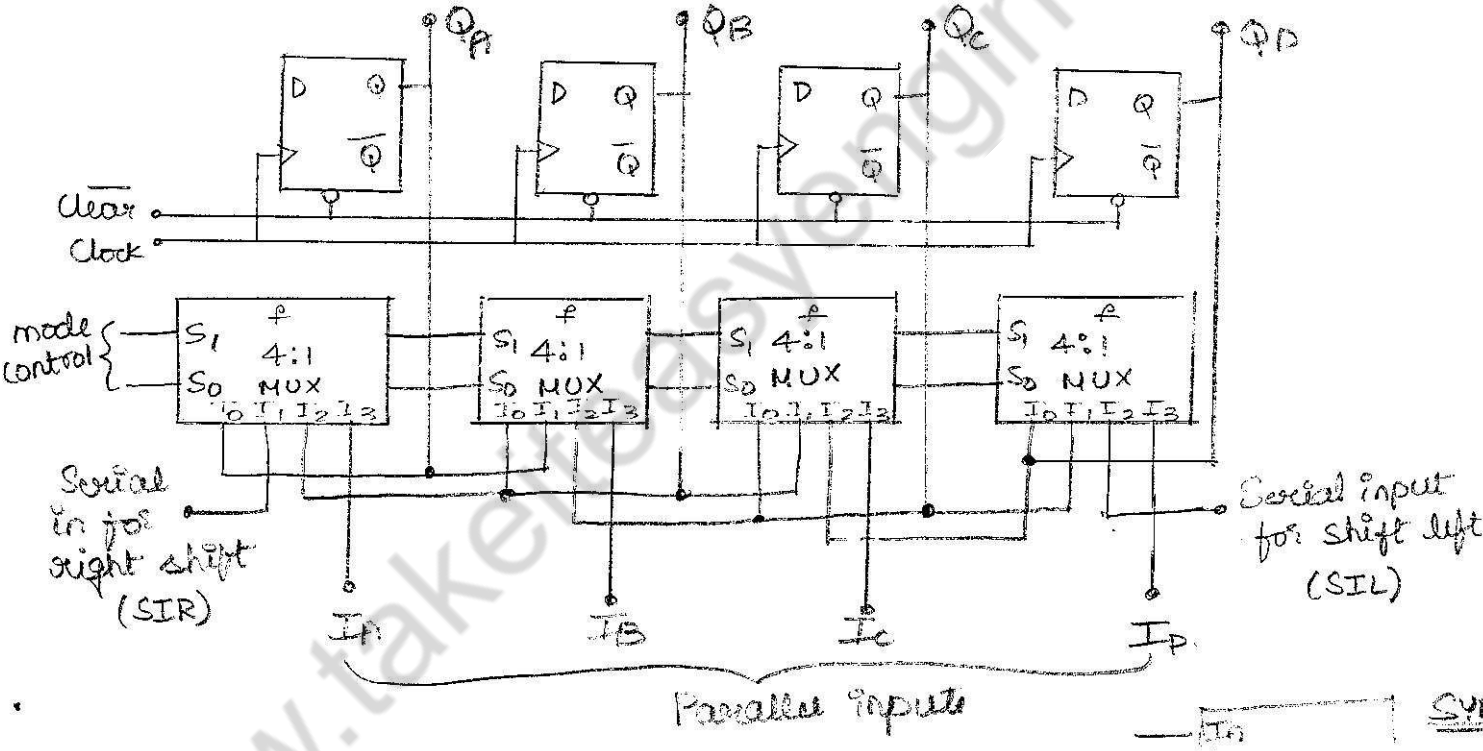


- The Parallel-In unidirectional shift register is as shown above. The operation of the register is controlled by the $\overline{\text{LOAD}}$ /**SHIFT** line.
- When the $\overline{\text{LOAD}}$ /SHIFT line is at logic'0', the lower AND gates are enabled and the signal on the parallel data input lines are I_A, I_B, I_C, I_D are transferred into the register upon the occurrence of a positive edge of the clock.
- When a logic'1' is applied to the $\overline{\text{LOAD}}$ /SHIFT line, the upper AND gates are enabled and it works as a unidirectional shift register.
- This register can also be configured as SISO and SIPO.

BIDIRECTIONAL SHIFT REGISTER:

UNIVERSAL SHIFT REGISTER:

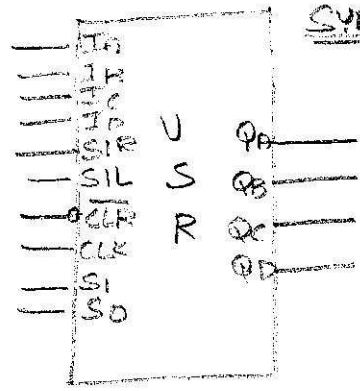
- A universal shift register is one which is capable of shifting its contents in both directions either left or right depending upon the signals on the control lines.
- A universal shift register also has the capability to accept both serial and parallel inputs as well as capability of serial and parallel output.
- The operation of a shift register is based on the **mode selection table** as follows:



MODE SELECTION TABLE

S_1 S_0	Data line selected	Operation
0 0	I_0	Hold
0 1	I_1	shift right
1 0	I_2	shift left
1 1	I_3	Parallel load

SYMBOL



- The value on the select lines (mode control lines) of the multiplexer, decide the various operations of the shift register like the hold, shift right, shift left and parallel load.
- Each of these operations occurs during the positive edge of the clock since it is constructed using positive edge triggered D-flip flop.
- **$\overline{\text{CLEAR}}$:** This is an active low signal, when a logic-0 is applied; it clears the contents of the register asynchronously (independent of the clock).

- When **S_1S_0 =00**, I_0 of the MUX is selected. According to mode selection table, the operation that is performed is the **hold operation** .
  The Q output of each flip flop gets connected to the D inputs and upon the application of the clock pulse; the data appears at the output lines.

- When **S_1S_0 =01**, I_1 of the MUX is selected. According to mode selection table, the operation that is performed is the **shift right operation**.
  The input to the leftmost D-flip flop is applied at the serial input for shift right line(SIR).
  The input to the 2nd D-flip flop is the output of the 1st D-flip flop, hence I_1 of the second MUX is connected to Q_A. Similarly output of the 2nd D-flip flop is the input o the 3rd D-flip flop and output of the 3rd D-flip flop is the input o the 4th D-flip flop.
  During the positive-edge of the clock, the contents of the register are shifted one position to the right.

- When **S_1S_0 =10**, I_2 of the MUX is selected. According to mode selection table, **shift right operation** is performed.
  The serial input for the left shift operation is applied to the rightmost flip flop at serial input for left shift operation (SIL).
  During the positive-edge of the clock, the contents of the register are shifted one position to the left.

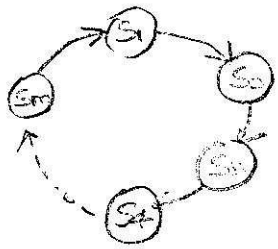
- When **S_1S_0 =11**, I_3 of the MUX is selected. **Parallel load** is the operation performed. The parallel data inputs are applied at I_3 input of each MUX, which appears at the D input of each flip flop respectively.

COUNTERS

- Counter is a collection of flip flops configured to produce a specified output pattern sequence; hence it is also called as a **pattern generator**.
  Each output sequence stored in the flip flop is referred to as the state of the counter.

- The total number of states is called as its **modulus.** Thus if a counter has 'm' states, it is called as a modulus-m-counter **or mod-m- counter.**

- A counter that counts from 0000 to 1001 has 10 states and thus is called as Mod-10-counter. State diagram is used to describe the operation of a counter. State diagram for a mod-m- counter is:

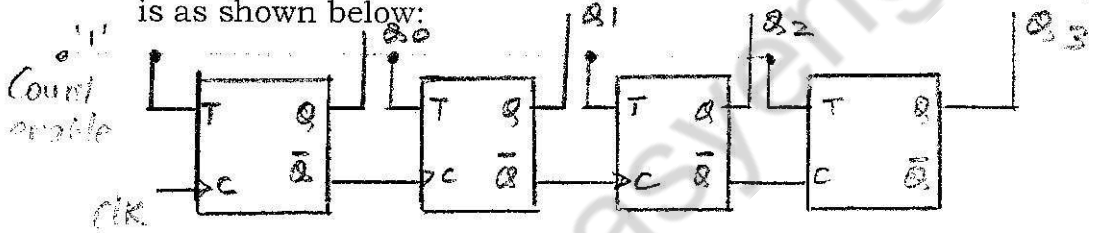


BINARY COUNTER:

- Binary counters output a binary number sequence.
- A cascade of 'n'flip flops can be used to configure a counter up to a modulus of 2^n. Thus a cascade of 4 flip flops can be used to configure a counter up to modulus 16.
- A binary up counter counts from $0_{(10)}$ to $(2^n - 1)_{(10)}$. Once the terminal count (max count value) is reached, the counting sequence is repeated.
- A binary down counter counts from $(2^n - 1)_{(10)}$ to $0_{(10)}$.

4-BIT BINARY RIPPLE COUNTER(ASYNCHRONOUS COUNTERS)

- A 4-bit binary ripple up counter configured using positive edge triggered T-flip flop is as shown below:



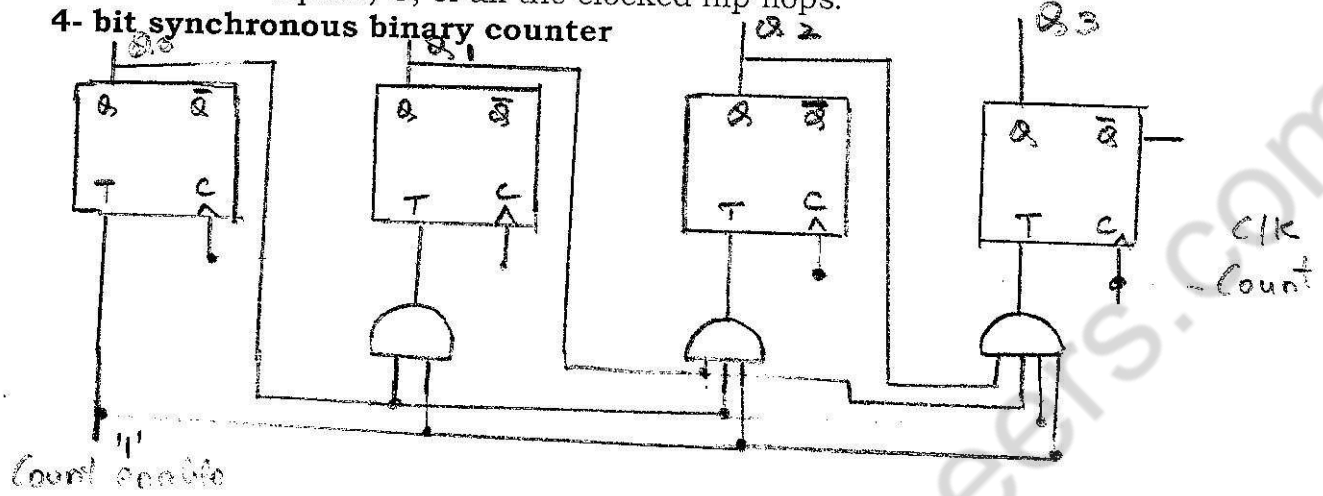
- The input to the counter is a count enable signal and a series of count pulses applied to the flip flop. Thus as long as the count enable signal is at logic-1, the Q_0 output flip flop toggles on each positive edge of the count pulse.
- The clock input of the remaining flip flops is connected to the $\bar{Q}$ output of the previous flip flop. These flip flops change their state during 0 to 1 transition of $\bar{Q}$ output, which corresponds to 1 to 0 transition of Q output.
- Since it is a 4-bit up counter, its modulus is 2^4=16 and its counting sequence is from $0000_{(2)}$ to $1111_{(2)}$. The output of the counter appears at the Q output terminals of the 4- flip flops.
- The binary counter is known as a ripple counter since a change in state of the previous flip flop is used to toggle the flip flop next to it.
- **Ripple counters** are also known as **asynchronous counters** since clock pulse is not applied simultaneously to all the flip flops.
- There is a propagation delay between the input and output of a flip flop, the delay is maximum when all the flip flops need to toggle from 1111 to 0000 state. Thus for an n-stage binary ripple counter, the worst –case settling time is $\mathbf{n * t_{pd}}$ where $\mathbf{t_{pd}}$- propagation time delay associated with each flip flop.

Counting sequence, Timing diagram (Refer class notes)

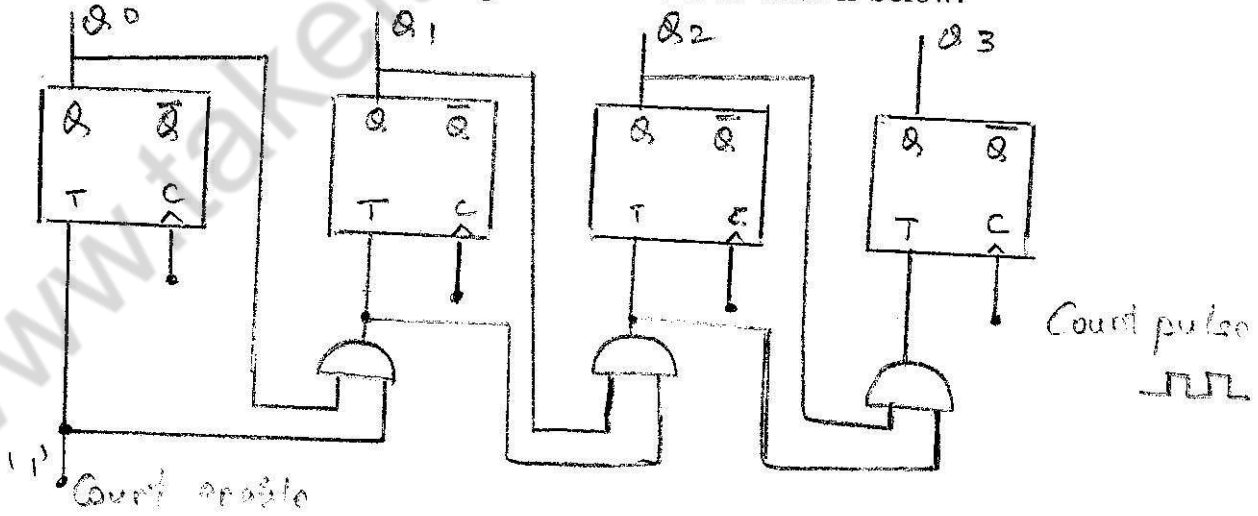
SYNCHRONOUS BINARY COUNTER

- The settling time problem associated with ripple counters is overcome in synchronous counter. In synchronous counters, the count pulses are applied directly to the control inputs, C, of all the clocked flip flops.

4- bit synchronous binary counter



- Observing the counting sequence, we find that Q_0 toggles with every clock pulse. The other outputs toggle whenever all the lower order flip flops outputs are at 1. Thus all the lower order flip flop's output can be **AND**ed to enable the toggle of a given flip flop.
- When count enable line is at logic'1', the AND gates outputs will be '1' only when all the previous flip flops outputs are at 1.This output will in turn be fed to the T-input.
  The flip flops whose T-input is at logic 1, toggles at the next clock pulse.
- The main drawback of this counter is :As the number of stages increase the number of inputs to the AND gate also increases.
- Thus making use of the fact that ANDed outputs of all previous flip flops are available at the output of each AND gate, the gating can be modified to keep the number of inputs to the AND gates constant as shown below.



4-BIT SYNCHRONOUS COUNTER WITH PARALLEL LOAD FACILITY

The logic symbol of a 4-bit synchronous binary counter with parallel load facility is as shown:

- The C_0 is the carry output which goes high during 1111 to 0000 transition.
- A logic-1 at the load input would load the count D_0, D_1, D_2, D_3 into the counter.
- A logic-1 at the count input would enable Up counting.
- The C_0 output is used to cascade several 4-bit Counters to form higher bit counter.

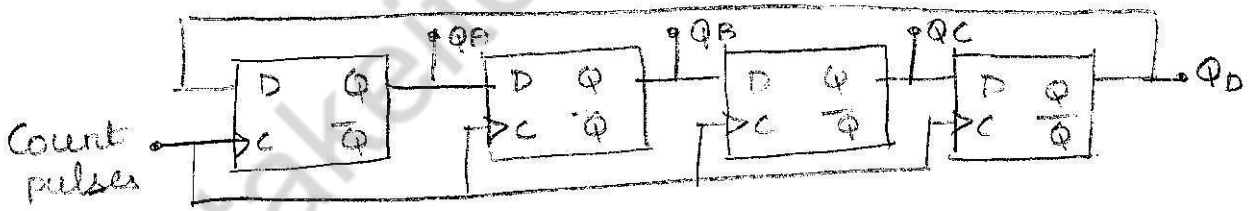
ms.

COUNTERS BASED ON SHIFT REGISTER

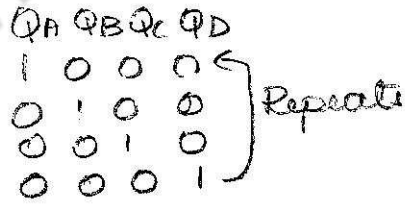
- The overall operation of a digital system is divided into a sequence of time-periods.
- There are two counters based on shift register.
  1. Ring counter 2. Johnson counter

RING COUNTER

- A ring counter is a circular shift register which is initialized such that only one of its flip flop is in the 1-state; while the others are in their 0-state.
- Upon the occurrence of each count pulse, the 1 is shifted around the register.
- A ring counter consisting of 'n' flip flops has only 'n' states.
- A mod-4 ring counter configured using a circular shift register is as shown:



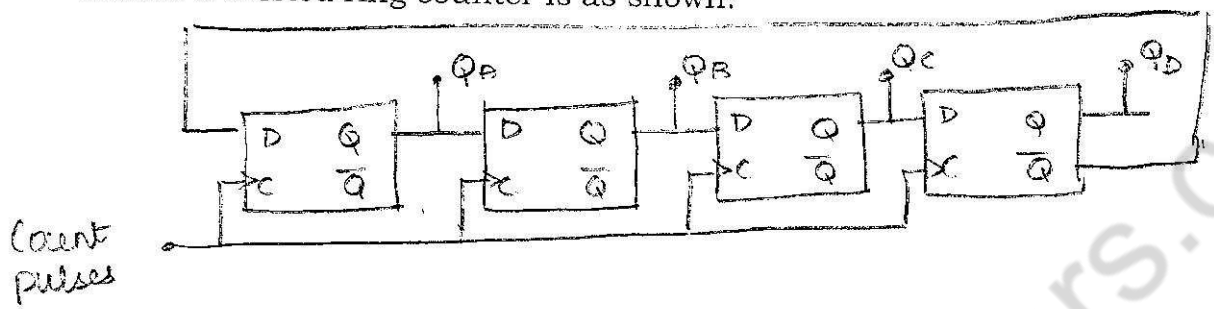
Initializing the counter to an initial value $Q_A Q_B Q_C Q_D = 1000$



Thus it can be observed that a ring counter produces a decoded output.

JOHNSON COUNTER: or TWISTED RING COUNTER or SWITCH TAIL COUNTER

- A variation of the ring counter is the switch-tail counter as shown, here the $\bar{Q}$ output of the last flip flop is provided as input to the first flip flop.
- An n-stage Johnson counter has 2n states.
- A two- input AND gate is sufficient to identify the decoder logic states.
- A mod-8 twisted ring counter is as shown:



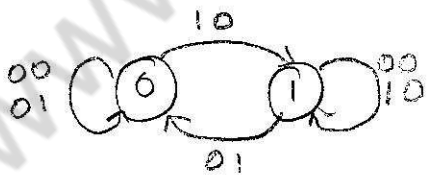
Sequence of mod-8 Johnson counter:

Q_A	Q_B	Q_C	Q_D
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

EXCITATION TABLE OF VARIOUS FLIP FLOPS:

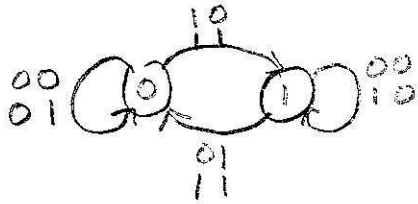
① SR Flipflop

Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0



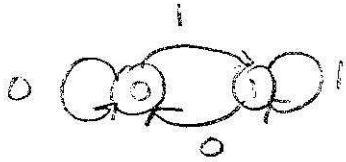
② JK Flipflop

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



(3) D Flip flop

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1



(4) T Flip flop

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

DESIGN OF SYNCHRONOUS COUNTER:

General procedure:

1. Obtain the state diagram from the given counting sequence.
2. Determine the number of flip flops.
3. Write the excitation table of the flip flop.
4. Write the excitation table of the mod-m counter.
5. Use K-map to find the expression for corresponding inputs to flip flops.
6. Draw the counter circuit by using flip flops and required gates from the above obtained Boolean expression.

MODULE-4 INTRODUCTION TO VERILOG

Introduction to VERILOG: Structure of Verilog Module, Operators, Data Types, Styles of description

VERILOG Data Flow Description: Highlights of Data Flow description, Structure of Data Flow Description

Why HDL?

What is Hardware description language (HDL):

HDL is a computer aided design (CAD) tool for the modern digital design and synthesis of digital systems.

Need for HDL

The advancement in the semiconductor technology, the power and complexity of digital systems has increased. Due to this, such digital systems cannot be realized using discrete integrated circuits (IC's).

Complex digital systems can be realized using high-density programmable chips such as application specific integrated circuits (ASIC's) and field programmable gate arrays (FPGA's). To design such systems, we require sophisticated CAD tool such as HDL.

HDL is used by designer to describe the system in a computer language that is similar to other software Language like C. Debugging the design is easy, since HDL package implement simulators and test benches. The two widely used Hardware description languages are VHDL and Verilog

A Brief History of Verilog

Evolution of Verilog

- In 1983, a company called **Gateway design Automation** developed a hardware-description language for its newly introduced logic simulator verilog_XL
- **Gateway** was bought by **cadence** in 1989 & **cadence** made Verilog available as public domain.
- In December 1995, Verilog HDL became IEEE standard 1364-1995.
- The language presently is maintained by the Open Verilog international (OVI) organization.
- Verilog code structure is based on C software language.

Structure of Verilog Module

The Verilog module has a declaration and a body. In the declaration, name, input and outputs of the modules are listed. The body shows the relationship between the input and the outputs with help of signal assignment statements.

The syntax of the Verilog module is shown below

```
module name of module(port_list);
  // declaration:
 input , output, reg, wire, parameter, inout;
functions, tasks;
// statements or body
Initial statement
always statement
module instantiation
continuous assignment
endmodule
```

The example program is halfadder

```
module halfadder (a,b,sum,carry);
input a;
input b;
output sum;
output carry;
 assign sum=a ^b; // statement 1
assign carry=a &b; // statement2
end module
```

- Verilog is case sensitive. Halfadder and halfadder are two different modules in verilog. The declaration starts with predefined word **module**.
- The name of the module should start with alphabetical letter and can include special character underscore (_). It is user selected.
- Semicolon (;) is a line separator. The order in which the inputs, &outputs and their declarations are written is irrelevant.
- "=" is assignment operator, and symbols ^ and & are used for: "xor" and "and" respectively.
- The doubles slashes (//) signal a comment command or /*……*/ the pair is used to write a comment of any length.
- The program ends with predefined word **endmodule**

Verilog ports

input: the port is only an input port. In any assignment statement, the port should appear only on the right hand side of the assignment statement.(i.e., port is read.)

output: the port is an output port. In contrast to VHDL, the Verilog output port can appear on either side of the assignment statement.

inout: this port can be used as both an input and output. The inout port represents a bidirectional bus.

1.4 Operators

HDL has a extensive list of operators. Operator performs a wide variety of functions.

Functions classified

1. Logical operators such as and, or, nand, nor, xor, xnor and not
2. Relational operators: to express the relation between objects. The operators include =, /=, <, <=, >and >=.
3. Arithmetic operators: such as +, -, * and division.

4. Shifts operators: To move the bits of an objects in a certain direction such as right or left sll, srl, sla, sra, rol and ror .

Logical operators

These operator performs Logical operations, such as and, or, nand, nor, xor, xnor, and not. The operation can be on two operands or on a single operand. The operand can be single bit or multiple bits.

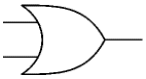
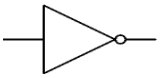
Verilog operator (bitwise)	Equivalent logic	Operand type	Result type
&		Bit	Bit
|		Bit	Bit
~(&)		Bit	Bit
~(|)		Bit	Bit
^		Bit	Bit
~^		Bit	Bit
~		Bit	Bit

Table 1.1 logical operators.

Verilog logical operators

Verilog logical operator can be classified as Bitwise, Boolean logical and reduction logical operators.

The bitwise operators are similar to VHDL logical operators. They operate on the corresponding bits of two operands. These are shown in table1.1

Example Z= x & y, if x=1011 and y=1010 are 4-bit signals then z=1010 is logical **and operation** of x and y.

Boolean operators operate on the two operands. The result is Boolean true (1) or false (0). These are shown in table 1.2

Example for z= x && y, if x=1011 and y=0001 then Z=1, 2nd case if x=1010 and y=0101 then z=0;

For z! =x if x=1111 then z=0;

Operators	**Operation**	**Number of operands**
&&	AND	two
||	OR	two

Table 1.2 Boolean operators

Reduction operators: These operators operate on a single operand. The result is Boolean.

Example y=&x, if x=1010 then y= (1&0&1&0) =0

Operators	**Operation**	**Number of operands**
&	Reduction AND	One
|	Reduction OR	One
~(&)	Reduction NAND	One
~(|)	Reduction NOR	One
^	Reduction XOR	One
~(^)	Reduction XNOR	One
!	Negation	One

Table 1.3 Verilog Reduction logical operators

1.4.2 Relational operators

These are implemented to compare the values of two objects. The result is false (0) or true (1).

Verilog Relational operators

Verilog has a set of Relational operator similar to VHDL. Returns Boolean values false (0) or true (1).

The result can also be of type unknown (X) when any of the operand include don't care or unknown (X) or high impedance. Table 1.4 shows the list of Verilog Relational operators

Example: if (A==B), if the values of A or B contains one or more don't care or Z bits. The value of the expression is unknown.

If A is equal to B, then result of the expression (A==B) is true (1).

If A is not equal to B, then result of the expression (A==B) is false (0).

Operators	Description	Result type
==	Equality	0,1,X
!=	Inequality	0,1,X
===	Equality Inclusive	0,1
!==	Inequality Inclusive	0,1
<	Less than	0,1,X
<=	Less than equal to	0,1,X
>	Greater than	0,1,X
>=	Greater than equal to	0,1,X

Table 1.5 List of Verilog Relational operators

1.4.3 Arithmetic operators

Arithmetic operators can perform a wide variety of operation, such as addition, subtraction, multiplication and division.

Verilog Arithmetic operators

It is not an extensive type-oriented language. Example y :=(A *B) calculates the values of y as the product of A times B. Table 1.7 shows the Verilog arithmetic operator

Operators	Description	A or B type	Y type
+	Addition A+B	A numeric B numeric	Numeric
-	Subtraction A-B	A numeric B numeric	Numeric
*	Multiplication A?B	A numeric B numeric	Numeric
/	division A/B	A numeric B numeric	Numeric

%	Modulus A%B	A numeric, not real B numeric, not real	Numeric, not real
**	Exponent A**B	A numeric B numeric	Numeric
{,}	Concatenation {A,B}	A numeric, or array B numeric, or array	Same as A

Table1.7 Verilog arithmetic operator

1.4.4 Shift and Rotate operators

A shift left represents multiplication by two, and a shift left represents division by two.

b). Verilog Shift and Rotate operators

It has basic shift operators. These are unary operators i.e., operate on single operand. Example if A=1110, is a 4 bit vectors table 1.8 shows the Verilog shift and rotate operators.

Operation	**Description**	**Operand A Before shift**	**Operand A After shift**
A <<1	Shift A one position left logical	1110	110X
A <<2	Shift A two position left logical	1110	10XX
A >>1	Shift A one position right logical	1110	X111
A >> 2	Shift A two position right logical	1110	XX11

Table 1.8 the Verilog shift operators

Data types

The data or operands used in the language must have several types to match the need for describing the hardware.

2Verilog Data types

There are different types of Verilog data types. Namely

1. Nets
2. Registers
3. Vectors
4. Integer
5. Real
6. Parameters
7. Array

Nets:

These are declared by the predefined word **"wire".** Nets values are change continuously by the circuits that are driving them. A wire represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block. Verilog supports 4 values for nets.

Value	Net Definition	Reg
0	Logic 0(false)	Logic 0
1	Logic 1(true)	Logic 1
X	Unknown	Unknown
Z	High impedance	High impedance

Eg. Wire sum; // statement declares a net by name sum.

Wire s1=1'b0; // this statement declares a net by the name of s1; it is initial value 1 bit with value 0.

Registers: Registers store values until they are updated. They are data storage elements. Declared by the predefined word **"reg"** Verilog supports 4 values for registers. As shown in above table.

Eg reg sum_total; // declares a register by the name sum_total.

Vectors:

These are multiple bits. A reg or net can be declared as a vector. Vectors are declared by brackets [].

Eg. Wire [3:0] a=4'b1010;

Reg [7:0] total =8'd12;

Integer: declared by the predefined word **"integer".** Integers are general-purpose variables. For synthesis they are used mainly loops-indices, parameters, and constants.

Eg. Integer no_bits;//The above statement declares no_bits as an integer.

Real:

Real (floating point) numbers are declared with the predefined word **"real".** Examples of real values are 2.4, 56.3 5e12.

Eg. Real weight; // the statement declares the register weight as real.

Parameters:

It represents global constants. Declared by the predefined word **"parameter"**

Eg. Module comp_genr (x, y, xgty, xlty, xeqy);
Parameter N=3;
Input [n:0] x,y;
Output xgty, xlty, xeqy;
Wire [N:0] sum, xb;

Array: there is no predefined word **"array".** Registers and integers can be used as arrays.
Parameter N=4;
Parameter M=3;
Reg signed [M: 0] carry [0:N]
Reg [M: 0] b [0: N];
Integer sum [0: N];
The above statement declares an array by the name sum. It has 5 elements, and each element is an integer type.
array carry has 5 elements, and each elements is 4bits. They are in 2'S complement form
The array b has 5 elements, each element is 4 bits. The value of each bit can be 0, 1, X or Z;

1.6 Style (Types) of Descriptions

1.6.1 Behavioral Descriptions

This models the system as to how the outputs behave with inputs.

The definition of Behavioral Description is one where architecture (VHDL) or module (Verilog) includes the predefined word process (VHDL) or always or initial (Verilog).

This description is considered pure behavioral if it does not contain any other features from other styles.

Listings refer class notes.

VHDL Behavioral Description

Verilog Behavioral Description

1.6.2 Structural Descriptions

This model the system as components or gates, this description is defined by the presence of the Keyword component in the architecture (VHDL) or gates construct such as "and", "or", or "not" in the module (Verilog).

If the VHDL architecture or the Verilog module consists of only components or gates; this style is coined as pure structural.

Listings refer class notes.

VHDL structural Description

Verilog structural Description

1.6.3 Dataflow Descriptions

It describes how the systems signals flow from the input to the output. The dataflow statements are concurrent; their execution is controlled by events.

Usually, the description is done by writing the Boolean function of the outputs. It should not include any of keywords that identify behavioral, structural, or switch level descriptions.

Listings refer class notes.

VHDL dataflow Description

Verilog dataflow Description

1.6.4 Switch level Descriptions

It is the lowest level of description. The system is described using switches or transistors. The Verilog keywords nmos, pmos, cmos, tran, or tranifo describe the system. VHDL does not have built in switch level primitives, we are constructing packages to include such primitives and attach them to the VHDL module.

Listings refer class notes.

VHDL switch level Description

Verilog switch level Description

1.6.5 Mixed-type Descriptions

It uses more than type. Here we may describe some parts of the system using one type and other parts using another type. Example of Mixed-type Description using both dataflow and behavioral style is explained in the listing.

Listings refer class notes.

VHDL mixed-type Description

Verilog mixed type Description

1.6.6 Mixed-language Descriptions

It is newly added tool for HDL descriptions. The user can write a module in one language (VHDL or Verilog) and invoke or import a construct (entity or module) written in the other language.

Listings refer class notes.

VHDL mixed language Description

Verilog mixed language Description

VERILOG DATA FLOW DESCRIPTION:

Highlights of Data Flow description

Dataflow is a type of hardware description which shows how the signal flows from system inputs to outputs. It uses signal assignment statements which are executed concurrently when an event occurs on the signals on the right side of the statement.

In HDL language, programming is carried out two standard methods.

1. Concurrent program execution: In this method of program execution, all the statements within the program are executed simultaneously.

   The above gate network has two inputs A and B, two outputs Y1 and Y2. The outputs will get evaluated simultaneously whenever an event occurs on either of the inputs A or B or both, assuming the propagation delay of both the gates to be same.

   In order to describe the above hardware we need concurrent program execution where the outputs are updated whenever an event occurs on its inputs, irrespective of the order of statements. All combinational circuits need this style of execution for accurate description of the hardware.

2. Sequential program execution: In this method all the statements are executed sequentially in the order of their appearance.

   An example of hardware that requires this method of program execution is a flip-flop. The data given at D input will be transferred to the output only after the rising or falling edge of the clock. All sequential circuits like flip-flops, counters, registers require this method of program execution.

Structure of Data-Flow Description

A dataflow model specifies the functionality of the system without explicitly specifying its structure. It specifies how the system's signal flow from inputs to the outputs. The description is usually done by writing the Boolean functions of the outputs.

The dataflow statements are concurrent and their execution is controlled by events.
EVENT: An event is a change in the value of a signal, such as a change from 0 to 1 or 1 to 0.

Dataflow description is modeled using **concurrent signal assignment statements** (VHDL) and **continuous signal assignment statements** (Verilog).

Example program1:

VHDL dataflow description	**Verilog dataflow description**
entity system is Port (I1, I2 : in bit ; O1, O2 : out bit) ; end; architecture dtf of system is	module system (I1, I2, O1,O2); input I1, I2; output O1, O2;

begin
 	
O1 <= I1 and I2 ; --st1
 	
O2 <= I1 xor I2 ; --st2
 	
end dtf;
	assign O1 = I1 & I2; //st1
	
	assign O2 = I1 ^ I2; //st2
	
	end module

Above example shows HDL code, describing a system using dataflow description. The entity (module) name is system. I1 and I2 are the two inputs and O1 and O2 are the two outputs. St1 and st2 are signal assignment statements which assigns value to the outputs O1 and O2.

SIGNAL DECLARATION:

Input and output signals are declared in the entity (module) as ports. Intermediate Signals (other than input and output signals) are declared using the predefined word **signal** in VHDL and **wire** in Verilog as shown in the below example. In Verilog signals are declared using **reg** when the value of the signal needs to be stored.

```
signal s1, s2 :  bit;   --VHDL
wire s1, s2;            // Verilog
```

SIGNAL ASSIGNMENT STATEMENTS:

A signal assignment statement is used to assign a value to a signal. The left hand side of the statement should be declared as a signal. The right hand side can be a signal, a variable, or a constant. **'<='** is a signal assignment operator in VHDL and in verilog predefined word **assign** is used.

Execution of signal assignment statement has two phases. In the above example of system, assume that an event at T_0 occurs on either signal I1 or I2. This event changes the value of I1 from 0 to 1 and also the value of I2 from 0 to 1.

1. **Calculation**: The value of O1 is calculated using the current values of I1 and I2 at time T0. The value 1 and 1=1 is calculated. This is not yet assigned to O1.
2. **Assignment:** The calculated value 1 is assigned to O1 after a delay time. The delay time can be implicitly or explicitly specified. If no delay time is specified, the HDL uses a default, small delay of Δ (delta) seconds.

Continuous signal assignment statements:

A continuous assignment statement is the most basic statement in Verilog dataflow modeling. It is used to drive a value onto a net or assigns a value to a net. It uses the keyword **assign**. It has the following form,

assign LHS_target = RHS_Expression;

Example: wire[3:0] Z, preset, clr;
assign Z = preset & clr;

The target of the continuous assignment is Z and the right hand expression is (preset & clr). The continuous assignment statement executes whenever an event occurs on an operand on the right hand side of the expression, it is evaluated and assigned to the target.

Concurrent signal assignment statements:

One of the primary mechanisms for dataflow modeling in VHDL is the **concurrent signal assignment statement**. It has the following form,

LHS_target <= RHS_expression;

The value computed by the RHS_expression is assigned to the LHS_target. '<=' is called the signal assignment oprator.

Example: C <= A and B;

Right hand side expression A and B is computed and the value is assigned to the LHS_target C.

In HDL dataflow descriptions, concurrent signal assignment statements constitute the major part. In exampleprogram1 both st1 and st2 are concurrent statements. For the execution of concurrent statement to start, an event on the right hand side of the statement has to occur. An event is a change in the value of the signal or a variable. If an event occurs on more than one statement, then all those statements regardless of their order in the architecture (module) are executed concurrently (simultaneously).

CONSTANT DECLARATION AND ASSIGNMENT STATEMENTS:

The value of a constant is constant within the segment of the program where it is visible. A constant in VHDL can be declared using the predefined word ***constant*** and in Verilog it is declared by its type, such as ***time*** or ***integer.***

Ex: *constant* **period**: *time* ; --VHDL
time period ; //verilog

To assign a value to the constant we use assignment operator := in VHDL or = in verilog.

Ex: period := 100ns; --VHDL
Period = 100; //verilog

The above example assigns a value of 100 nanoseconds to the constant period which was declared above. In verilog there are no explicit units of time. 100 means 100 simulation screen time units. The declaration and assignment can be combined in one statement as:

constant **period**: *time* := 100ns; --VHDL
time period = 100 ; //verilog

Conditional signal assignment statement:

The conditional signal assignment statement selects one out of different values for the target signal based on the specified condition.

The typical syntax for this statement is as follows.

target_signal <= *expression1* **when** *boolean_condition1* **else**
expression2 **when** *boolean_condition2* **else**
.
.
expression n;

When there is an event on any of the operands present in the boolean_condition or the expression, the execution of when-else starts. The boolean condition1 is evaluated first. If the result is true then expression1 is assigned to the target signal. If the result is false, next condition2 is checked. If the result is true then expression2 is assigned to the target or else

condition3 is cheked and so on. If no conditions are true then expression n is assigned to the target_signal.

The target_signal will receive the value of the first expression whose boolean condition is TRUE. If more than one condition is true, the value of the first condition that is TRUE will be assigned.

Though the when-else statement is a concurrent statement, within the body the execution is sequential. Hence the first condition gets the highest priority. This special feature of when-else can be used to describe "priority encoders".

Example:
```
Z <= A when s0='0' and s1='0' else
     B when s0='0' and s1='1' else
     C when s0='1' and s1='0' else
     D when s0='1' and s1='1';
```

In this example, the statement is executed any time an event occurs on A, B, C, D, s0 or s1. The first condition(s0=0 and s1=0) is checked, if false, the second condition is checked and so on and when the condition is true the corresponding value is assigned to Z.

The conditional signal assignment will be executed if any of the signals in the conditions or expression change.

```
entity MUX is
    port   (A, B, C, D: in std_logic;
            SEL: in std_logic_vector (1 down to 0);
            Z : out std_logic );
    end MUX;
architecture MUX41 of MUX is
begin
    Z <= A when SEL = "00" else
         B when SEL = "01" else
         C when SEL = "10" else
         D;
end MUX41;
```

Selected signal assignment statement:

The selected signal assignment is another concurrent statement in VHDL. It provides selective signal assignment. The syntax is as follows,

```
with choice_expression select
        target_name <= expression1 when choices1,
                       expression2 when choices2,
                       .
                       .
                       expression n when choices n;
```

Whenever an event occurs on a signal in the choice_expression or any expression1, expression2…, the statement gets executed. The choice_expression gets evaluated first and then this value is compared with all the choices simultaneously. When the value matches with any of the choices, the corresponding expression is assigned to the target_signal.

For example if the value of the choice_expression matches with choice1, then expression1 is assigned to the target_signal.

* The choice expression must contain atleast one signal because an EVENT can occur only on a signal. For example if the choice_expression is Y then Y must be a signal. If choice_expression is X+Y, then either of X or Y must be a signal.

The following rules must be followed for choices:

1. All possible values of the choice expression must be covered by the choices **exactly once**. Values not covered explicitly may be covered by an "**others**" choice.
2. The choices must be static expressions. Ex: 5, 5+6 or w+u where w and u are constants which are already declared.
3. The choices must be mutually exclusive.

Example 2.1: Data-flow description of a half adder

Truth table
Logic diagram
Logic symbol
VHDL and Verilog code
Simulation waveform

ASSIGNING A DELAY TIME TO THE SIGNAL-ASSIGNMENT STATEMENT

To assign a delay time to a signal-assignment statement, we use the predefined word **after** in VHDL or # (delay time) in Verilog.

```
s1 <= sel and b after 10ns;      --VHDL for 10ns delay
assign #10 s1 = sel and b;       //Verilog for 10 screen units delay
```

Note: In Verilog, we cannot specify the units of delay time. The delay is in simulation screen unit time.

Example 2.2: 2×1 Multiplexer with Active Low Enable

Truth table
Logic diagram
Logic symbol
VHDL and Verilog code
Simulation waveform

MODULE-5 INTRODUCTION TO VERILOG

Verilog Behavioral description: Structure, Variable Assignment Statement, Sequential Statements, Loop Statements, Verilog Behavioral Description of Multiplexers (2:1, 4:1, 8:1). (Section 3.1 to 3.4 (only Verilog) of Text 3)

Verilog Structural description: Highlights of Structural description, Organization of structural description, Structural description of ripple carry adder. (Section 4.1 to 4.2 of Text 3)

5.1 Behavioral Description highlights

The behavioral description is a powerful tool to describe the systems for which the digital logic structures are not known or hard to generate. Examples of such system are complex arithmetic units, computer control units, and some biological mechanisms.

In dataflow modeling a system was described by using logic equations. In behavioral modeling a system is described by showing how the outputs behave according to the changes in inputs.

In this style of description (modeling), it is not necessary to know the logic diagram of the system, but we must know the behavior of the outputs in response to the changes in the inputs

The primary mechanisms used for behavioral modeling in Verilog are

i) Initial statement: An initial statement executes only once. It begins its execution at the start of the simulation which is at time 0. The syntax for the initial statement is,

```
initial
[timing control]
Procedural_statements
```

*timing_control can be a delay or an event control.
*Where procedural_statement can be one of the following:

1. Continuous_assignment
2. Conditional_statement
3. Case_statement
4. Loop_statement
5. Wait_statement
6. Sequential_block

*execution of an initial statement will result in execution of the procedural_statement once. Here is an example of an initial statement

```
reg yurt;
.....
initial
Yurt=2;
```

In the above example the initial statement contains a procedural assignment with no timing control(delay). The initial statement executes at time 0 and yurt is assigned value 2 at time 0 only.

Consider

```
Reg curt;
……..
Initial
#5 curt=1;
```

Register curt is assigned value 2 at time 5. The initial statement starts execution at time 0 but completes execution at time 5.

ii) always statement: Just like the **initial** statement, an **always** statement also begins execution at time 0. But In contrast to the **initial** statement, an **always** statement executes repeatedly. The syntax of an always statement is:

```
always
  [timing_control]
  Procedural_statement.          //same as explained in initial statement.]
```

Example:

```
always
 clk=~clk;                       //will loop indefinitely.
```

In the above example the **always** statement has one procedural_statement. Since always statement executes repeatedly and there is no timing control the statement clk=~clk will loop indefinitely. Therefore an **always** statement must have a timing control (delay or an event).

Consider

```
always
#5 clk=~clk;
```

This **always** statement upon execution, produces a waveform with a period of 10 time units.

5.2 Structure of Verilog Behavioral Description

Verilog description

```
module halfadd(a, b, sum, carry);
input a, b;
output sum, carry;
reg sum, carry;                  /*since sum and carry are outputs and they are written inside
                                 "always", they should be declared as "reg" or else it will result
                                 in syntax error/*
always @(a, b)
   begin
     #10 sum = a^b;              //statement1-procedural as it is inside always.
     #10 carry = a&b;            //statement2- procedural as it is inside always.
   end
endmodule
```

In the above example

*The name of the module is halfadd. It has two inputs a and b, two outputs sum and carry.

*Any signal declared as output should be declared as a register (reg).Therefore sum and carry are declared as registers.

5.3 Sequential statements

There are several statements associated with the behavioral descriptions which appear inside initial or always in Verilog.

5.3.1 IF statement

"IF" is a sequential statement that appears inside inside always or initial in Verilog. An "IF" statement selects a sequence of statements for execution, depending upon the value of a condition. The condition can be an expression that evaluates to a Boolean value. The general form of an IF statement is:

Verilog if format

```
If (boolen expression)
begin
   statement1;
   statement2; ….
end
else
begin
   statement a;
   statement b; ….
end
```

The execution of "if" is controlled by the Boolean expression. If the expression is true, then statements A are executed. If the expression is false, statements B are executed.

```
if(temp==1)              //Verilog
  temp=s1;
else
temp=s2;
```

Execution of IF as ELSE-IF

```
Verilog

if (Boolean expression1) begin
     statement 1;
statement 2;
  end
  else if (Boolean expression2) begin
```

```
    statement i;
    statement ii;

  end

  else begin
    statement a;
    statement b;
  end
```

Example program:

```
if(signal 1==1)   //Verilog
temp=s1;
else if(signal 2==1)
 temp=s2;
else
temp=s3;
```

Examples1*: Behavioral description of the **2X1 Multiplexers with tristate output**.*
Flowchart & Logic symbol

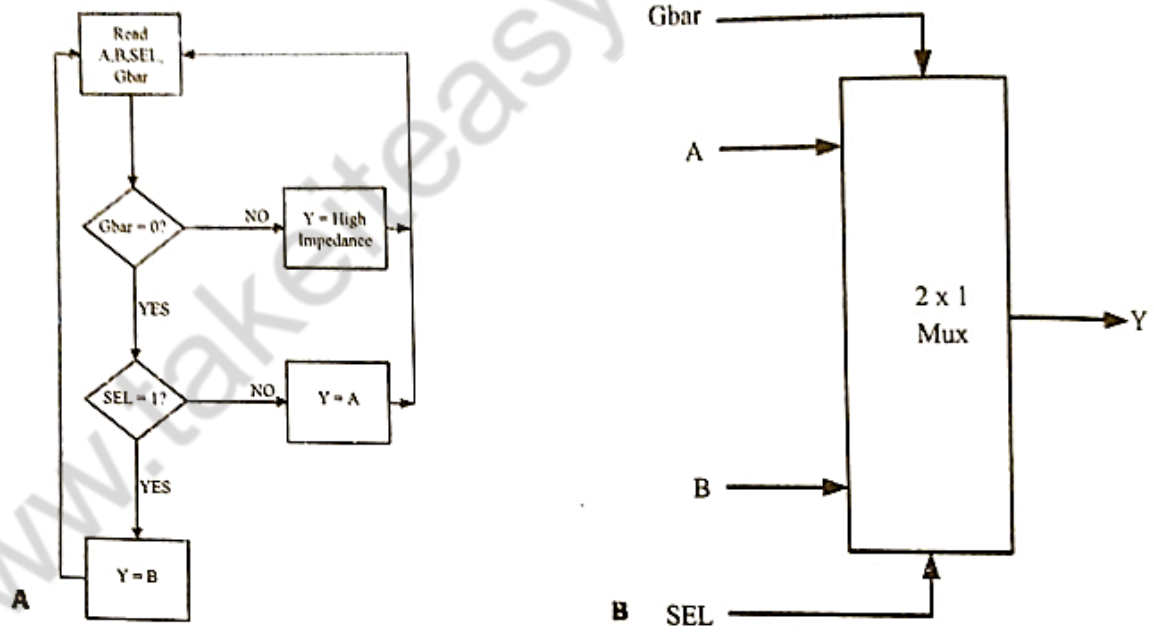


FIGURE 3.1 2x1 Multiplexer. (a) Flow chart. (b) Logic symbol.

Verilog Program using IF-ELSE and ELSE-IF

Verilog 2x1 Multiplexer Using IF-ELSE

```
module mux2x1 (A, B, SEL, Gbar, Y);
input A, B, SEL, Gbar;
output Y;
reg Y;
always @(SEL, A, B, Gbar)
begin
        if (Gbar == 1)
                Y =1'bz;
        else
        begin
                if (SEL)
                        Y = B;
                else
                        Y = A;
        end
end
endmodule
```

Verilog 2x1 Multiplexer Using ELSE-IF

```
module MUXBH (A, B, SEL, Gbar, Y);
input A, B, SEL, Gbar;
output Y;
reg Y;
 always @ (SEL, A, B, Gbar)
begin
                if (Gbar == 0 & SEL == 1)
                begin
                        Y = B;
                end
                else if (Gbar == 0 & SEL == 0))
                Y =A;
                else
                Y = 1'bz;
end
endmodule
```

5.3.2 Signal and Variable assignment

With the help of *Behavioral description* of a D-latch, here we study the difference between the signal- and Variable assignment statements.

A process is written based on signal-assignment statements, and then another program is written based on Variable assignment statements. A comparison of the simulation waveforms highlights the difference between the two methods.

Examples2: *Behavioral description of the **D-Latch**.*

```
module D_latch (d, E, Q, Qb);
input d, E;
output Q, Qb;
reg Q, Qb;
always @ (d, E)
begin
        if (E == 1)
                begin
                        Q =     d;
                        Qb =    ~Q;
                end
end
endmodule
```

Examples3: *Behavioral description of a **Positive edge triggered JK Flip-Flop** using the CASE statements.*

```
module JK_FF (JK, clk, q, qb);
input [1:0] JK;
input clk;
output g, gb;
reg q, qb
always @(posedge clk)
begin
case (JK)
2'd0 :q =q;
2'd1 :q = 0;
2'd2 :q =1;
2'd3: q= ~q;
endcase
```

Examples4: *Behavioral description of a **3-bit Binary counter with Active High Synchronous Clear***

```
module CT_CASE (clk, clr, q);
input clk, clr
output [2:0] q;
reg [2:0] q;
initial
        q= 3'b101;
always @ (posedge clk)
begin
        if (clr == 0)
begin
        case (q)
                3'd0 : q = 3' d1;
                3'd1 : q = 3'd2;
                3'd2 : q = 3'd3;
                3'd3 : q = 3'd4;
                3'd4 : q = 3'd5;
                3'd5 : q = 3'd6;
                3'd6 : q =3'd7;
                3'd7 : q = 3'd0;
                endcase
                end
else
q= 3'b000;
end
endmodule
```

5.3.4 Loop statement

- Loop is used to repeat the execution of the statements inside its body; this repeatation is controlled by the range of an index parameter.
- The loop reduces the size of the code.
- Loop is a sequential statement that has to appear inside process in VHDL or inside always or initial in verilog.
- There are several formats of loop statement namely for,while, verilog repeat and forever.

a).For -loop

The general syntax of for loop in Verilog is as shown below.
The general syntax of for loop in both VHDL and Verilog is as shown below.

Verilog
for(initial_assignment;condition;step_assignment) **begin** Statements……; **End**

For loop execution in Verilog:

Example for Verilog **for** loop

for(i=0;i<=2;i=i+1)
 	
begin
 	
if(temp[i]==1)
 	
begin
 	
result=result+2;
 	
end
 	
end

b).while loop

A while loop is another iterative statement. The general format of while loop is

Verilog
while(condition) begin Statement 1; Statement 2; end

As long as the condition is true, all the statements within the body of the loop are executed. If the condition is false the loop is suspended and the next statement after loop is executed.
Note: 1. In a while loop, we must ensure that statements inside the while loop will cause the loop's condition to evaluate false or else the loop infinite.

"For" loop and "while" loop are common to both VHDL and Verilog except some minor differences in the syntax. But apart from that there are some language specific loop statements.

c). Verilog repeat: The repeat construct executes the set of statements between it's begin and end a fixed number of times. A repeat construct must contain a number which can be a constant, variable or a signal. It cannot contain an expression or condition.
Example:

```
repeat (32)
begin
        #100 i=i+1;
end
```

*In the above example, the statement is executed 32 times. Each time "i" is incremented and assigned to itself after a delay of 100 time units.

d). Verilog forever: The statement "forever" in Verilog repeats the loop endlessly. The loop does not contain any expression and executes until it encounters "$finish" task. A forever can be exited or stopped by using "disable" statement.

- "forever" loop is equivalent to "while" loop with an expression that always evaluates to true.
- One common use of "forever" is to generate clocks in code-oriented test benches.

```
Initial
Begin
        clk=1'b0;
Forever #20 clk=~clk;
End
```

5.4 Highlights of Structural description.

The structural description is a method of defining a system with its basic hardware components. The structural description is best implementation when the hardware components are known. For example consider 2:1 multiplexer, this can be easily implemented if structural descriptions of the basic components are known i.e. 'AND, OR and NOT gates. Structural description can easily describe these components. Structural description is very close to schematic simulation. Here we are going to discuss about gate level and register level descriptions for VERILOG and VHDL.

Facts:

1. Structural description stimulates the system by describing its logical components (AND gate, OR gate and NOT gate).or can be a higher logical level such as Register Transfer Level (RTL) or processor level.

2. The structural description is more suitable rather than behavioral description for the system that required a specific design. Consider for example, a system is performing the operation A+B=C. In behavioral description, we usually write C= A+B and we have no choice in the type of adders used to perform this addition. In structural description, we can specify the type of the adder. For example, look-ahead adders.

3. All statements in structural description are concurrent. At any simulation time, all statements that have an event executed concurrently.

4. The main difference between the VHDL and VERILOG structural description is availability of the components to the user. Verilog recognizes all the primitive gates, such as AND, OR, XOR, NOT, and XNOR gates. Basic VHDL packages do not recognize any gates unless the package is linked to one or more libraries, packages, or modules that have the gate descriptions.

5.5 Organization of the structural description:

The following program (HDL code) describes a system with the help of structural description, Example 4.1. In this program architecture part has parts **declaration and instantiation**.

In the declaration part all the components used in the system description are declared. For example following description declares XOR gate component.

Component xor2

port (i1, i2: in std_logic:o1:

O1: out std_logic:

end component;

The xor2 component has tow inputs “i1” and “i2”, and one output “o1”.

Once the component is used we can use the same component one or more times in the system description. The instantiation part of the code maps the generic input/output to the actual input/output of the system. For example, the statement

X1: xor2 port map (A, B, sum) ;

Maps A to input i1 of xor2, input B to input i2 of xor2., and output sum to output o1 of xor2. This mapping means that the logic relationship between A,B and sum is same as between i1, i2 and o1.

Verilog has a large number of built-in gates; for example the statement

Xor X1(sum, a , b);

Describes a two-input XOR gate, the inputs a and b, and the output is sum. X1 is an optional identifier for the gate: we can also write it as

Xor (sum, a ,b);

Verilog has a complete list of built-in primitive gates. The output of the gate sum has to be listed before the inputs a and b. Figure4.1 shows a list of gates and their code in Verilog.

Program 4.1:HDL Structural Description—Verilog

Verilog Structural Description

```
module system (a, b, sum, cout);
input a, b;
output sum, cout;
xor X1 (sum, a, b);          /* X1 is an optional identifier; it can be omitted.*/
and a1 (cout, a, b);         /* a1 is optional identifier; it can be omitted.*/
endmodule
```

Verilog Half Adder Description

```
module half_add (a, b, S, C);
  input a, b;
  output S, C;
  xor (S, a, b);
  and (C, a, b);
  endmodule
```

Example : Structural description of a 3 bit ripple-carry adder

Refer Class Notes