1. Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.

AIM: To simplify a 4-variable logic expression using K-map and simulate the same using the basic gates.

PRIREQISITES: A Karnaugh map (K-map) is a visual method used to simplify the algebraic expressions in Boolean functions without having to resort to complex theorems or equation manipulations. K-maps are also referred to as 2D truth tables as each K-map is nothing but a different format of representing the values present in a one-dimensional truth table. K-maps basically deals with the technique of inserting the values of the output variable in cells within a rectangle or square grid according to a definite pattern. The number of cells in the K-map is determined by the number of input variables and is mathematically expressed as two raised to the power of the number of input variables, i.e., 2n, where the number of input variables is n.

Thus, to simplify a logical expression with two inputs, we require a K-map with 4 (= 22) cells. A four-input logical expression would lead to a 16 (= 24) celled-K-map, and so on.

K-maps basically deals with the technique of inserting the values of the output variable in cells within a rectangle or square grid according to a definite pattern. The number of cells in the K-map is determined by the number of input variables and is mathematically expressed as two raised to the power of the number of input variables, i.e., 2n, where the number of input variables is n.

Thus, to simplify a logical expression with two inputs, we require a K-map with 4 (= 22) cells. A four-input logical expression would lead to a 16 (= 24) celled-K-map, and so on.

Further, each cell within a K-map has a definite place value obtained using an encoding technique known as **Gray code**. The specialty of this code is the fact that the adjacent code values differ only by a single bit. That is, if the given code-word is 01, then the previous and the next code-words can be 11 or 00, in any order, but cannot be 10 in any case. In K-maps, the rows and the columns of the table use Gray code-labeling which in turn represents the values of the corresponding input variables. This means that each K-map cell can be addressed using a unique Gray Code-Word. Although K-maps can be used for functions with five or six variables, the process is more difficult.

Steps to solve expression using the K-map

1. Select K-map according to the number of variables.
2. Identify minterms or maxterms as given in the problem.
3. For SOP put 1's in blocks of K -map respective to the minterms (0's elsewhere).
4. For POS put 0's in blocks of K -map respective to the maxterms (1's elsewhere).
5. Make rectangular groups containing total terms in power of two like 2, 4, 8... (Except 1) and try to cover as many elements as you can in one group.
6. From the groups made in step 5 find the product terms and sum them up for SOP form.

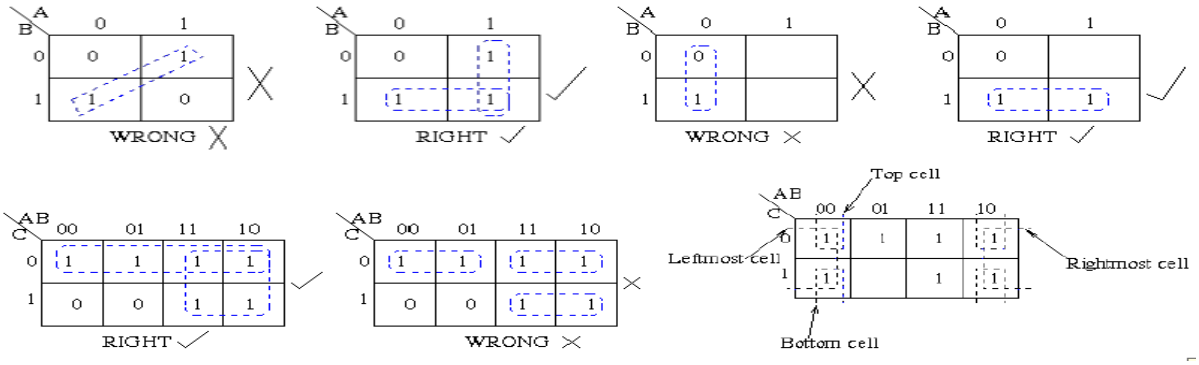
The K-map Fill Order

2 - Variable Map	3- Variable Map	4 - Variable Map
A\B 0 1 0 0 1 1 2 3	A\BC 00 01 11 10 0 0 1 3 2 1 4 5 7 6	AB\CD 00 01 11 10 00 0 1 3 2 01 4 5 7 6 11 12 13 15 14 10 8 9 11 10

Grouping Rules

The Karnaugh map uses the following rules for the simplification of expressions by grouping together adjacent cells containing ones

1. No zeros allowed.
2. No diagonals.
3. Only power of 2 number of cells in each group.
4. Groups should be as large as possible.
5. Everyone must be in at least one group.
6. Overlapping allowed.
7. Wrap around is allowed.
8. Get the fewest number of groups possible.



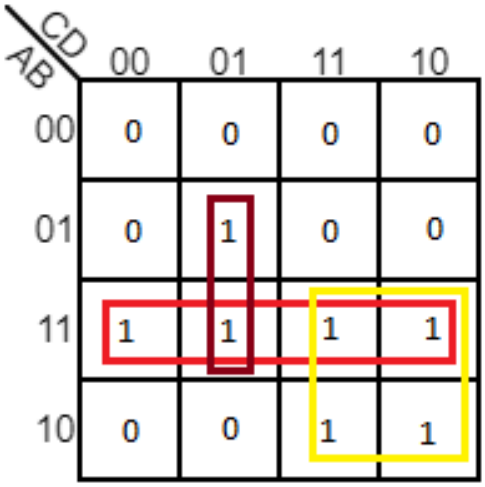
The following function defines a basic Boolean expression that incorporates the function's four variables:

$$F(A, B, C, D) = \bar{A}B\bar{C}D + A\bar{B}C\bar{D} + A\bar{B}CD + AB\bar{C}\bar{D} + AB\bar{C}D + ABC\bar{D} + ABCD$$

$$F(A, B, C, D) = \sum m(5, 10, 11, 12, 13, 14, 15)$$

The one way to help better understand this expression is to break it down into a truth table. Because there are four parameters, the table includes a column for each one. The table also contains a row for each possible combination of parameter values. The last column lists the function's returned value for each combination.

DECIMAL	MINTERMS	MAXTERMS	A	B	C	D	F
0	$\bar{A}\,\bar{B}\,\bar{C}\,\bar{D}$	$(A + B + C + D)$	0	0	0	0	0
1	$\bar{A}\,\bar{B}\,\bar{C}\,D$	$(A + B + C + \bar{D})$	0	0	0	1	0
2	$\bar{A}\,\bar{B}\,C\,\bar{D}$	$(A + B + \bar{C} + D)$	0	0	1	0	0
3	$\bar{A}\,\bar{B}\,C\,D$	$(A + B + \bar{C} + \bar{D})$	0	0	1	1	0
4	$\bar{A}\,B\,\bar{C}\,\bar{D}$	$(A + \bar{B} + C + D)$	0	1	0	0	0
5	$\bar{A}\,B\,\bar{C}\,D$	$(A + \bar{B} + C + \bar{D})$	0	1	0	1	1
6	$\bar{A}\,B\,C\,\bar{D}$	$(A + \bar{B} + \bar{C} + D)$	0	1	1	0	0
7	$\bar{A}\,B\,C\,D$	$(A + \bar{B} + \bar{C} + \bar{D})$	0	1	1	1	0
8	$A\,\bar{B}\,\bar{C}\,\bar{D}$	$(\bar{A} + B + C + D)$	1	0	0	0	0
9	$A\,\bar{B}\,\bar{C}\,D$	$(\bar{A} + B + C + \bar{D})$	1	0	0	1	0
10	$A\,\bar{B}\,C\,\bar{D}$	$(\bar{A} + B + \bar{C} + D)$	1	0	1	0	1
11	$A\,\bar{B}\,C\,D$	$(\bar{A} + B + \bar{C} + \bar{D})$	1	0	1	1	1
12	$A\,B\,\bar{C}\,\bar{D}$	$(\bar{A} + \bar{B} + C + D)$	1	1	0	0	1
13	$A\,B\,\bar{C}\,D$	$(\bar{A} + \bar{B} + C + \bar{D})$	1	1	0	1	1
14	$A\,B\,C\,\bar{D}$	$(\bar{A} + \bar{B} + \bar{C} + D)$	1	1	1	0	1
15	$A\,B\,C\,D$	$(\bar{A} + \bar{B} + \bar{C} + \bar{D})$	1	1	1	1	1

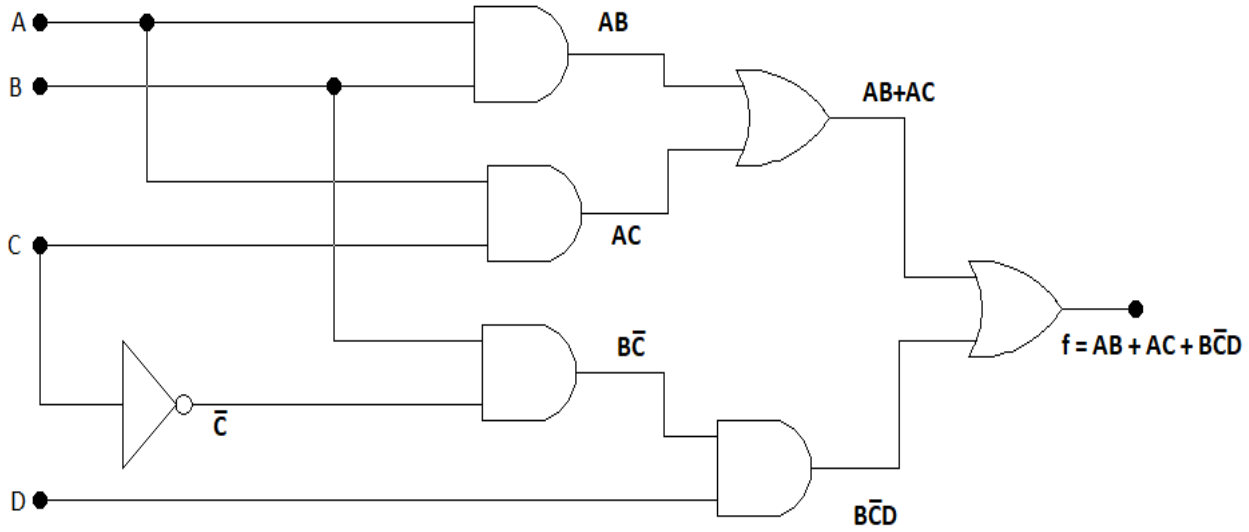


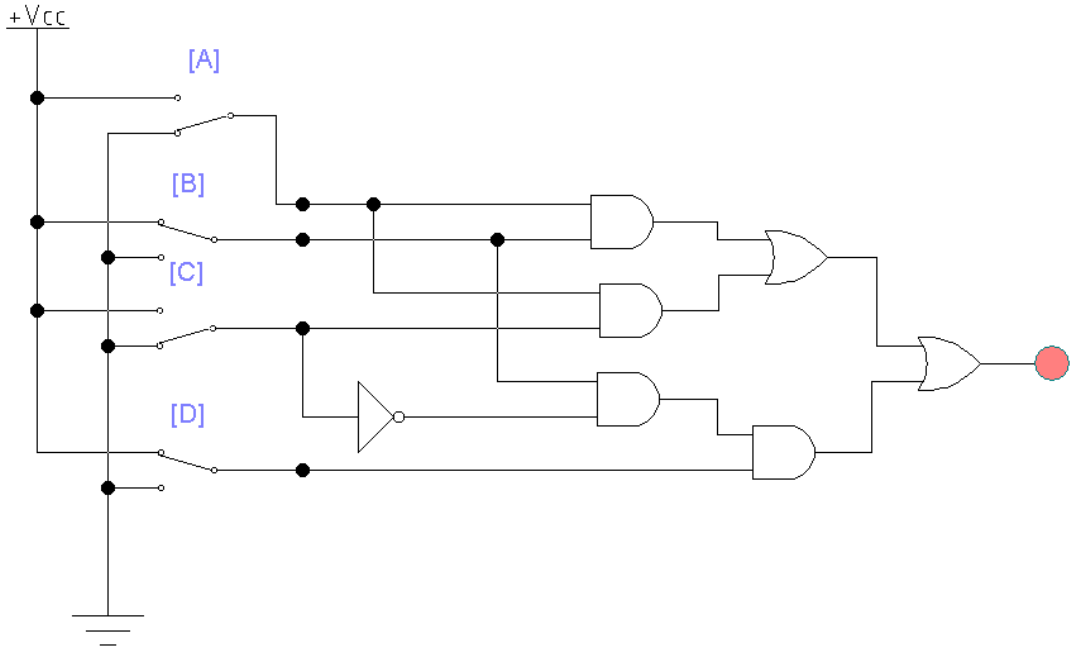
The red group includes all cells in the third row (12, 13, 15 and 14), the yellow group includes four cells from the third and fourth rows (15, 14, 11 and 10) and the brown group includes cells 5 and 13. Each 1 cell needs to be accounted for, even if it means overlapping. For example, the red group overlaps with the yellow and brown groups.

After finding the equation for each group, they can be put all together to create the final simplified expression:

$$f = AB + AC + B\bar{C}D$$

Figure below shows the logic diagram of the above simplified equation implemented with basic gates.





RESULT:

2. Design a 4 bit full adder and subtractor and simulate the same using basic gates.

AIM: To Design a 4 bit full adder and subtractor and simulate the same using basic gates.

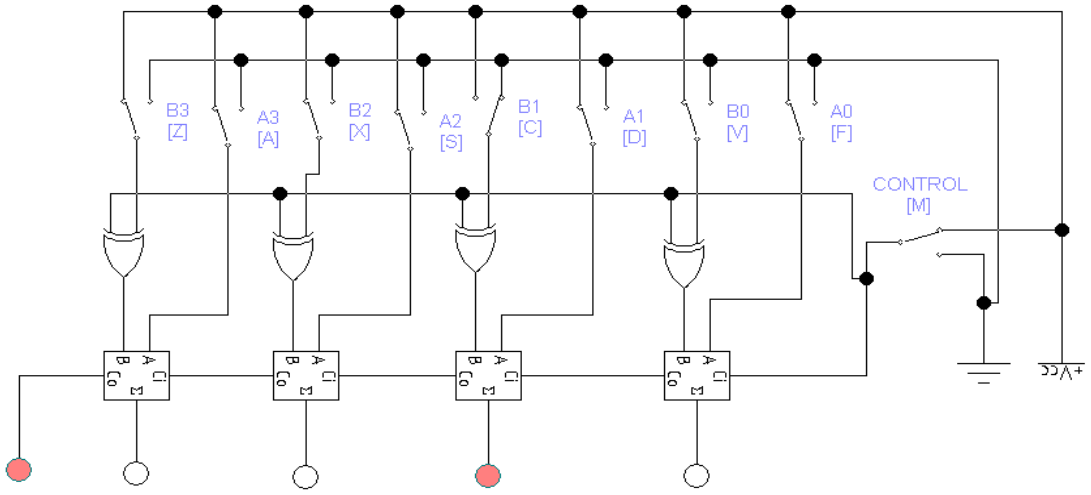
PRIREQISITES:

BINARY ADDER: A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of next full adder in the chain. The augend bits of 'A' and the addend bits of 'B' are designated by subscript numbers from right to left, with subscript '0' denoting the least significant bit. The carries are connected in chain through the full adders. The input carry to the adder is C0 and it ripples through the full adders to the output carry C4. The 'S' outputs generate the required sum bits.

BINARY SUBTRACTOR: The subtraction of unsigned binary numbers can be done most conveniently by means of complements. The subtraction A – B can be done by taking 2's complement of B and adding it to A. The 2's complement can be obtained by taking 1's complement and adding 1 to the least significant pair of bits. The 1's complements can be implemented with inverters, and a 1 can be added to the sum through the input carry. The input carry C0 must be equal to 1 when performing subtraction.

BINARY ADDER/SUBTRACTOR: The addition and subtraction operation can be combined into one circuit with one common binary adder. This is done by including an exclusive – OR gate with each full- adder. The mode input M controls the operation of the circuit. When M=0, the circuit is an adder and when M=1, the circuit becomes a subtractor.

LOGIC DIAGRAM:



TRUTH TABLE:

Input Data A				Input Data B				Addition					Subtraction				
A4	A3	A2	A1	B4	B3	B2	B1	C	S4	S3	S2	S1	B	D4	D3	D2	D1
1	0	0	0	0	0	1	0	0	1	0	1	0	1	0	1	1	0
1	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	1	0	1	0	0	0	0	1	0	1	0	0	1	0	1	0
0	0	0	1	0	1	1	1	0	1	0	0	0	0	1	0	1	0
1	0	1	0	1	0	1	1	1	0	0	1	0	0	1	1	1	1
1	1	1	0	1	1	1	1	1	1	0	1	0	0	1	1	1	1
1	0	1	0	1	1	0	1	1	0	1	1	1	0	1	1	0	1

RESULT:

3. Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural model.

AIM:

To design Verilog HDL to implement simple circuits.

F (A, B, C, D) = ∑m (5, 10, 11, 12, 13, 14, 15)

F = AB + AC + B$\bar{C}$D after K-Map Simplification

//Dataflow Modeling

```
module simple(a,b,c,d, f);
input a,b,c,d;
output f;
assign f=(a & b)|(a & c)|(b & (~c) & d);
endmodule
```

//behavioral Modeling

```
module simple (a,b,c,d, f);
input a,b,c,d;
output reg f; always@(a,b,c,d) begin
if
(({a,b,c,d}==5)||({a,b,c,d}==10)||({a,b,c,d}==11)||({a,b,c,d}==12)||({a,b,c,d}==13)||({a,b,c,d}==
14)||({a,b,c,d}==15))
f=1;
else f=0; end endmodule
```

//structural Modelling

```
module simple(a,b,c,d, f);
input a,b,c,d;
output f;
wire w1,w2,w3,w4; not(w1,c); and(w2,a,b); and(w3,a,c); and(w4,b,w1,d); or(f,w2,w3,w4);
endmodule
```

TESTBENCH

```
module t_ simple_v;
// Inputs reg a;
reg b; reg c; reg d;
// Outputs
wire f;
// Instantiate the Unit Under Test (UUT)
p3behav uut (
.a(a),
.b(b),
.c(c),
.d(d),
.f(f)
);
initial begin
{a,b,c,d}=0;
repeat(15)
#10 {a,b,c,d}={a,b,c,d}+1;
end endmodule
```

//SIMULATION OUTPUT

4. Design Verilog HDL to implement binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.

AIM:

To design Verilog HDL to implement half adder, full adder, half subtractor and full subtractor circuits.

Half Adder:

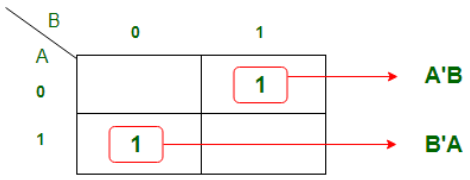
- It is a arithmetic combinational logic circuit designed to perform addition of two single bits.
- It contain two inputs and produces two outputs.
- Inputs are called Augend and Added bits and Outputs are called Sum and Carry.

Let us observe the addition of single bits,

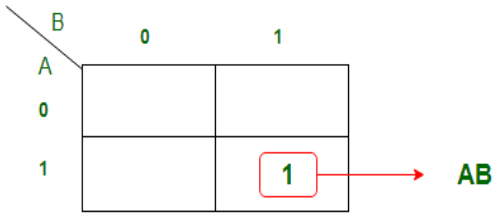
```
0+0=00
0+1=01
1+0=01
1+1=10
```

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

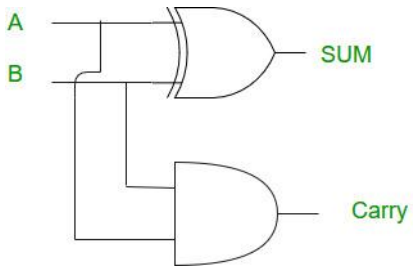
K-map for output variable Sum 'S': **K-map for output variable Carry 'C':**



A'B+B'A=A xor B



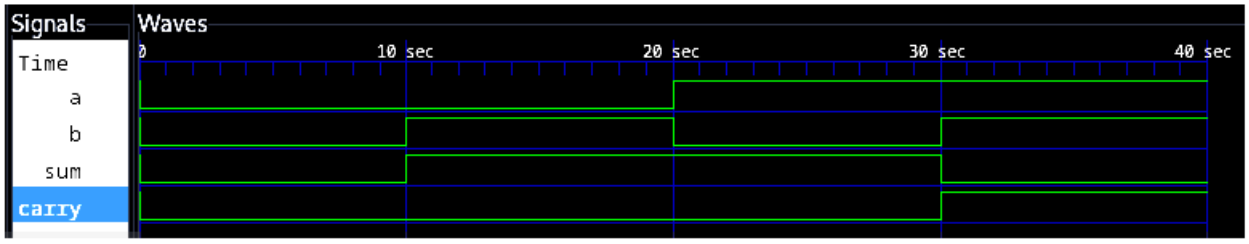
S = A xor B C = AB



Verilog Code Structural Method

```
module half_adder_structural (
  input a,    // Input 'a'
  input b,    // Input 'b'
  output s,   // Output 's' (Sum)
  output c    // Output 'c' (Carry)
);
xor gate_xor (s, a, b);  // XOR gate for sum
 and gate_and (c, a, b);  // AND gate for carry
endmodule

//TESTBENCH

module half_adder_tb;
reg a,b;
```

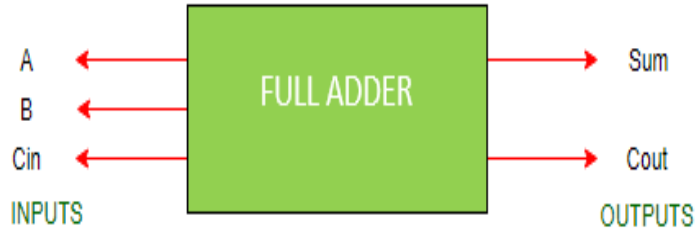
```
wire s,c;

half_adder_s uut(a,b,s,c);

initial begin
{a,b}=0;
Repeat(3)
#10 {a,b}={a,b}+1;

$finish();
end

endmodule
```



2. Full Adder:

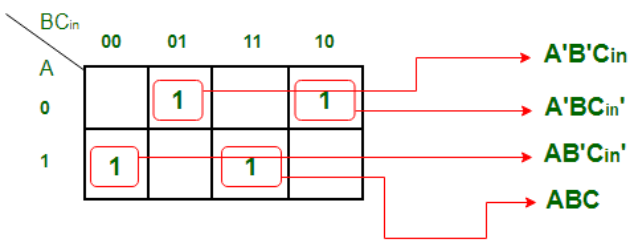
- It is a arithmetic combinational logic circuit that performs addition of three single bits.
- It contains three inputs (A, B, Cin) and produces two outputs (Sum and Cout).

  Where, Cin -> Carry In and Cout -> Carry Out



Inputs			Outputs	
A	B	C_{in}	Sum	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

K-map Simplification for Sum 'S':



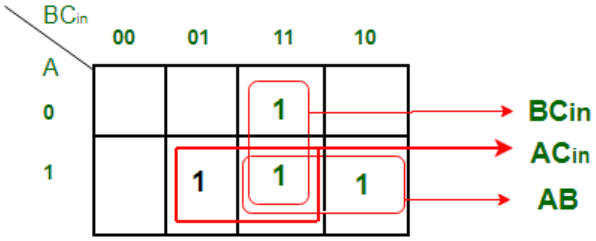
S = A'B'Cin + AB'Cin' + ABC + A'BCin'

S = B'(A'Cin+ACin') + B(AC + A'Cin')

S = B'(A xor Cin) + B (A xor Cin)'

S = A xor B xor Cin

K-map Simplification 'C_{out}':



Cout = BCin + AB + ACin

//full adder using data-flow modeling

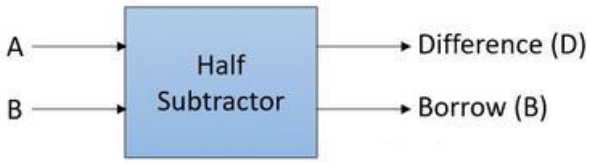
```
module full_adder_d (
   input a,b,cin,
   output sum,carry
);
```

```
assign sum = a ^ b ^ cin;
assign carry = (a & b) | (b & cin)  | (cin & a) ;

endmodule
```

//TESTBENCH

```
module full_adder_tb;
reg a,b,cin;
wire sum,carry;

full_adder_s uut(a,b,cin,sum,carry);
initial begin

{a,b,cin}=0;
Repeat(7)
#10 {a,b,cin}={a,b,cin}+1;

$finish();
End
```

3. HALF SUBTRACTOR



Truth Table

A	B	Difference (D)	Borrow (B)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

D = A ^ B
= A' · B

B

//Half subtractor using data-flow modeling

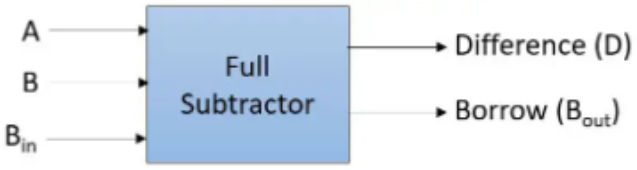
```
module half_subtractor(input a, b, output D, B);
 assign D = a ^ b;
 assign B = ~a & b;
endmodule
```

//TESTBENCH

```
module tb_top;
  reg a, b;
  wire D, B;

  half_subtractor hs(a, b, D, B);

initial begin
{a,b}=0;
```

```
Repeat(3)
#10 {a,b}={a,b}+1;

$finish();
  end
endmodule
```

5. Full Subtractor:



Truth Table

A	B	Bin	Difference (D)	Borrow (Bout)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

D = A ^ B ^ Bin

Bout = A' · B + (A ^ B)' · Bin

//Full subtractor using data-flow modeling

```
module full_subtractor(input a, b, Bin, output D, Bout);
  assign D = a ^ b ^ Bin;
  assign Bout = (~a & b) | (~(a ^ b) & Bin);
endmodule
```

```
module tb_top;
  reg a, b, Bin;
  wire D, Bout;

  full_subtractor fs(a, b, Bin, D, Bout);

initial begin

{a,b,cin}=0;
Repeat(7)
#10 {a,b,cin}={a,b,cin}+1;

$finish();
  end
endmodule
```

5. Design Verilog HDL to implement Decimal adder.

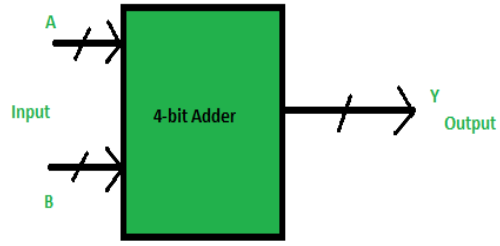
AIM:

To design Verilog HDL to implement decimal adder.

DECIMAL ADDER:

BCD stands for binary coded decimal. It is used to perform the addition of BCD numbers. A BCD digit can have any of ten possible four-bit representations. Suppose, we have two 4-bit numbers A and B. The value of A and B can vary from 0(0000 in binary) to 9(1001 in binary) because we are considering decimal numbers. The output will vary from 0 to 18 if we are not considering the carry from the previous sum. But if we are considering the carry, then the maximum value of output will be 19 (i.e. 9+9+1 = 19). When we are simply adding A and B, then we get the binary sum. Here, to get the output in BCD form, we will use BCD Adder.

Note: If the sum of two numbers is less than or equal to 9, then the value of BCD sum and binary sum will be same otherwise they will differ by 6(0110 in binary). Now, let's move to the table and find out the logic when we are going to add "0110".



Decimal	Binary Sum					BCD Sum				
	C'	S3'	S2'	S1'	S0'	C	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

Example 1:
A = 0101 and B = 1001
sum = 1 0100

Explanation: We are adding A(=5) and B(=9). The value of binary sum will be 1110(=14). But, the BCD sum will be 1 0100, where 1 is 0001 in binary and 4 is 0100 in binary.

//VERILOG CODE FOR BCD ADDER

```
module bcd_adder(a,b, carry_in, sum, carry);
input [3:0] a,b;
input carry_in;
output [3:0] sum;
output carry;
//Internal variables
reg [4:0] sum_temp;
reg [3:0] sum;
reg carry;
//always block for doing the addition
always @(a,b,carry_in)
begin
sum_temp = a+b+carry_in; //add all the inputs
if(sum_temp > 9) begin
sum_temp = sum_temp+6; //add 6, if result is more than 9.
carry = 1; //set the carry output
sum = sum_temp[3:0]; end
else begin
carry = 0;
sum = sum_temp[3:0];
end
end
endmodule
```

//TESTBENCH

```
module tb_bcdadder_v;
// Inputs
reg [3:0] a;
reg [3:0] b;
reg carry_in;
// Outputs
wire [3:0] sum;
wire carry;
// Instantiate the Unit Under Test (UUT)
bcd_adder uut (
.a(a),
.b(b),
.carry_in(carry_in),
.sum(sum),
.carry(carry)
);
initial begin
// Apply Inputs
```

```
a = 0; b = 0; carry_in = 0; #10;
a = 6; b = 9; carry_in = 0; #10;
a = 3; b = 3; carry_in = 1; #10;
a = 4; b = 5; carry_in = 0; #10;
a = 8; b = 2; carry_in = 0; #10;
a = 9; b = 9; carry_in = 1; #10;
end
endmodule
```

SIMULATION WAVE FORMS:

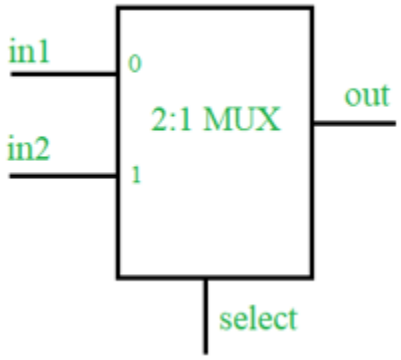
6. Design Verilog HDL to implement different types of multiplexer like 2:1, 4:1 and 8:1.

AIM: To design Verilog HDL to implement different types of multiplexer.

MULTIPLEXER:
A multiplexer is a combinational circuit that has 2^n input lines and a single output line. Simply, the multiplexer is a multi-input and single-output combinational circuit. The binary information is received from the input lines and directed to the output line. On the basis of the values of the selection lines, one of these data inputs will be connected to the output.

2×1 Multiplexer:

In 2×1 multiplexer, there are only two inputs, i.e., I_0 and I_1, 1 selection line, i.e., S0 and single outputs, i.e., Y. On the basis of the combination of inputs which are present at the selection line S0, one of these 2 inputs will be connected to the output. The block diagram and the truth table of the 2×1 multiplexer are given below.



Truth Table:

INPUT- (S)	OUTPUT- (Y)
0	I_0
1	I_1

//VERILOG CODE FOR 2:1 MULTIPLEXER

```
module mux_2_1(sel,i0,i1, y);
input sel,i0,i1;
output y;
assign y = sel ? i1 : i0;
endmodule
```

//TESTBENCH

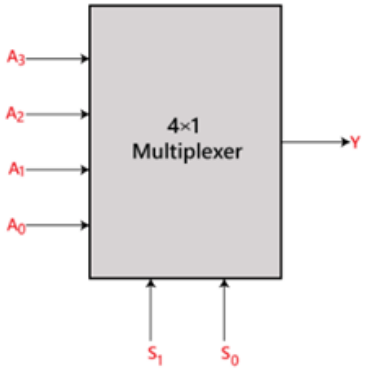
```
module mux_tb_v;
// Inputs
reg sel;
reg i0;
reg i1;
// Outputs
wire y;
// Instantiate the Unit Under Test (UUT)
```

```
mux_2_1 uut (
.sel(sel),
.i0(i0),
.i1(i1),
.y(y)
);
initial begin
i0 = 0; i1 = 1;
sel = 0;
#1;
sel = 1;
end
endmodule
```

SIMULATION WAVEFORM:

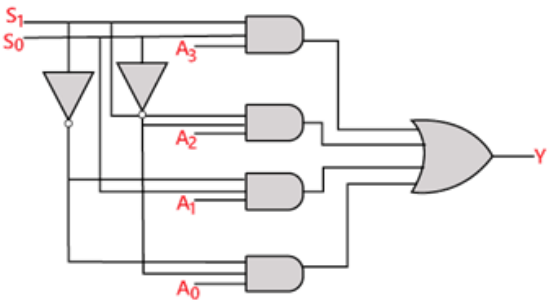
4×1 Multiplexer:

In the 4×1 multiplexer, there is a total of four inputs, i.e., A0, A1, A2, and A3, 2 selection lines, i.e., S0 and S1 and single output, i.e., Y. On the basis of the combination of inputs that are present at the selection lines S0 and S1, one of these 4 inputs are connected to the output. The block diagram and the truth table of the 4×1 multiplexer are given below.



Truth Table:

INPUTS		Output
S_1	S_0	Y
0	0	A_0
0	1	A_1
1	0	A_2
1	1	A_3



//4:1 MULTIPLEXER

```
module mux4x1_gate_level(i,s0,s1, y);
input [3:0] i;
input s0,s1;
output y;
wire n1, n2, n3, n4, n5, n6;
not (n1, s1);
not (n2, s0);
and (n3, i[0], n1, n2);
and (n4, i[1], n1, s0);
and (n5, i[2], s1, n2);
and (n6, i[3], s1, s0);
or (y, n3, n4, n5, n6);
endmodule
```

//TESTBENCH

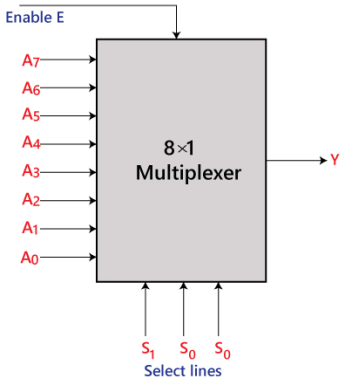
```
module tb_mux4x1_v;
// Inputs
reg [3:0] i;
reg s0;
reg s1;
// Outputs
wire y;
// Instantiate the Unit Under Test (UUT)
mux4x1_gate_level uut (
.i(i),
.s0(s0),
.s1(s1),
```

```
.y(y)
);
initial begin
i=4'b0001;s0 = 0;s1 = 0;
#100; i=4'b0001;s0 = 1;s1 = 0;
#100; i=4'b0100;s0 = 0;s1 = 1;
#100; i=4'b1100;s0 = 1;s1 = 1;
#100;
end
endmodule
```

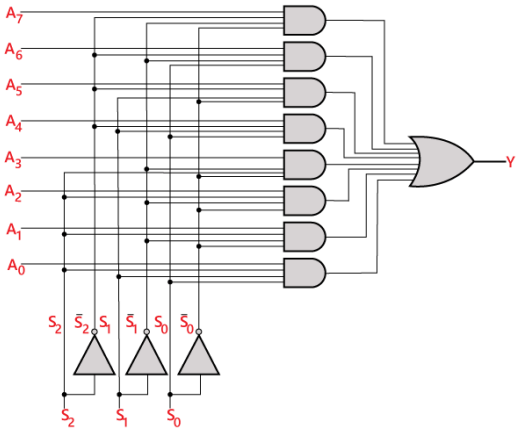
SIMULATION WAVEFORM

8 to 1 Multiplexer

In the 8 to 1 multiplexer, there are total eight inputs, i.e., A0, A1, A2, A3, A4, A5, A6, and A7, 3 selection lines, i.e., S0, S1and S2 and single output, i.e., Y. On the basis of the combination of inputs that are present at the selection lines S0, S1, and S2, one of these 8 inputs are connected to the output. The block diagram and the truth table of the 8×1 multiplexer are given below.



INPUTS			Output
S_2	S_1	S_0	Y
0	0	0	A_0
0	0	1	A_1
0	1	0	A_2
0	1	1	A_3
1	0	0	A_4
1	0	1	A_5
1	1	0	A_6
1	1	1	A_7



//8:1 MULTIPLEXER

```
module mux8x1(i, s, y);
input [7:0] i;
input [2:0] s;
output reg y;
always@(s)
begin
case(s)
3'b000:y=i[0];
3'b001:y=i[1];
3'b010:y=i[2];
3'b011:y=i[3];
3'b100:y=i[4];
3'b101:y=i[5];
3'b110:y=i[6];
3'b111:y=i[7];
endcase
end
endmodule
```

//TESTBENCH

```
module tb_mux8x1_v;
// Inputs
reg [7:0] i;
reg [2:0] s;
// Outputs
wire y;
// Instantiate the Unit Under Test (UUT)
mux8x1 uut (
.i(i),
```

```
.s(s),
.y(y)
);
initial begin
// Initialize Inputs
i=8'b00000001;s=0;
#10; i=8'b00000001;s=1;
#10; i=8'b00000001;s=2;
#10; i=8'b00000101;s=3;
#10; i=8'b00000001;s=4;
#10; i=8'b00010001;s=5;
#10; i=8'b01000001;s=6;
#10; i=8'b10000001;s=7;
#10;
end
endmodule
```

SIMULATION WAVEFORM

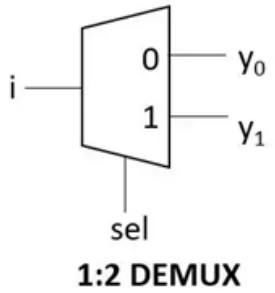
7. Design Verilog HDL to implement different types of Demultiplexer.

AIM: To design Verilog HDL to implement different types of demultiplexer.

DEMULTIPLEXER:

De-Multiplexer is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of 2n outputs. The input will be connected to one of these outputs based on the values of selection lines.
Since there are 'n' selection lines, there will be 2n possible combinations of zeros and ones. So, each combination can select only one output. De-Multiplexer is also called as De-Mux.

1:2 demultiplexer



1:2 DEMUX

Truth Table

sel	y_0	y_1
0	i	0
1	0	i

// VERILOG CODE FOR 1:2 DEMUX

```
module demux1x2(sel,i, y0,y1);
input sel,i;
output y0,y1;
assign {y0,y1} = sel?{1'b0,i}: {i,1'b0};
endmodule
```

// TESTBENCH

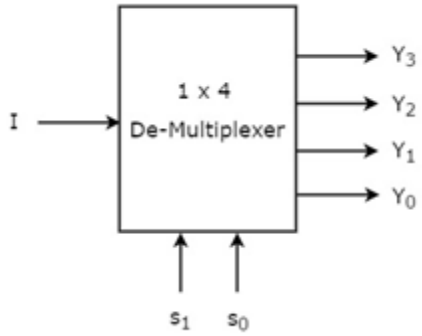
```
module t_demux_v;
// Inputs
reg sel;
reg i;
// Outputs
wire y0;
wire y1;
// Instantiate the Unit Under Test (UUT)
demux1x2 uut (
.sel(sel),
.i(i),
.y0(y0),
.y1(y1)
);
```

```
initial begin
sel=0; i=0; #10;
sel=0; i=1; #10;
sel=1; i=0; #10;
sel=1; i=1; #10;
end
endmodule
```

SIMULATION WAVEFORMS:

1x4 De-Multiplexer

1x4 De-Multiplexer has one input I, two selection lines, s1 & s0 and four outputs Y3, Y2, Y1 &Y0. The block diagram of 1x4 De-Multiplexer is shown in the following figure.



Truth Table

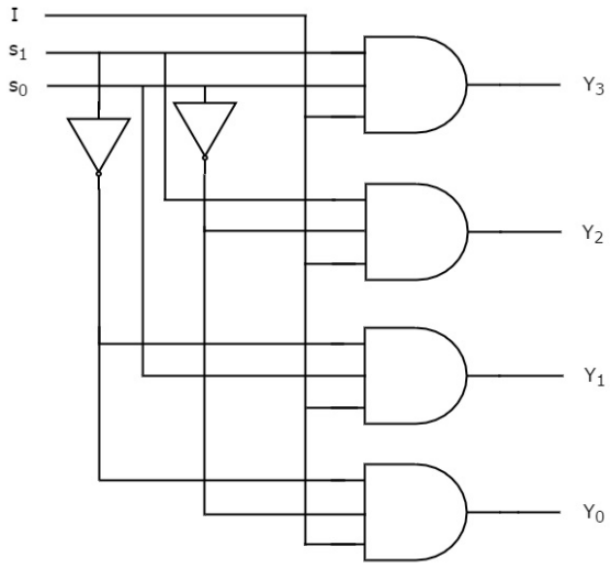
Selection Inputs		Outputs			
S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

$$Y_3 = s_1 s_0 I$$

$$Y_2 = s_1 s_0' I$$

$$Y_1 = s_1' s_0 I$$

$$Y_0 = s_1' s_0' I$$



//1:4 Demux Verilog Code

```
module demux_1_4(
input [1:0] sel,
input i,
output reg y0,y1,y2,y3);
always @(sel,i) begin
case(sel)
2'h0: {y0,y1,y2,y3} = {i,3'b0};
2'h1: {y0,y1,y2,y3} = {1'b0,i,2'b0};
2'h2: {y0,y1,y2,y3} = {2'b0,i,1'b0};
2'h3: {y0,y1,y2,y3} = {3'b0,i};
//default: $display("Invalid sel input");
endcase
end
endmodule
```

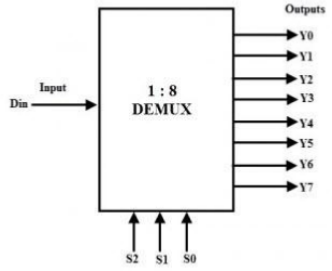
//Testbench Code

```
module t_demux_1_4_v;
// Inputs
reg [1:0] sel;
reg i;
// Outputs
wire y0;
wire y1;
wire y2;
wire y3;
// Instantiate the Unit Under Test (UUT)
demux_1_4 uut (
.sel(sel),
.i(i),
.y0(y0),
.y1(y1),
.y2(y2),
.y3(y3)
);
initial begin
sel=2'b00; i=0; #10;
sel=2'b00; i=1; #10;
sel=2'b01; i=0; #10;
sel=2'b01; i=1; #10;
sel=2'b10; i=0; #10;
sel=2'b10; i=1; #10;
sel=2'b11; i=0; #10;
sel=2'b11; i=1; #10;
end
endmodule
```

SIMULATION WAVEFORM

1x8 De-Multiplexer

Let the 1x8 De-Multiplexer has one input I, three selection lines s2, s1 & s0 and outputs Y7 to Y0. The Truth table of 1x8 De-Multiplexer is shown below.



S2	S1	S0	O0	O1	O2	O3	O4	O5	O6	O7
0	0	0	I	0	0	0	0	0	0	0
0	0	1	0	I	0	0	0	0	0	0
0	1	0	0	0	I	0	0	0	0	0
0	1	1	0	0	0	I	0	0	0	0
1	0	0	0	0	0	0	I	0	0	0
1	0	1	0	0	0	0	0	I	0	0
1	1	0	0	0	0	0	0	0	I	0
1	1	1	0	0	0	0	0	0	0	I

//VERILOG CODE FOR 1X8 DE-MULTIPLEXER

```
module demux_8_1(i,s0,s1,s2, y);
input i,s0,s1,s2;
output [7:0] y;
reg [7:0]y=0;
always @ (i or s0 or s1 or s2)
case ({s2,s1,s0})
0: y[0] = i;
1: y[1] = i;
2: y[2] = i;
3: y[3] = i;
4: y[4] = i;
5: y[5] = i;
6: y[6] = i;
7: y[7] = i;
endcase
endmodule
```

//TESTBENCH

```
module t_demux_8_1_v;
// Inputs
reg i;
reg s0;
reg s1;
reg s2;
// Outputs
wire [7:0] y;
// Instantiate the Unit Under Test (UUT)
demux_8_1 uut (
.i(i),
.s0(s0),
.s1(s1),
```

```
.s2(s2),
.y(y)
);
initial begin
// Initialize Inputs
i = 1;s0 = 0;s1 = 0;s2 = 0;
#10 s2=0; s1=0; s0=1;
#10 s2=0; s1=1; s0=0;
#10 s2=0; s1=1; s0=1;
#10 s2=1; s1=0; s0=0;
#10 s2=1; s1=0; s0=1;
#10 s2=1; s1=1; s0=0;
#10 s2=1; s1=1; s0=1;
end
endmodule
```

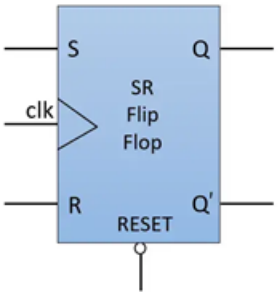
SIMULATION WAVEFORM:

8. Design Verilog HDL for implementing various types of Flip-Flops such as SR, JK and D.

AIM: To design Verilog HDL for implementing various types of Flip-Flops such as SR, JK and D.

1. SR FLIP-FLOP

The SR flip flop has two inputs SET 'S' and RESET 'R'. As the name suggests, when S = 1, output Q becomes 1, and when R = 1, output Q becomes 0. The output Q' is the complement of Q.



SR Flip Flop

Truth Table

S	R	Q_{n+1}
0	0	Q_n(No Change)
0	1	0
1	0	1
1	1	x

//VERILOG CODE FOR SR FLIP-FLOP

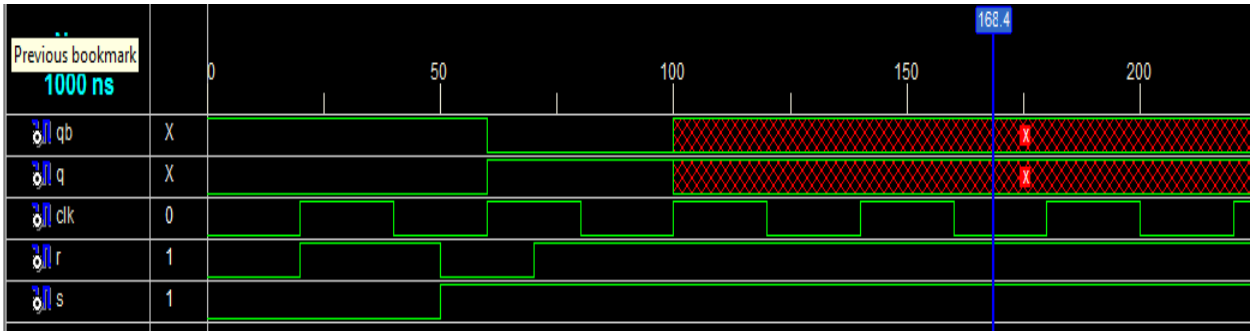
```
module sr(s,r,clk, q,qb);
   input s,r,clk;
   output reg q=0;
           output qb;
 always@(posedge clk) begin
       case({s,r})
       2'b00: q <= q;    // No change
       2'b01: q <= 1'b0; // reset
       2'b10: q <= 1'b1; // set
       2'b11: q <= 1'bx; // Invalid inputs
     endcase
  end
  assign qb = ~q;
endmodule
```

// TESTBENCH CODE

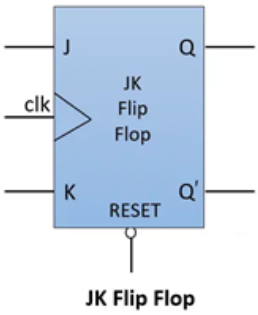
```
module t_sr_v;

          // Inputs
          reg s;
          reg r;
          reg clk;

          // Outputs
```

```
        wire q;
        wire qb;

        // Instantiate the Unit Under Test (UUT)
        sr uut (
                .s(s),
                .r(r),
                .clk(clk),
                .q(q),
                .qb(qb)
        );

        initial begin
   s <= 0;r <= 0;
         clk = 1'b0;
  forever #20 clk = ~clk ;
 end
initial begin
    s <= 0;r <= 0;
    #20 s <= 0;r <= 1;
    #30 s <= 1;r <= 0;
    #20 s <= 1;r <= 1;

  end
endmodule
```

SIMULATION WAVEFORM



2. JK FLIP-FLOP



JK Flip Flop

Truth Table

J	K	Q_{n+1}
0	0	Q_n(No Change)
0	1	0
1	0	1
1	1	$\overline{Q_n}$(Toggles)

//VERILOG CODE FOR JK FLIP-FLOP

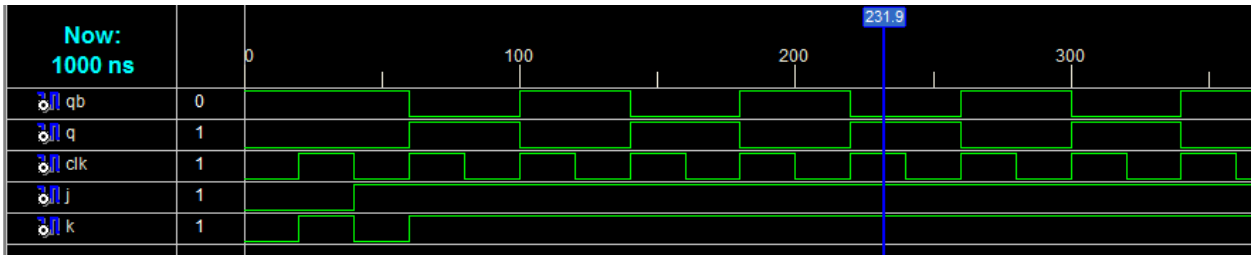
```
module jk(j,k,clk,q,qb);
   input j,k,clk;
   output reg q=0;
          output qb;

always @ (posedge clk)
     case ({j,k})
       2'b00 :  q <= q;
       2'b01 :  q <= 0;
       2'b10 :  q <= 1;
       2'b11 :  q <= ~q;
     endcase
                    assign qb=~q;
endmodule
```

// TESTBENCH CODE

```
module t_jk_v;

          // Inputs
          reg j;
          reg k;
          reg clk;

          // Outputs
          wire q;
          wire qb;

          // Instantiate the Unit Under Test (UUT)
          jk uut (
                    .j(j),
```

```
                .k(k),
                .clk(clk),
                .q(q),
                .qb(qb)
        );
initial begin
    j <= 0;k <= 0;
          clk = 1'b0;
   forever #20 clk = ~clk ;
 end
 initial begin
     j <= 0;k <= 0;

     #20 j <= 0;k <= 1;
     #20 j <= 1;k <= 0;
     #20 j <= 1;k <= 1;

  end

endmodule
```

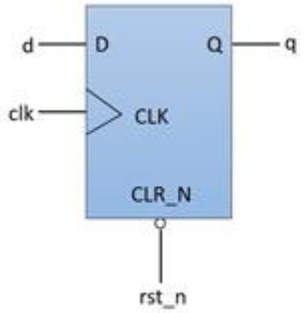
SIMULATION WAVEFORM



3. D FLIP-FLOP

The D flip flop is a basic sequential element that has data input ‘d’ being driven to output ‘q’ as per clock edge. Also, the D flip-flop held the output value till the next clock cycle. Hence, it is called an edge-triggered memory element that stores a single bit.

The below D flip flop is positive edge-triggered and has asynchronous active low reset. As soon as reset is triggered, the output gets reset



Clock Edge	D	Q	Description
↓	X	Q	No Change (Store previous input)
↑	1	1	Set Q to 1
↑	0	0	Reset Q to 0

Asynchronous active low reset D flip flop

//VERILOG CODE FOR D FLIP-FLOP

```
module d_flip_flop(q,qb,d,clk,reset);
   input d,clk,reset;
          output reg q;
          output qb;
always @(posedge clk or posedge reset)
  begin

   if (reset == 1'b1 )
      q <= 1'b0;
   else
      q <= d;
  end
  assign qb=~q;
endmodule
```

// TESTBENCH CODE

```
module t_d_flip_flop_v;

          // Inputs
          reg d;
          reg clk;
          reg reset;

          // Outputs
          wire q;
          wire qb;
```

```
        // Instantiate the Unit Under Test (UUT)
        d_flip_flop uut (
                .q(q),
                .qb(qb),
                .d(d),
                .clk(clk),
                .reset(reset)
        );

        initial
 begin
  d=0;clk = 1'b0;
  forever #20 clk = ~clk ;
 end

initial
 begin
         reset = 1'b1;#20;
        reset = 1'b0;#20;
         d = 1'b0;   #20;
        d = 1'b1;   #20;

 end
endmodule
```

SIMULATION WAVEFORM