INDEX

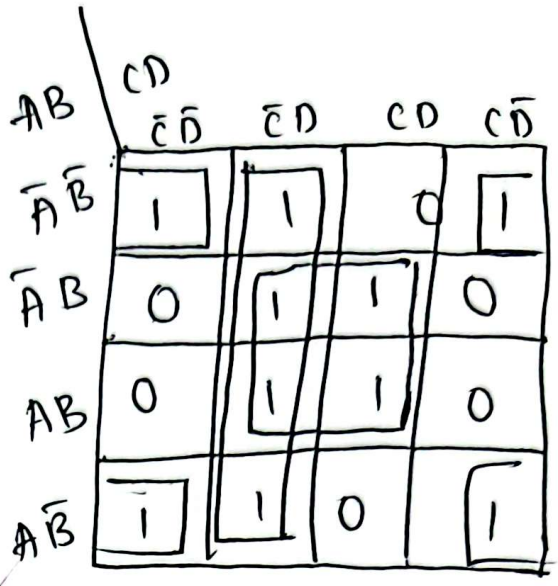
Sl.No.	Date	DESCRIPTION	Page No.	Marks	Signature
1	20/11/23	Program on Structural	1-2	8	
2	4/12/23	Program on half addu and full addus and Subtrac tor	3-6	10	
3	18/12/23	Program on multiplex	7-9	10	
4	18/1/24	Program On de-muttiplex	10-12	10	
5	21/1/24	Program on flip-flop	13-15	10	
6	22/2/24	program on 4 bit full addus	16-17	10	
7	12/2/24	Program on BCD addu	17-18	16	
8	12/2/24	Program on behavioural model	19-20	10	

EXPERIMENT No. : 01

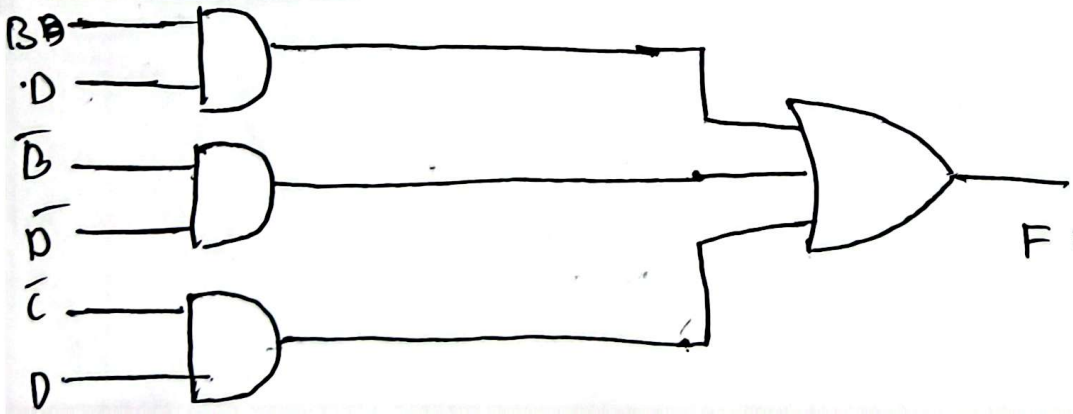
Truth table

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

K-map



Logic gate



$F = BD + \bar{B}\bar{D} + \bar{C}D$

TITLE

Given a 4 variable logic expression, Simplify it using appropriate tecnique & stimulate the same using basic gate $F(A,B,C,D) = \Sigma m(0,1,2,5,7,8,9,10,13,15)$

Aim:- To Simply the given 4-variable logic expression using K-map & Stimulate using basic gate

Theory:- A K-map is a graphical representation of truthtable which is a Systematic way to Simplify & optimize boolean algebra Expression used in digital Circuit design K-maps are helpful for minimizing the no of logic gates in a given logic func

Basic Concepts:-

Truth table:- A truthtable is a tabular representation of all posible input combination & their Corresponding output values for a logic func It helps define the behaviour of the func

Boolean:- A mathamatical func System used to manipulate binary variable & perform logical operation Such as AND, OR & NOT. It is essential for Simplify & optimizing logic func

TITLE

Program :

```
Module. AOI_Structural (A,B,C,D,F);
        input A,B,C,D;
        output F;
        assign F = ~((~A&B) | (C&D));
        endmodule
```

Program

```
module. AOI-Structural (F,A,B,C,D);
input A,B,C,D;
output F;
assign F = (B&D) | (~C&D);
end module;
```


Msgs
/AOI_structural/F
/AOI_structural/A
/AOI_structural/B
/AOI_structural/C
/AOI_structural/D
St1
St0
St0
St1
St0

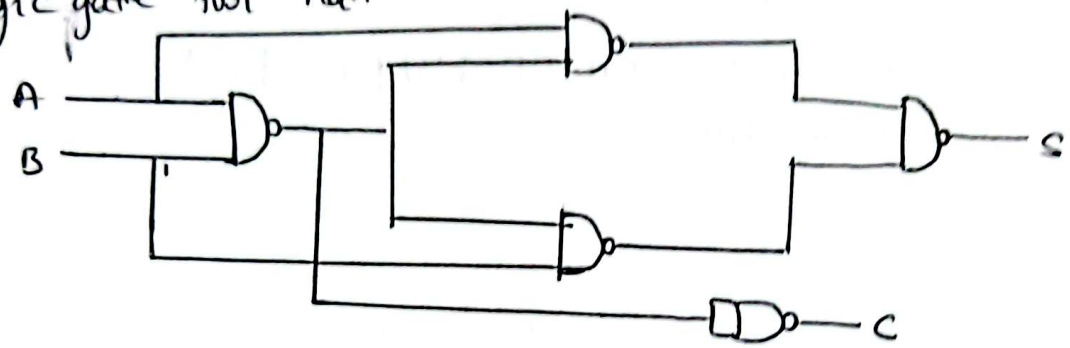
Truth table for half adder

n	A	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S = A \oplus B$

$C = AB$

logic gate for half



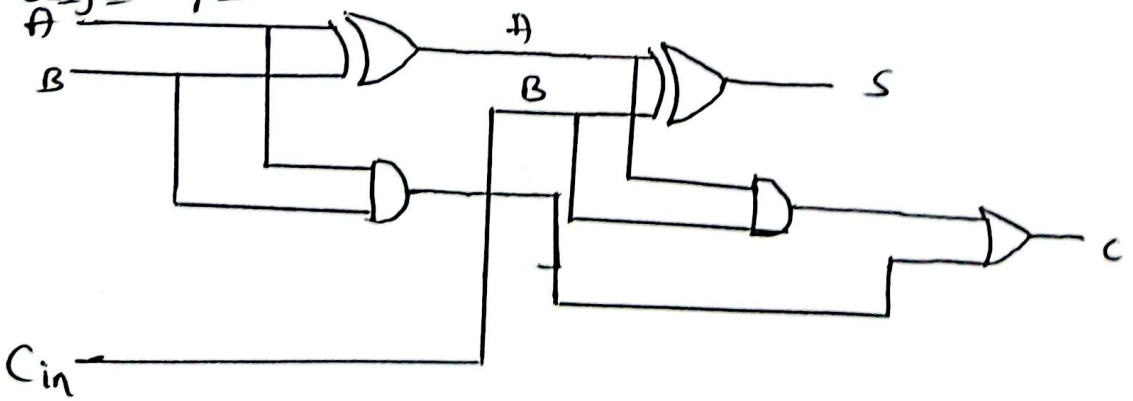
Truth table for full adder

A	B	C_{in}	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S = A \oplus B \oplus C$

$C_{in} = AB + BC_{in} + AC_{in}$

logic gate for full adder



TITLE

Design Verilog HDL to find implement Binary Adders-Subtractors Half & full adders, Half & full-Subtractor

Aim: To design & verify
i) Half adders & full adders
ii) Half subtractors & full Subtractor

Theory:

Half adders: A Combinational logical circuit that perform the addition of 2 bits of data. A & B is called one of which is Sum bit S and the other is the Carry bit C. The boolean func are

$S = A \oplus B$ $\quad C = AB$

Full Adders: A Combinational logical circuit that perform the half adders which doesn't take carry bit from its privious stage into the Carry bit in privious stage called Carry-in-bit that adders 2-bits, A & B & Carry a in Cin, Cin is called a full Adders. The boolean func are.

$S = (x \oplus y) \oplus Cin$ $\quad C = xy + Cin(x \oplus y)$

TITLE

Half Subtractor :- Subtracting a single bit binary value B another A produces a difference bit D & a below outbf bit B-out. The operation is called half subtractor & the Circuit to realize it is called half-Subtractor. The boolean func are

$$S = A \oplus BC = A'B$$

Full Subtractor :- Subtracting 2-Single bit binary value B, Cin func a Single bit value. A produces a difference bit D & a boolean out B bit this is called full Subtractor. The boolean func are

$$D = (x \oplus y) \oplus Cin \quad . \quad B = A'B + A'(Cin) + B(cin)$$

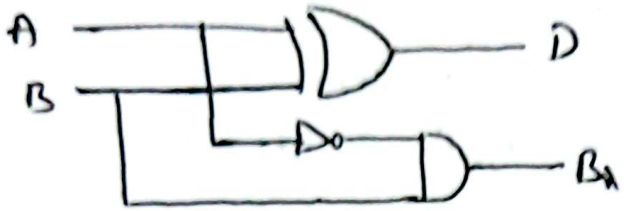
Half Subtractor Truth table

A	B	D	B_h
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

$D = A \oplus B$

$B_h = \bar{A}B$

Logic gate Half Subtractor



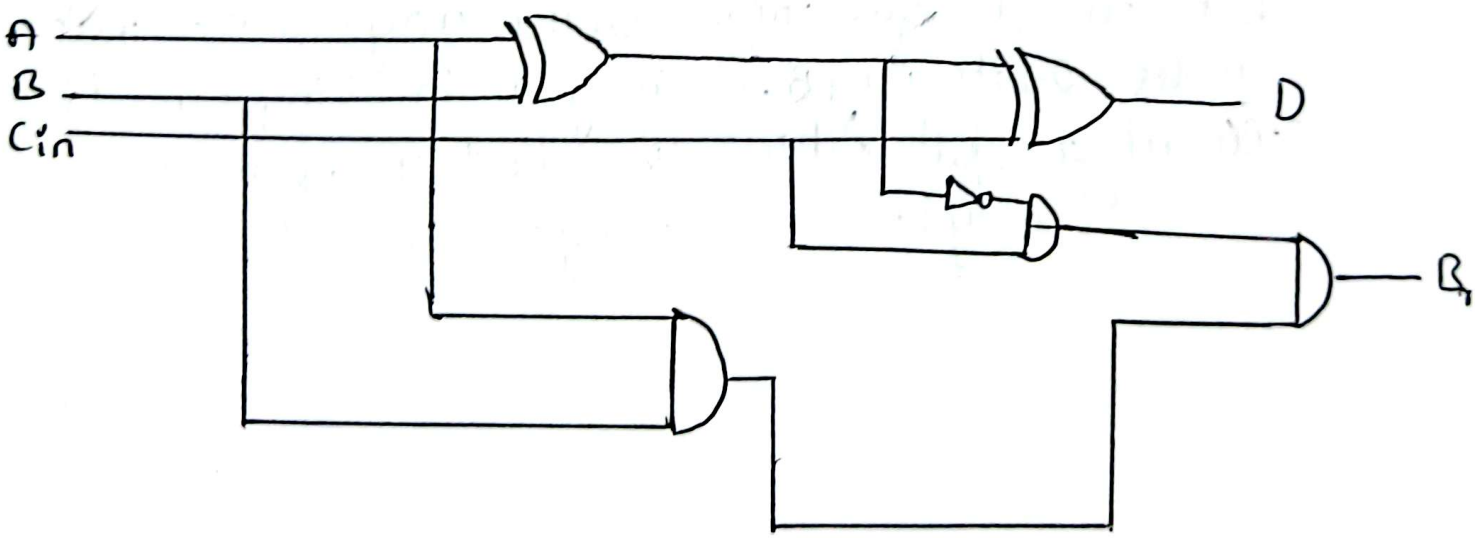
Full Subtractor - Truth table

A	B	C_{in}	D	B_h
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

$D = A \oplus B \oplus C$

$B_h = \bar{A}B + BC_{in} + \bar{A}C_{in}$

Logic gate - full substractor



TITLE

Program on Half Adder

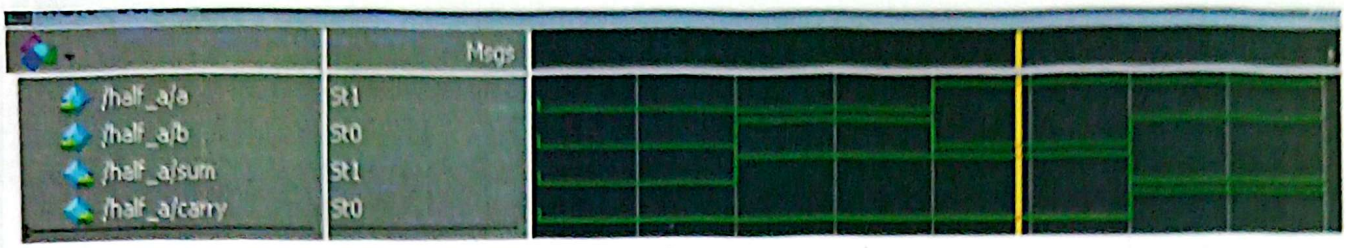
```
module half_adder (a, b, Sum, carry);
    input a, b;
    output Sum, carry;
    assign Sum = a^b;
    assign carry = (a & b);
    endmodule
```

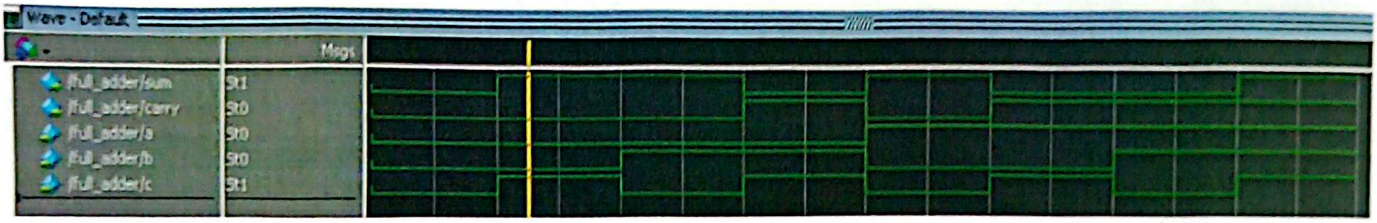
Program on Full adder

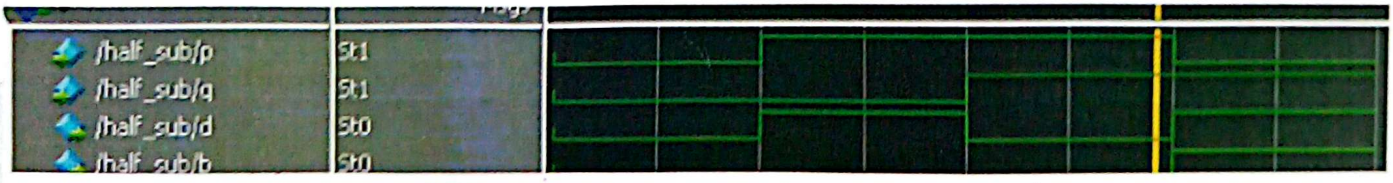
```
module full_adder (a, b, c, Sum, carry);
    input a, b, c;
    output Sum = a ^ b ^ c;
    assign Sum = (a & b) | (a&c) | (c&b);
    endmodule
```

Program on half Subtractor.

```
module half Subtractor (A, B, D, B);
    input A, B;
    Output D, B;
    assign D = A ^ B;
    assign B = (~A) & B;
    endmodule
```


Msgs
/half_a/a
/half_a/b
/half_a/sum
/half_a/carry
St1
St0
St1
St0


Wave - Default
Msgs
/full_adder/sum
/full_adder/carry
/full_adder/a
/full_adder/b
/full_adder/c
St1
St0
St0
St0
St1

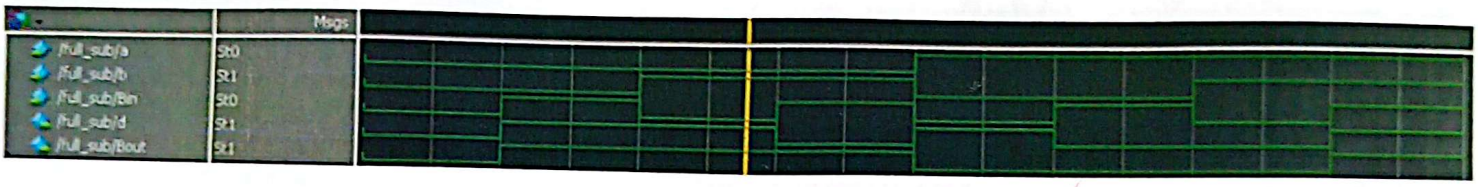

/half_sub/p
/half_sub/q
/half_sub/d
/half_sub/b
St1
St1
St0
St0

TITLE

Program on full Subtractor

```
module full_ Subtractor (A, B, Cin, D, Bi);
input A, B, Cin;
output B, Bi;
assign D = A ^ B ^ Cin;
assign B = (~A & B) | (~(A ^ B) & Cin);
end module
```

EXPERIMENT No. :.....



TITLE

Design Verilog program to implement different types of multiplex 2:1, 4:1, 8:1

Aim :- Implement different types of multiplex 2:1, 4:1 & 8:1

Theory :-

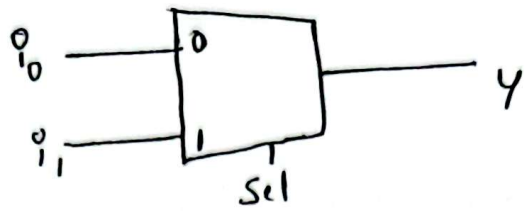
2:1 multiplex : It has two input & one select line
4:1 multiplexer : It has 4 input & 2 select line
8:1 multiplexer : It has 8 input & 3 select lines

A multiplex is a combination circuit that selects binary information from many input lines & direct it to a single output line. The selection of particular input lines is controlled by a set of selection lines.

EXPERIMENT NO. .....

Truth table of 2:1 multiple

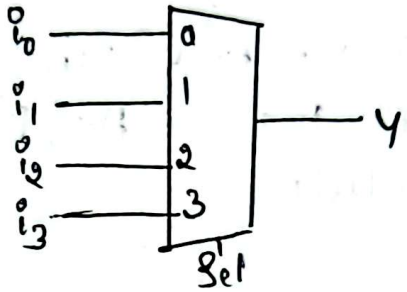
Sel	Y
0	i_0
1	i_1



block diagram

Truth table 4:1 multiplexer

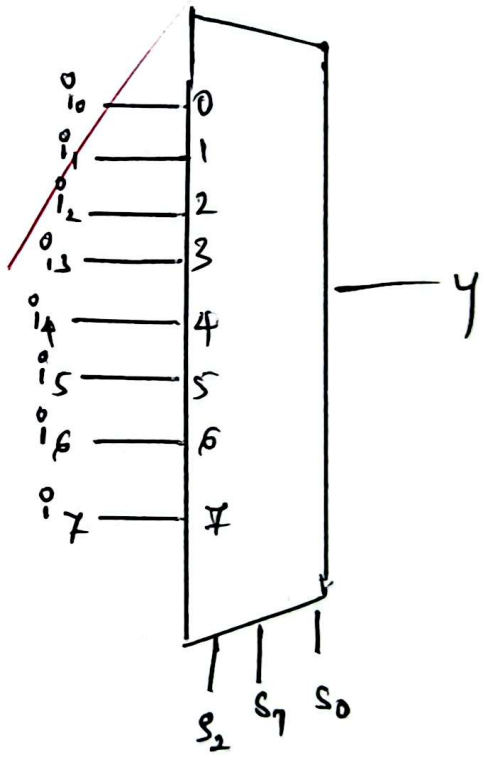
Sel[0]	Sel[1]	Y
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_3



black diagram

Truth table 8:1 Multiplexer

S_1	S_2	S_3	Y
0	0	0	i_0
0	0	1	i_1
0	1	0	i_2
0	1	1	i_3
1	0	0	i_4
1	0	1	i_5
1	1	0	i_6
1	1	1	i_7



TITLE

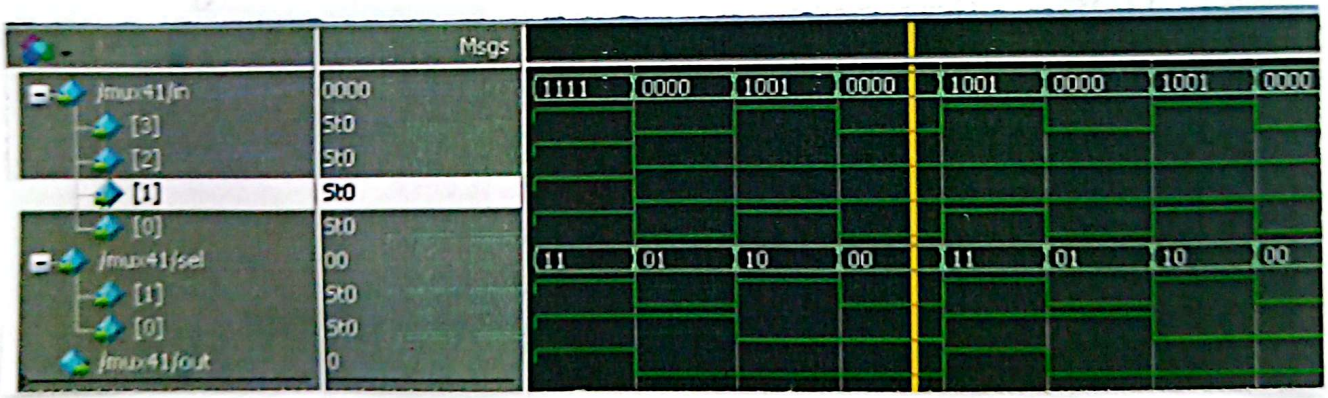
Program on 2:1 multiplex

```
module m21(D0, D1, S, Y);
    input wire D0, D1, S;
    output reg Y;
    always @(D0 or D1 or S)
    begin
    if(S)
     Y = D1;
    else
       Y=D0;
    end
    endmodule
```

Program on 4:1 Multiplex

```
module mux4(in, sel, out);
    input [3:0] in;
     input [1:0] sel;
     output reg out;
     always@(c
      begin
      case(sel)
      2'b00: out = in[0];
      2'b01: out = in[1];
      2'b10: out = in[2];
      2'b11 : out = in[3]
      end case
      end
      end module
```


Msgs
St0
St1
St0
0


Msgs
/mux41/in
0000
[3]
St0
[2]
St0
[1]
St0
[0]
St0
/mux41/sel
00
[1]
St0
[0]
St0
/mux41/out
0
1111
0000
1001
0000
1001
0000
1001
0000
11
01
10
00
11
01
10
00

TITLE

Program on 8:1 multiplexer

```
module mux81(y, a, s)
    input [7:0]a;
    input [2:0] s;
    output reg y;
    always @(a,s)
     begin
     case(s)
     3'b000: y = a[0];
     3'b001: y = a[1];
     3'b010: y = a[2];
     3'b011: y = a[3];
     3'b100: y = a[4];
     3'b101: y = a[5];
     3'b110: y = a[6];
     3'b111: y = a[7];
     end case
     end
     endmodule
```

TITLE

Design verilog program to implement types of De-multiplex

Aim: Program on De-multiplex

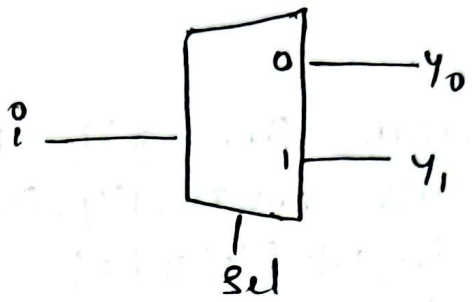
Theory:-
A demultiplex is a Combinational Circuit that work-s exactly opposite to a multiplex. A Demux has a Single input line that Connects to any one of the output lines based on its control input Signal ou selection lines.

There are 2 types of Demultiplex
1:2 Demultiplex: It has one select line and 2 output lines
1:4 Demultiplex: It has one Select line & 4 Outputlines

Truth table 1:2 DEMUX

Sel	Y_0	Y_1
0	i	0
1	0	i

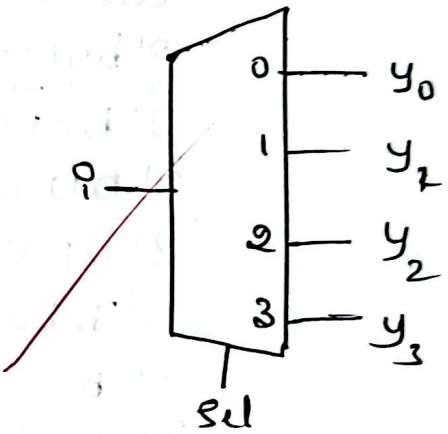
Block Diagram



Truth table 1:4 DEMUX

Sel [0]	Sel [1]	Y_0	Y_1	Y_2	Y_3
0	0	i	0	0	0
0	1	0	i	0	0
1	0	0	0	i	0
1	1	0	0	0	i

Block Diagram



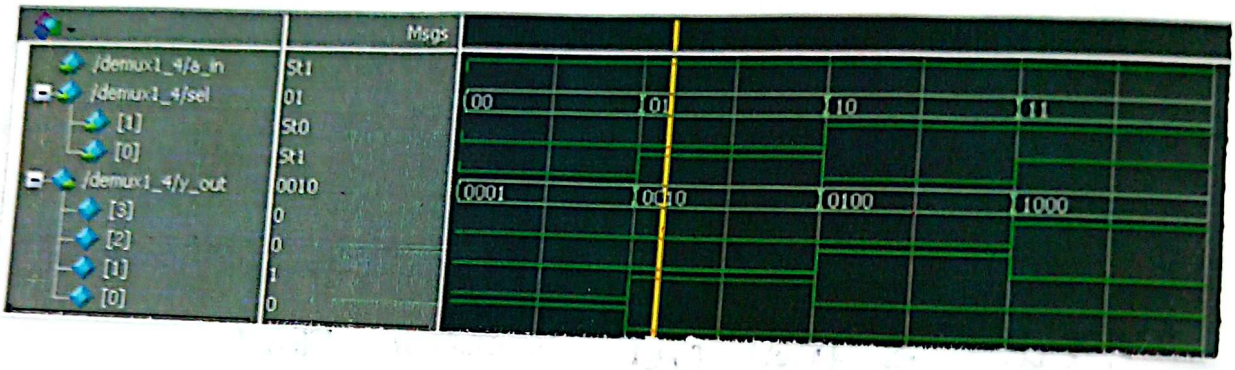
Truth table 1 to 4 DEMUX

input			output			
S_1	S_2	a-in	Y_3	Y_2	Y_1	Y_0
0	0	1	0	0	0	1
0	1	1	0	0	1	0
1	0	1	0	1	0	0
1	1	1	1	0	0	0

TITLE

Program On demux

```
module demux1_4(a_in, sel, y);
input a_in;
input [1:0]sel;
output [3:0] y;
reg [3:0]y;
always@(a_in, sel)
begin
case(sel)
2'b00 = begin y[0]=a_in;
  y[1] = 1'b0;
  y[2] = 1'b0;
  y[3] = 1'b0;
  end
  2'b01= begin y[0]=1'b0;
  y[1] = a_in;
  y[2] = 1'b0;
  y[3] = 1'b0;
  end
  2'b10= begin y[0]=1'b0;
  y[1]= 1'b0;
  y[2] = a_in;
  y[3]= 1'b0;
  end
```


Msgs
/demux1_4/a_in
/demux1_4/sel
[1]
[0]
/demux1_4/y_out
[3]
[2]
[1]
[0]
St1
01
St0
St1
0010
0
0
1
0
00
01
10
11
0001
0010
0100
1000

TITLE

```
2'b11 : begin y[0] => 1'b0;
   y[1] = 1'b0;
   y[2] = 1'b0;
   y[3] = a-in;
   end
   default : y = 3'b000;
   endcase
   end
   endmodule.
```

TITLE

Design Verilog program for implement in various types of flips such as SR, JK and D

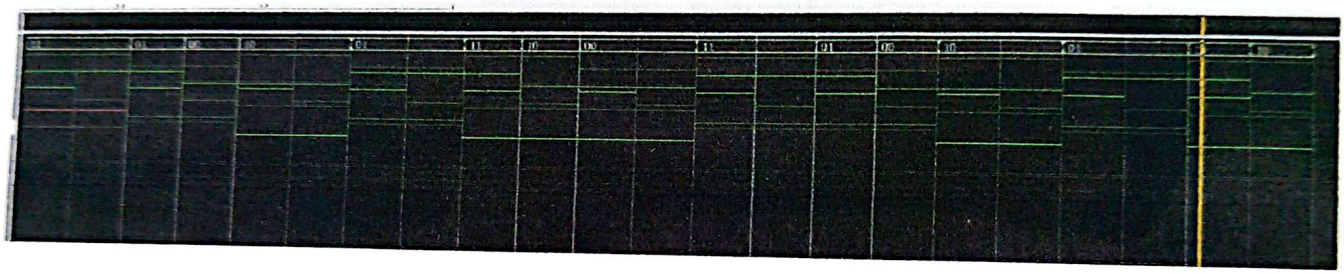
Aim :-

To design and implement various types of flip-flop like D usig verilog HDL

Verilog code for JK flip-flop

```
module JK_FF(JK, clk, q, qb)
input [1:0]jk;
input clk;
output q;
output qb;
always @(posedge clk)
begin
case (jk)
2'b00: q=q;
2'b01 : q=0;
2'b00 ; q=1;
2'b11 : q=~q;
endcase
qb=~q;
end
endmodule
```

JK FLIPFLOP

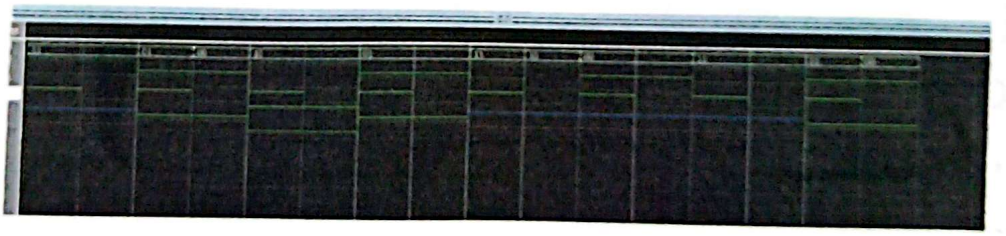


TITLE

Verilog Code for SR-flipflop

```
module SR-ff(sr, clk, q, qb);
 input [1:0]sr;
  input Eclk;
 Output q, qb;
 reg q, qb;
 always @(posedge clk)
 begin
 case(SR)
 2'b00: q=q;
 2'b01: q=0;
 2'b10: q=1;
 2'b11: q=1'bz;
 endcase
 end
 endmodule
```

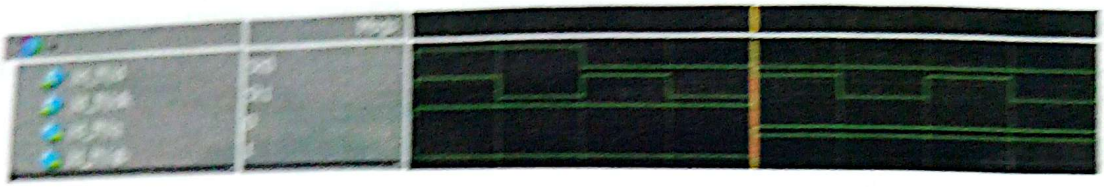
SR FLIPFLOP



TITLE

Verilog Code for D-flip flop

```
module dff(d,clk,q,qb)
 input d;
 input clk;
 output q,qb;
 reg q,qb;
 always@(posedge clk)
 begin
 case(d)
 1'b0: q=0;
 1'b1: q=1;
 endcase
 qb=~q;
 end
 endmodule
```



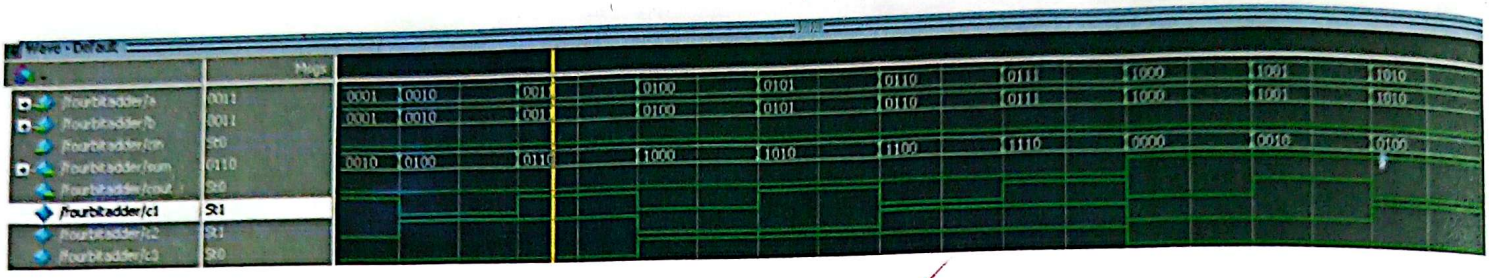
TITLE

Design a ~~bett~~ 4 bit full adder and Subtract -or and Stimulate the Same using basic gates

Aim :- To design 4 bit full adder using logic gates

Verilog Code for 1-bit fulladder

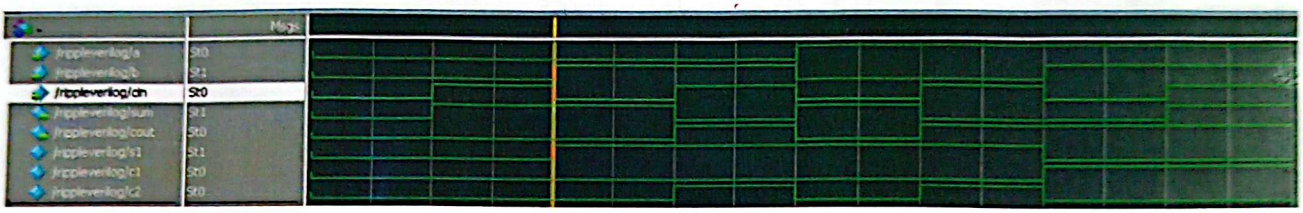
```
module ripplerverilog (sum, cout, a, b, cin);
input a, b, cin;
output sum, cout;
wire s1, c1, c2;
xor (s1, a, b)
xor (sum, s1, cin);
and (c1, a, b)
and (c2, s1, cin)
or (cout, c1, c2);
endmodule
// origin 4 bit ripple adder
module fourbitadder (sum, cout, a, b, cin);
input [3:0] a;
input [3:0] b;
input cin;
output [3:0] sum;
output cout;
wire c1, c2, c3;
ripplerverilog u1 (sum[0], c1, a[0], b[0], cin);
ripplerverilog u2 (sum[1], c2, a[1], b[1], c1);
```



TITLE

```
ripple verilog r3 [sum[2], c3, a[1], b2, c2);
ripple verilog r4 [sum[3], cout, a[3], b[3], c3);
endmodule
```

EXPERIMENT No. :.....



TITLE

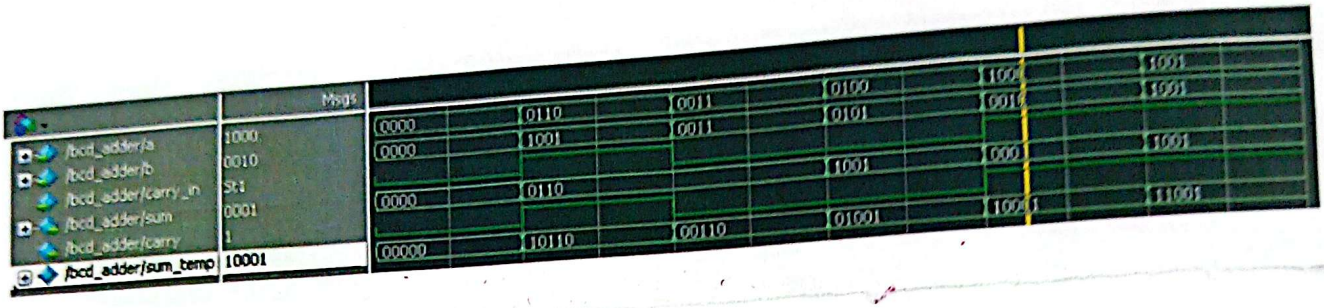
Design verilog HDL in implement Decimal Adder

Aim:

To design a decimal or BCD adder using Verilo HDL

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;
 reg[4:0] sum_temp;
 reg [3:0] sum;
 reg carry;
 always @(a,k,carry_in);
 begin
     sum_temp = a+b+ carry_in;
     if (sum_temp>9) begin
     sum_temp = sum_temp+6;
     carry=1;
      sum= sum_temp [1:0]; end
     else begin
        carry=0;
         sum = sum_temp[3:0]; end
```


Msgs
/bcd_adder/a
/bcd_adder/b
/bcd_adder/carry_in
/bcd_adder/sum
/bcd_adder/carry
/bcd_adder/sum_temp
1000
0010
St1
0001
1
10001
0000
0110
0011
0100
1001
0000
0011
0101
1001
0000
0110
1001
00000
10110
00110
01001

TITLE

Design verilog HDL to implment Simple Circuit using structured, Data flow an behavioural model

Aim:- To design a Simple Circuit using ~~Standerd~~ Structural, data flow and behavioural model

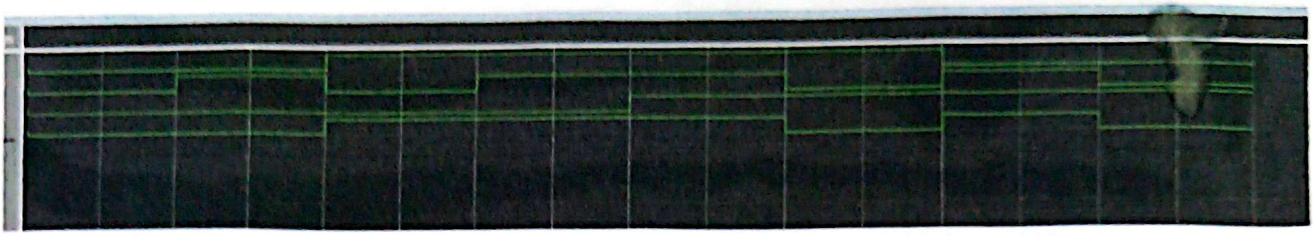
Verilog Code for Structural model

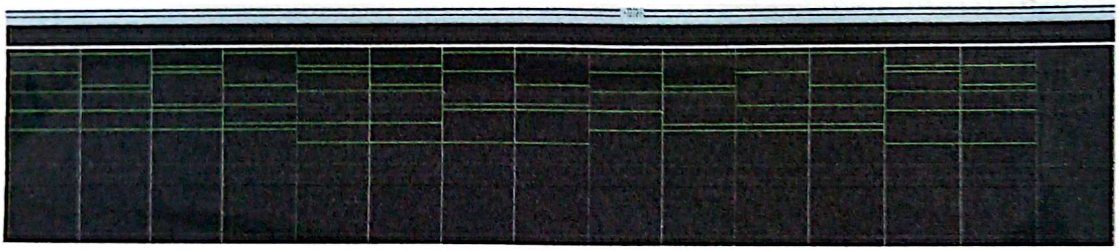
```
module rippleverilog ( sum, cout, a, b, cin);
input a, b, cin;
output sum, cout;
wire s1, c1, c2;
xor (s1, a, b);
xor (sum, s1, cin)
and (c1, a, b);
and (c2, s1, cin);
 or (cout, c1, c2)
endmodule
```

Verilog Code for dataflow model:-

```
module and2 ( x1, x2, z1);
output x1, x2;
output z1;
wire x1, x2;
wire z1;
assign z1 = x1 & x2;
endmodule
```

Behavioural model





TRUCTURAL model

TITLE

Verilog Code for behavioural model

```
module mux2x1 (input I0, I1, S, output reg Y);
always @(*)
begin
    if (S)
    Y = I1;
    else
       Y = I0;
end
endmodule
```

Dataflow model