ACS COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Digital Design and Computer OrganizationLaboratory

(BCS302)

MANUAL

(Effective from the academic year 2023-2024)

SEMESTER – III

Syllabus

PRACTICAL COMPONENT OF IPCC

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

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

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

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

5. Design Verilog HDL to implement Decimal adder.

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

7. Design Verilog program to implement types of De-Multiplexer.

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

Course outcomes (Course Skill Set):

CO's	
CO1	Apply the K–Map techniques to simplify various Boolean expressions
CO2	Design different types of combinational and sequential circuits along with Verilog programs
CO3	Describe the fundamentals of machine instructions, addressing modes and Processor performance
CO4	Explain the approaches involved in achieving communication between processor and I/O devices.
CO5	Analyze internal Organization of Memory and Impact of cache/Pipelining on Processor Performance

List of Program Outcomes

PO1	Apply knowledge of mathematics and science, with fundamentals of Computer Science & Engineering to be able to solve complex engineering problems related to CSE
PO2	Apply mathematical foundations, algorithmic principles, and computer Science theory in the modelling and design of computer based systems in a way that demonstrates comprehension of trade-offs involved in design choices
PO3	Analyze a problem, and identify and define the computing requirements appropriate to its solution
PO4	Design and development principles in the construction of software systems of varying Complexity
PO5	Design, implement, and evaluate a software or a software/hardware system, component, or process to meet desired needs within realistic constraints such as memory, runtime efficiency, as well as appropriate constraints related to economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability considerations
PO6	Use the techniques, skills, and modern engineering tools necessary for practice as a CSE professional
PO7	Work effectively as an individual, and as a member or leader in diverse teams and in multidisciplinary environment.
PO8	Demonstrate knowledge of contemporary issues and understand professional, ethical, legal, security and social issues and responsibilities
PO9	Analyze the local and global impact of computing on individuals, organizations, and society
PO10	Demonstrate knowledge and understanding of the engineering and management principles including financial implications and apply these to his/her work, as a member and leader in a team, and to manage project work as part of a multidisciplinary team.
PO11	Communicate effectively in both verbal and written forms
PO12	Recognize the need for, and be motivated to engage in life-long learning and continuing professional development.

List of Program Specific Outcomes

PSO1	Foundation of mathematical concepts: To use mathematical methodologies to crack problem using suitable mathematical analysis, data structure and suitable algorithm
PSO2	Foundation of Computer System: The ability to interpret the fundamental concepts and methodology of computer systems. Students can understand the functionality of hardware and software aspects of computer systems.
PSO3	Foundations of Software development: the ability to grasp the software development lifecycle and methodologies of software systems. Possess competent skills and knowledge of software design process. Familiarity and practical proficiency with a broad area of programming concepts and provide new ideas and innovations towards research

CO-PO Matrix

CO-PO MATRIX

CO's	PO's												PSO's		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	2	2	1	1	3								1	3	-
CO2	2	2	2	2	3								1	3	-
CO3	2	2	-	-	2								--	3	2
CO4	2	2	1	-	2								-	3	1
CO5	2	2	1	-	1								-	3	1

3 - High Correlation 2 –MediumCorrelation 1– Low Correlation

PROCEDURE:
The Procedure to be followed for Software and Hardware Programs are as follows:

Step 1: Go to Start Menu ->All Programs ->select Xilinx ISE 14.7.

Step 2: Go to File Menu and select Close project to close previously opened project if any, and then Select New Project.

Step 3: Enter the Project name and location and Select the Top level module type as HDL.

Step 4: Select the
Device family- Spartan3E
Device name- xc3s50
Package-TQ144(PQ208)
Speed- (-4)
Top level Source Type- HDL
Synthesis Tool XST(VHDL/Verilog)
Simulator –Isim(VHDL/Verilog)
Preferred Language- Verilog
VHDL Source analysis standard – VHDL 93

Click on NEXT and Finish

Step 5: Right click on the source file and select new source followed by Verilog module and Give the file name same as the name of the entity.

Step 6: Define the ports used and their respective directions in the next window that opens.

Step 7: Write the architecture body and the generics etc.

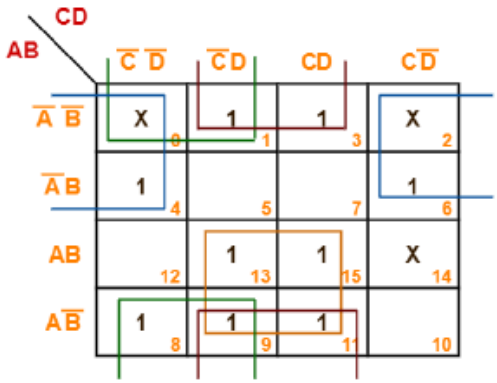
Step 8: Go to the **Process view window** and right click on the Synthesize - XST and Select Run. Correct the errors if any.

Step 9: Go to Process view and under Xilinx ISE Simulator Right click on the Simulate Behavioral model to see the output for the input conditions.

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

We can minimize Boolean expressions of 3, 4 variables very easily using K-map without using any Boolean algebra theorems. K-map can take two forms Sum of Product (SOP) and Product of Sum (POS) according to the need of problem. K-map is table like representation but it gives more information than TRUTH TABLE. We fill grid of K-map with 0's and 1's then solves it by making groups.

$$F(ABCD)= \sum m\,(1,3,4,6,8,9,11,13,15) + d\,(0,2,14)$$



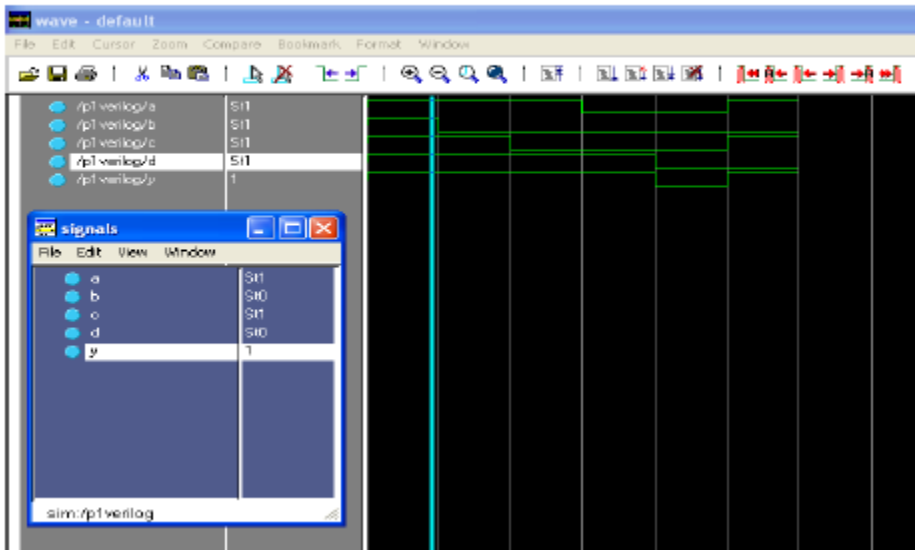
F(ABCD)= A'D' + B'C'+B'D+AD OR F(ABCD)= A'D' + B'C'+A'B'+AD

Truth Table to be Verified: -

SL NO	Decimal Value	MINTERMS	m terms	A	B	C	D	A'	B'	C'	D'	A'D'	B'C'	B'D	AD	F(ABCD)=A'D'+B'C'+B'D+AD
1	0	A'B'C'D'	m0	0	0	0	0	1	1	1	1	1	1	1	0	1
2	1	A'B'C'D	m1	0	0	0	1	1	1	1	0	0	1	0	0	1
3	2	A'B'CD'	m2	0	0	1	0	1	1	0	1	1	0	1	0	1
4	3	A'B'CD	m3	0	0	1	1	1	1	0	0	0	0	1	0	1
5	4	A'BC'D'	m4	0	1	0	0	1	0	1	1	1	0	0	0	1
6	5	A'BC'D	m5	0	1	0	1	1	0	1	0	0	0	0	0	0
7	6	A'BCD'	m6	0	1	1	0	1	0	0	1	1	0	0	0	1
8	7	A'BCD	m7	0	1	1	1	1	0	0	0	0	0	0	0	0
9	8	AB'C'D'	m8	1	0	0	0	0	1	1	1	0	1	1	0	1
10	9	AB'C'D	m9	1	0	0	1	0	1	1	0	0	1	0	1	1
11	10	AB'CD'	m10	1	0	1	0	0	1	0	1	0	0	0	0	0
12	11	AB'CD	m11	1	0	1	1	0	1	0	0	0	0	0	1	1
13	12	ABC'D'	m12	1	1	0	0	0	0	1	1	0	0	0	0	0
14	13	ABC'D	m13	1	1	0	1	0	0	1	0	0	0	0	1	1
15	14	ABCD'	m14	1	1	1	0	0	0	0	1	0	0	0	0	0
16	15	ABCD	m15	1	1	1	1	0	0	0	0	0	0	0	1	1

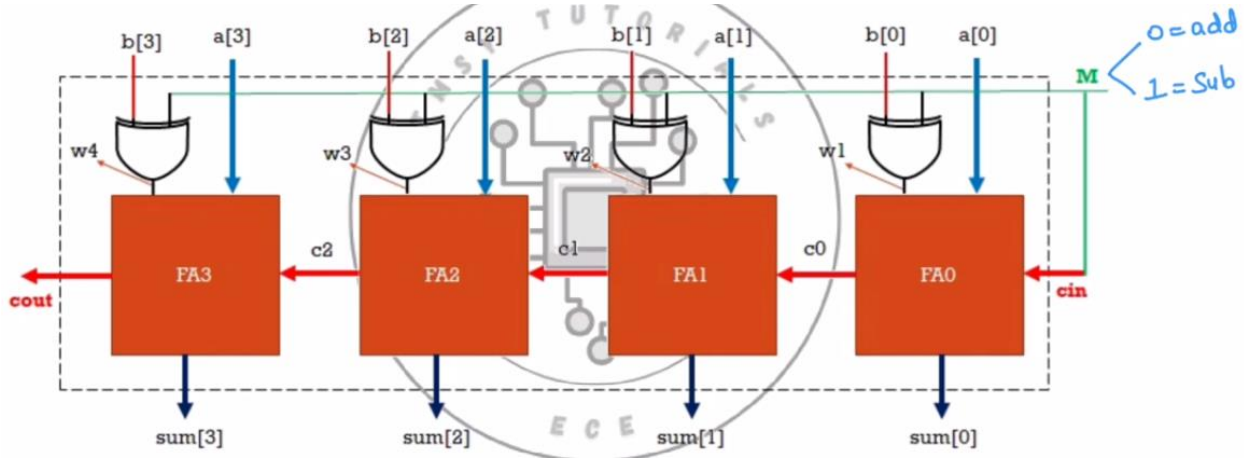
```
module p1verilog(a,b,c,d,y);
    input a;
    input b;
    input c;
    input d;
    output y;

        reg y;
        always @ (a, b,c,d)
        begin
        y= (~a & ~d) | (~b & ~c) | (~b & d) | (a & d);
        end
        endmodule
```

OUTPUT:



Experiment 2

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

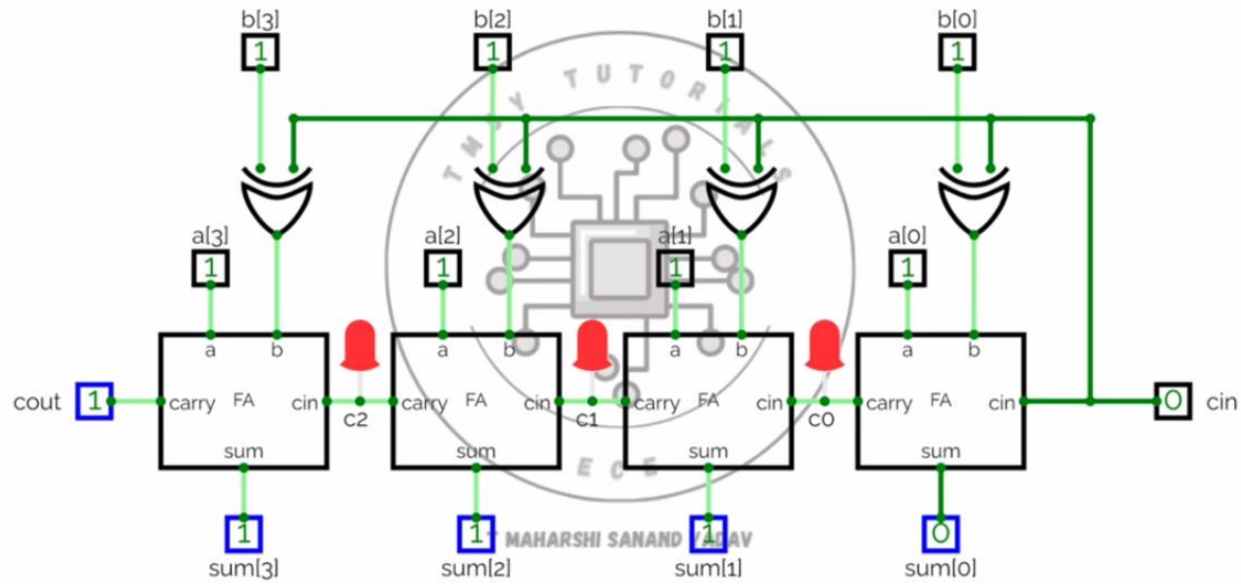


- A 4-bit binary adder and subtractor is a digital circuit that can perform both addition and subtraction of two 4-bit binary numbers.
- To implement a 4-bit binary adder and subtractor, we can use a combination of a 4-bit ripple carry adder and a 4-bit ripple carry subtractor.
- The 4-bit ripple carry adder takes two 4-bit binary numbers as inputs and produces a 4-bit binary sum output. The 4-bit ripple carry subtractor takes two 4-bit binary numbers as inputs and produces a 4-bit binary difference output.

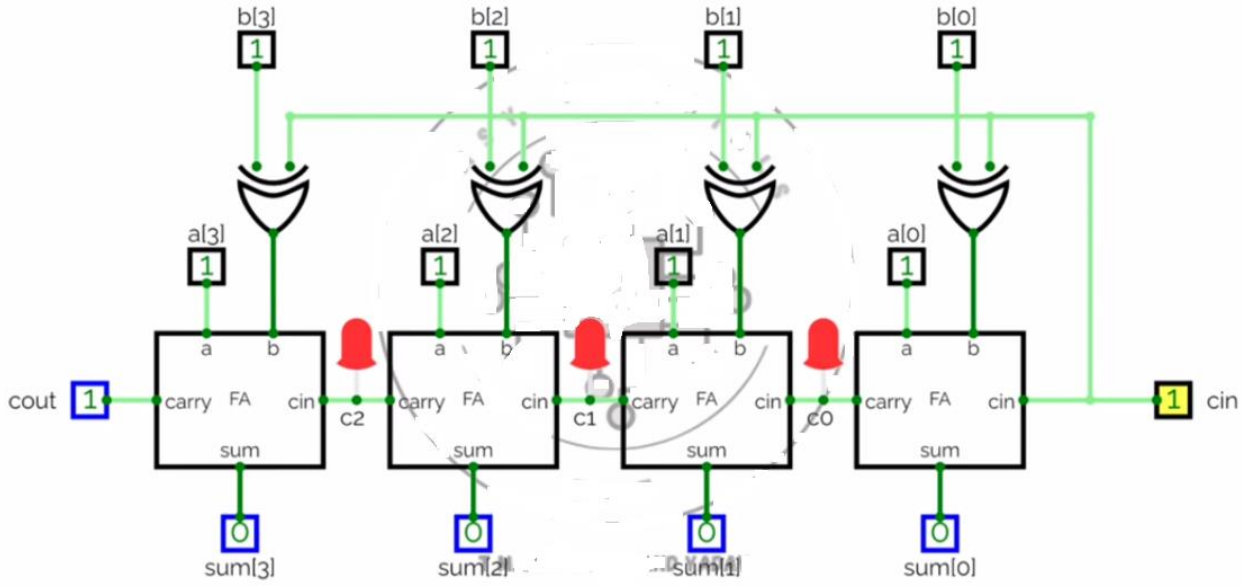
- To perform addition or subtraction using the same circuit, we can use an XOR gate to switch between the two modes of operation.
- When the XOR input is 0, the circuit will perform addition, and
- when the XOR input is 1, the circuit will perform subtraction.
- In this circuit, a3-a0 and b3-b0 are the two 4-bit binary numbers to be added or subtracted. s3-s0 is the 4-bit binary output of the adder or subtractor.
- The XOR gate selects between addition and subtraction modes based on the value of the XOR input.
- c2-c0 are the carry inputs to the full adders in the adder and subtractor circuits.
- The operation of the 4-bit binary adder and subtractor can be understood as follows:
- **FOR ADDITION MODE:**
  - The 4-bit binary numbers a3-a0 and b3-b0 are added using a 4-bit ripple carry adder to produce a 4-bit binary sum output s3-s0.
  - The XOR input is set to '0' to select the addition mode, and the circuit performs addition.
- **FOR SUBTRACTION MODE:**
  - The 4-bit binary numbers a3-a0 and b3-b0 are subtracted using a 4-bit ripple carry subtractor to produce a 4-bit binary difference output s3-s0.
  - The XOR input is set to '1' to select the subtraction mode, and the circuit performs subtraction.

Note that when performing subtraction, we need to use two's complement representation of the second input number (B) to obtain its negative value, which is added to the first input number (A) to perform subtraction.

When Cin = 0 it will Perform Addition

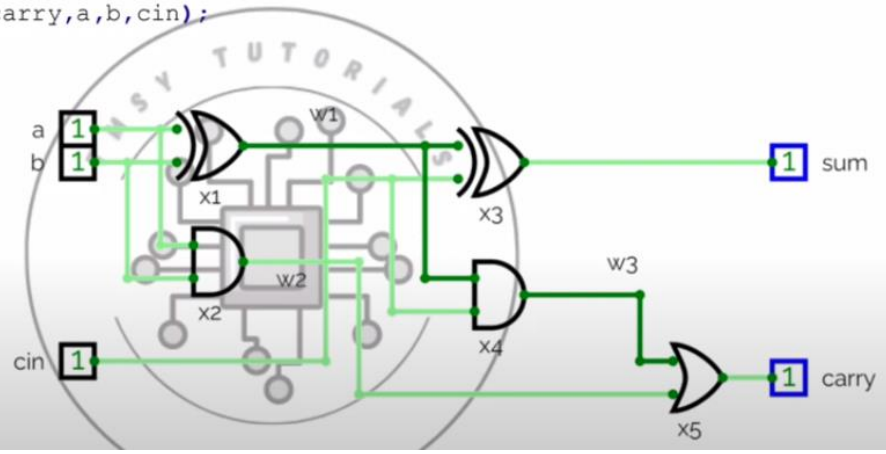


When Cin = 1 it will Perform Subtraction



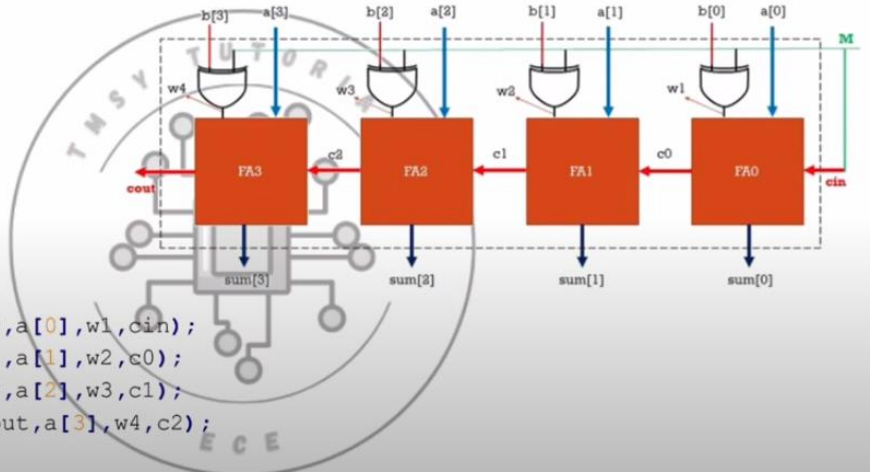
Step 1: 1 – Bit Full Adder Verilog Source code

```
module fulladder(sum,carry,a,b,cin);
output sum;
output carry;
input a;
input b;
input cin;
wire w1,w2,w3;
xor x1(w1,a,b);
and x2(w2,a,b);
xor x3(sum,w1,cin);
and x4(w3,w1,cin);
or x5(carry,w2,w3);
endmodule
```



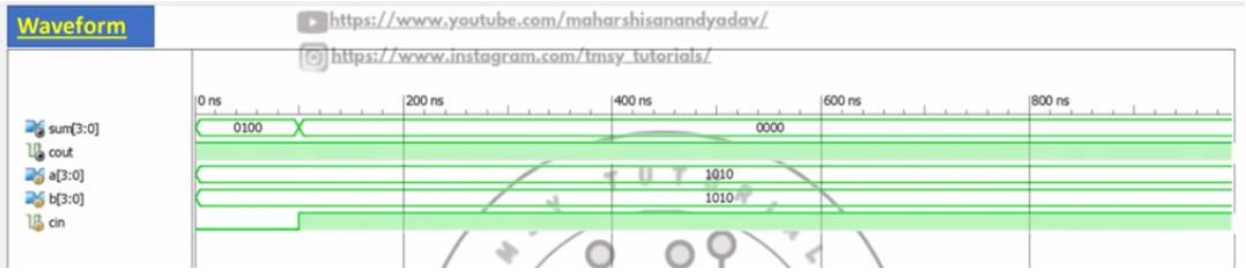
Step 2: Binary Adder cum Subtractor – Source Code

```
module binary_adder_subtractor(sum,cout,a,b,cin);
output [3:0]sum;
output cout;
input [3:0]a,b;
input cin;
wire c0,c1,c2;
wire w1,w2,w3,w4;
xor n1(w1,b[0],cin);
xor n2(w2,b[1],cin);
xor n3(w3,b[2],cin);
xor n4(w4,b[3],cin);
fulladder FA0(sum[0],c0,a[0],w1,cin);
fulladder FA1(sum[1],c1,a[1],w2,c0);
fulladder FA2(sum[2],c2,a[2],w3,c1);
fulladder FA3(sum[3],cout,a[3],w4,c2);
endmodule
```



Step 3: Binary Adder cum Subtractor – Test Bench

```
module binary_adder_subtractor_TB;
    // Inputs
    reg [3:0] a;
    reg [3:0] b;
    reg cin;
    // Outputs
    wire [3:0] sum;
    wire cout;
binary_adder_subtractor hhh(sum,cout,a,b,cin);
    initial
    begin
        $monitor($time,"a=%b,b=%b,cin=%b,cout=%b,sum=%b",a,b,cin,cout,sum);
        a=4'd10;//1010
        b=4'd10;//1010
        cin=0;

        #100;
        a=4'd10;
        b=4'd10;
        cin=1;
    end
endmodule
```



Experiment 3

Design a Verilog HDL to implement simple circuits using structral , Dataflow and Behavaoural model

Data flow modeling

Logical Expression

The equation for 2:1 mux is:

Y = D0.S' + D1.S

where Y is the final output, D0, D1, and S are inputs.

```
module m21(D0, D1, S, Y);
output Y;
input D0, D1, S;
assign Y=(S)?D1:D0;
endmodule
```

Behavioral modeling

This level describes the behavior of a digital system. In most of the cases, we code the behavioral model using the truth table of the circuit.

Truth-table for 2:1 MUX

Select line	Input		Output
S	**D0**	**D1**	**Y**
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	1

Truth Table for 2:1 MUX

Now to find the expression, we will use K- map for final output Y.

Equation from the truth table: Y = D0.S' + D1.S

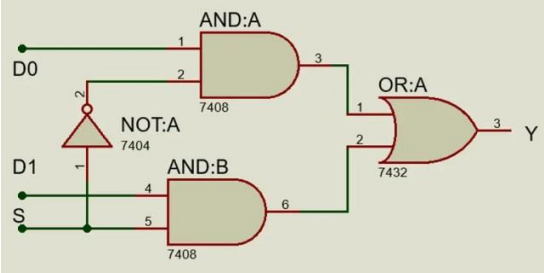
```
module m21( D0, D1, S, Y);
input wire D0, D1, S;
output reg Y;
always @(D0 or D1 or S)
begin
if(S)
Y= D1;
```

```
else
Y=D0;
end
endmodule
```

Structural modeling

Logic circuit

Now the logical diagram for a 2:1 MUX shows that we need two AND gates, one OR gate and one NOT gate. We'll structurize for each of the gates separately.



Verilog code for 2:1 MUX using structural modeling

```
module and_gate(output a, input b, c);
assign a = b & c;
endmodule

module not_gate(output d, input e);
assign d = ~ e;
endmodule

module or_gate(output l, input m, n);
assign l = m | n;
endmodule

module m21(Y, D0, D1, S);
output Y;
input D0, D1, S;
wire T1, T2, T3;
and_gate u1(T1, D1, S);
not_gate u2(T2, S);
and_gate u3(T3, D0, T2);
or_gate u4(Y, T1, T3);
endmodule
```

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

Half adder: -

BOOLEAN EXPRESSIONS:

sum=A ⊕ B

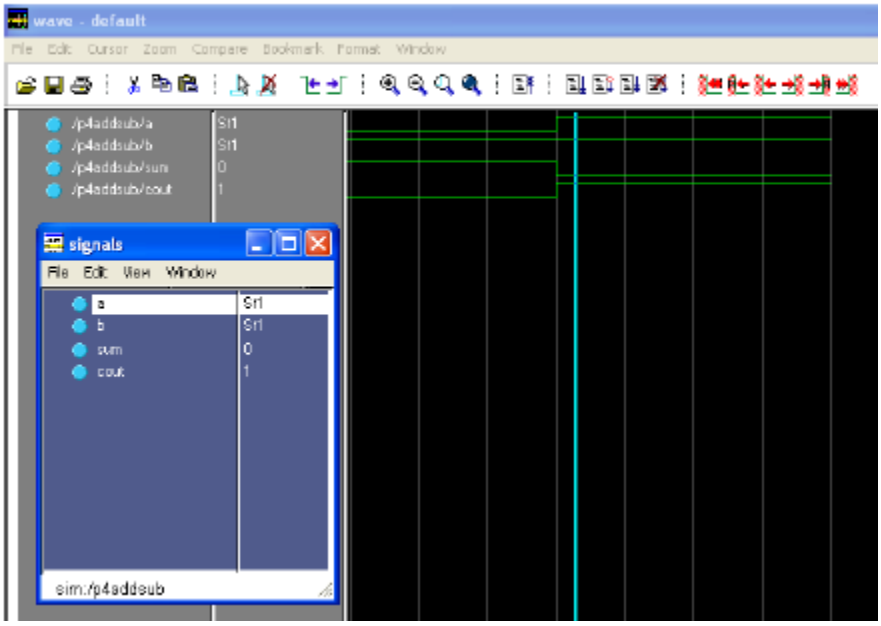
cout=A B

TRUTH TABLE

INPUTS		OUTPUTS	
A	B	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
module p4addsub(a,b,sum,cout);
   input a;
   input b;
   output sum;
   output cout;

reg sum, cout;
always @(a,b)
begin
sum = a ^ b;
cout = a & b;
end
endmodule
```

OUTPUT: -



Half Subtractor

BOOLEAN EXPRESSIONS:

Diff = A ⊕ B

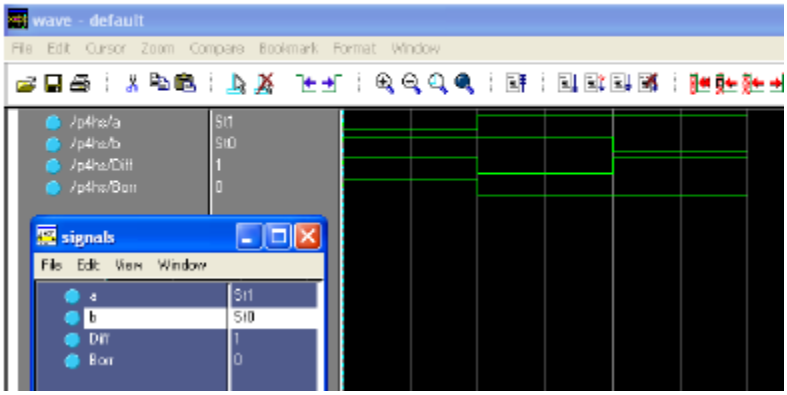
Borr = $\bar{A}$B

TRUTH TABLE

INPUTS		OUTPUTS	
A	B	Diff	Borr
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

```
module p4hs(a,b,Diff,Borr);
   input a;
   input b;
   output Diff;
   output Borr;

reg Diff, Borr;
always @(a,b)
begin
Diff = a ^ b;
Borr = (~ a) & b;
end
endmodule
```

OUTPUT: -



Full Adder

BOOLEAN EXPRESSIONS

Sum= A ⊕ B ⊕ Cin

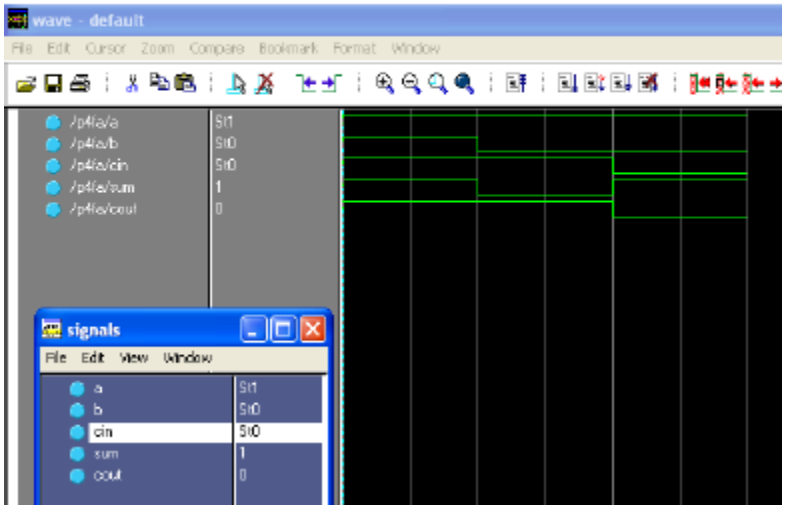
Cout=A B + B Cin + A Cin

TRUTH TABLE

INPUTS			OUTPUTS	
A	B	Cin	sum	cout
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

```
module p4fa(a,b,cin,sum,cout);
   input a;
   input b;
   input cin;
   output sum;
   output cout;
reg sum, cout;
always @(a,b,cin)
begin
sum = a^b^cin;
cout= (a & b) | (b & cin) | (a & cin) ;
end
endmodule
```

OUTPUT: -



Full Subtractor

BOOLEAN EXPRESSIONS

Diff=A⊕B⊕C

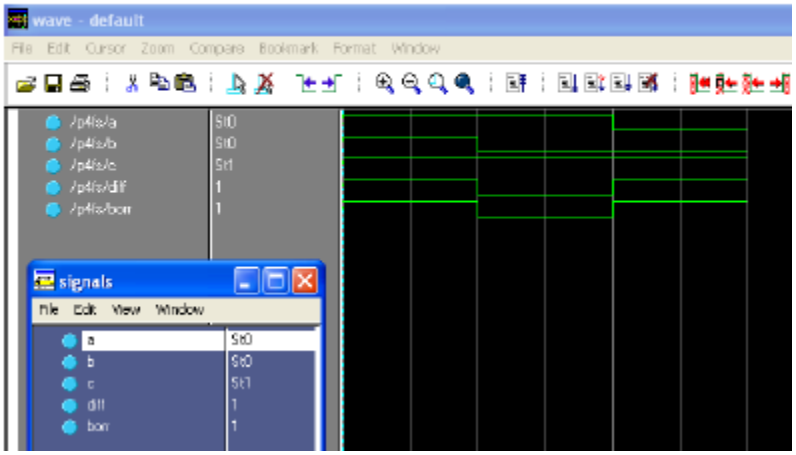
borr= $\bar{A}$ B + B C + $\bar{A}$ C

TRUTH TABLE

INPUTS			OUTPUTS	
A	B	C	diff	borr
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

```
module p4fs(a,b,c,diff,borr);
   input a;
   input b;
   input c;
   output diff;
   output borr;

reg diff, borr;
always @(a,b,c)
begin
diff = a^b^c;
borr= (~a & b) | (b & c) | (~a & c) ;
end
endmodule
```

OUTPUT: -

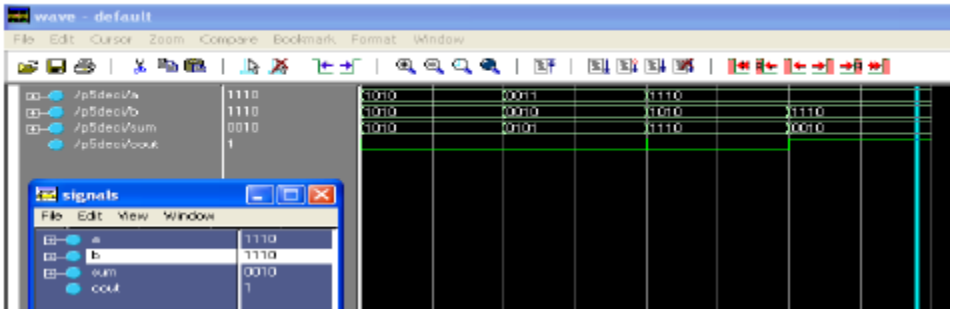

wave - default
File Edit Cursor Zoom Compare Bookmark Format Window
/p4fa/a
/p4fa/b
/p4fa/c
/p4fa/dif
/p4fa/bor
St0
St0
St1
1
1
signals
File Edit View Window
a
b
c
dif
bor
St0
St0
St1
1
1

5.Design Verilog HDL to implement Decimal adder.

This Verilog module, "DecimalAdder," takes two 4-bit decimal inputs A and B and produces a 4-bit sum (Sum) and a carry-out (CarryOut) output. The logic inside the "always" block performs decimal addition with carry propagation, and it also handles the case when the result is greater than 9. In such cases, it adds 6 to the result and updates the carry accordingly.

```
module p5deci(a,b,sum,cout);
  input [3:0] a;
  input [3:0] b;
  output [3:0] sum;
  output cout;

    reg [3:0] sum;
    reg cout;
    always@ (a,b)
    begin
    {cout,sum} = a+b;
    if(a>9 || b>9 || sum>9)
    begin
    {cout,sum} = sum+6;
    end
    end
    endmodule
```

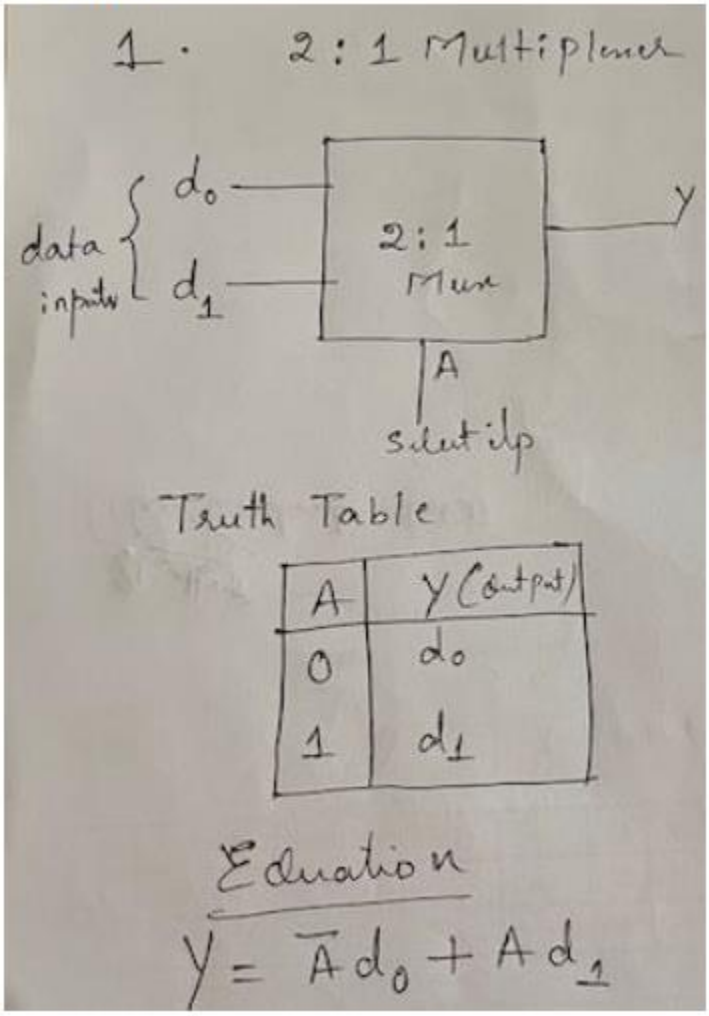
OUTPUT: -



Experiment 6

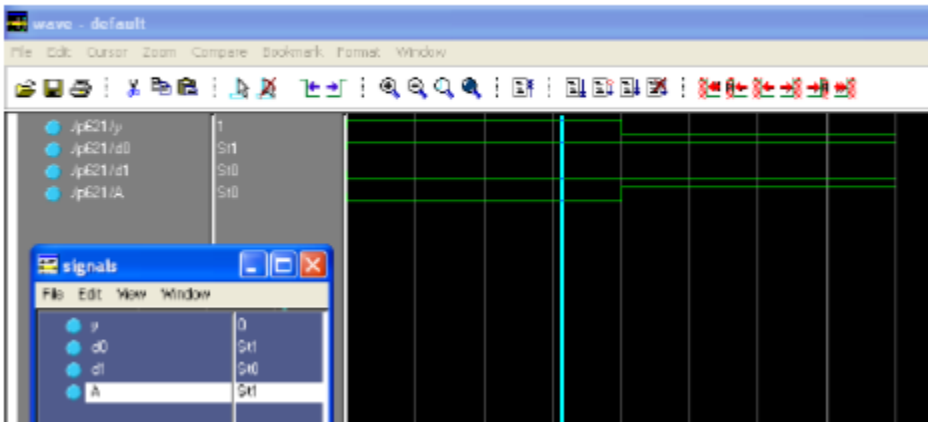
Design Verilog program to implement different types of multiplexer like 2:1, 4:1 and 8:1

2:1 Mux

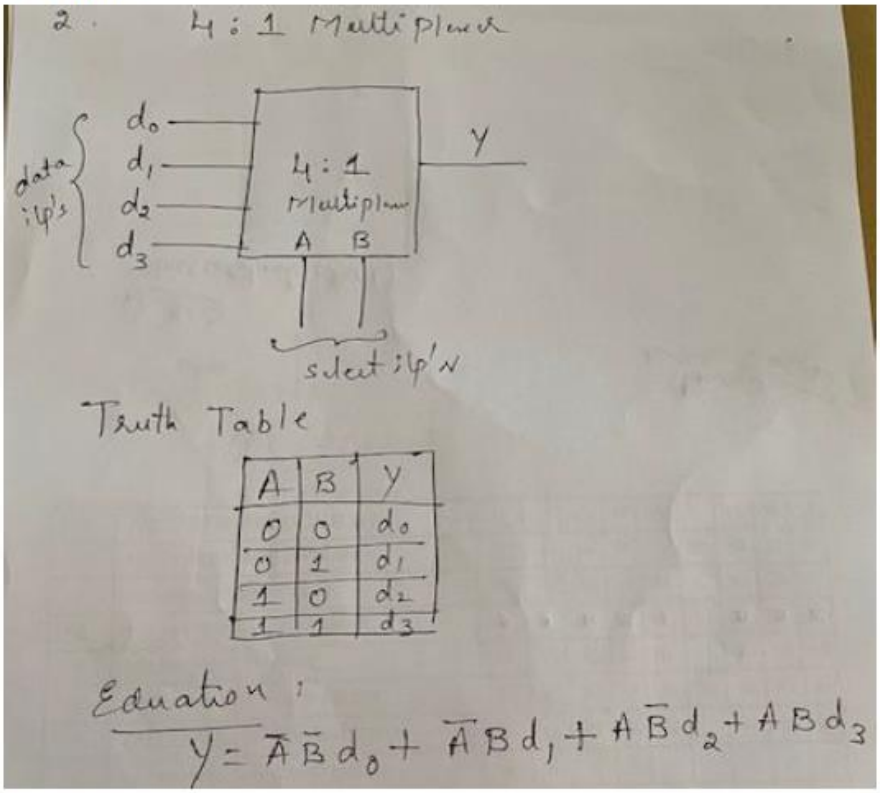


```
module p621(y,d0,d1,A);
   output y;
   input d0;
   input d1;
   input A;
   reg y;
   always @ (d0,d1,A)
   begin
   y=((~A & d0)|(A & d1));
   end
   endmodule
```

output: -

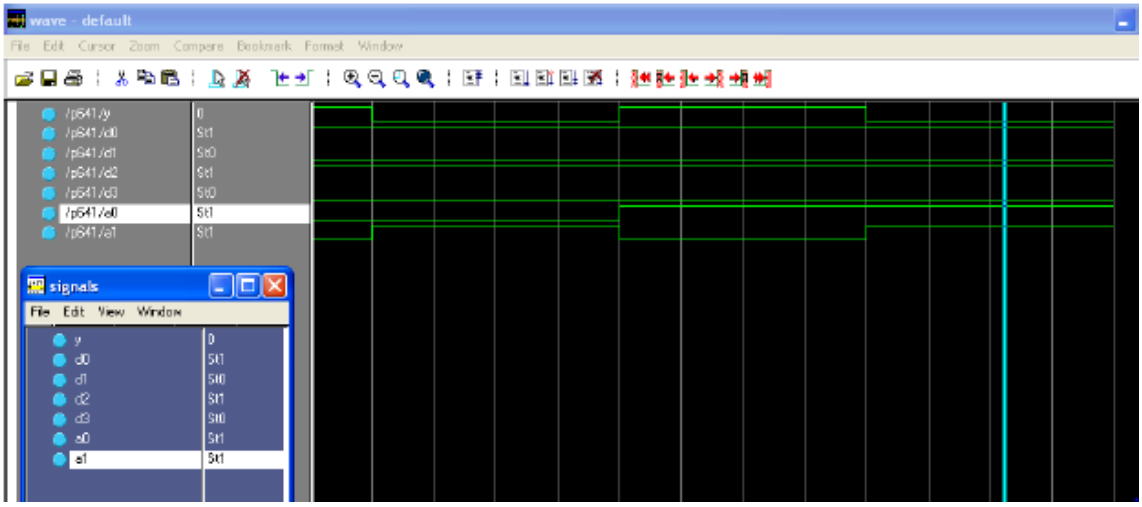


4:1 Mux

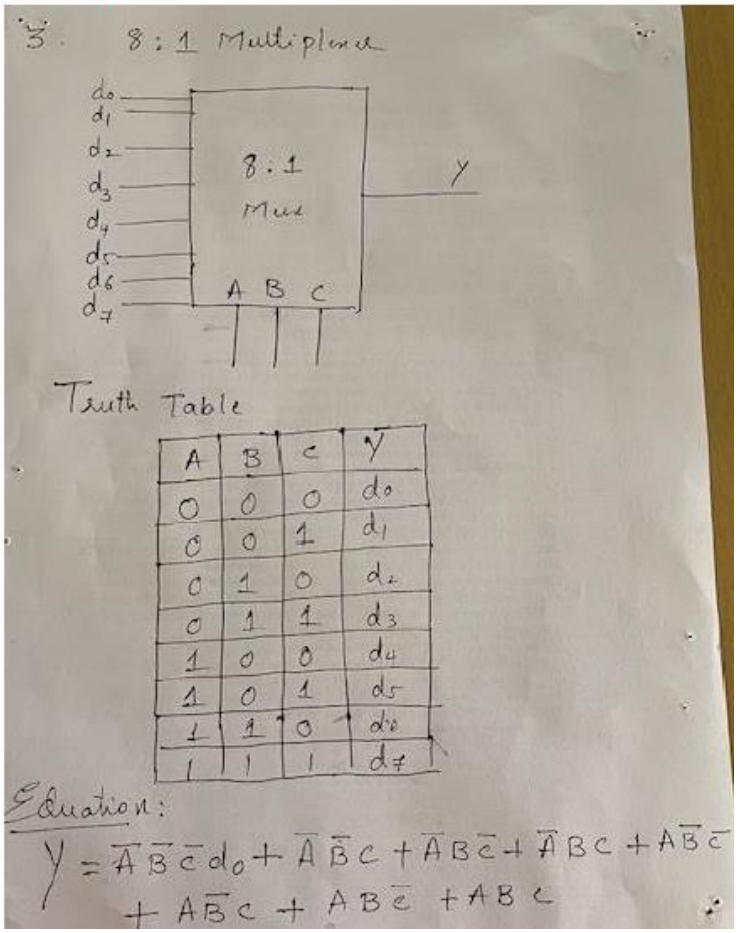



```
module p641(y,d0,d1,d2,d3,a0,a1);
  output y;
  input d0;
  input d1;
  input d2;
  input d3;
  input a0;
  input a1;

  reg y;
  always @ (d0,d1,d2,d3,a0,a1)
  begin
  y= (~a0 & ~a1 & d0) | (~a0 & a1 & d1) | (a0 & ~a1 & d2) | (a0 & a1 & d3);
  end
  endmodule
```

OUTPUT: -

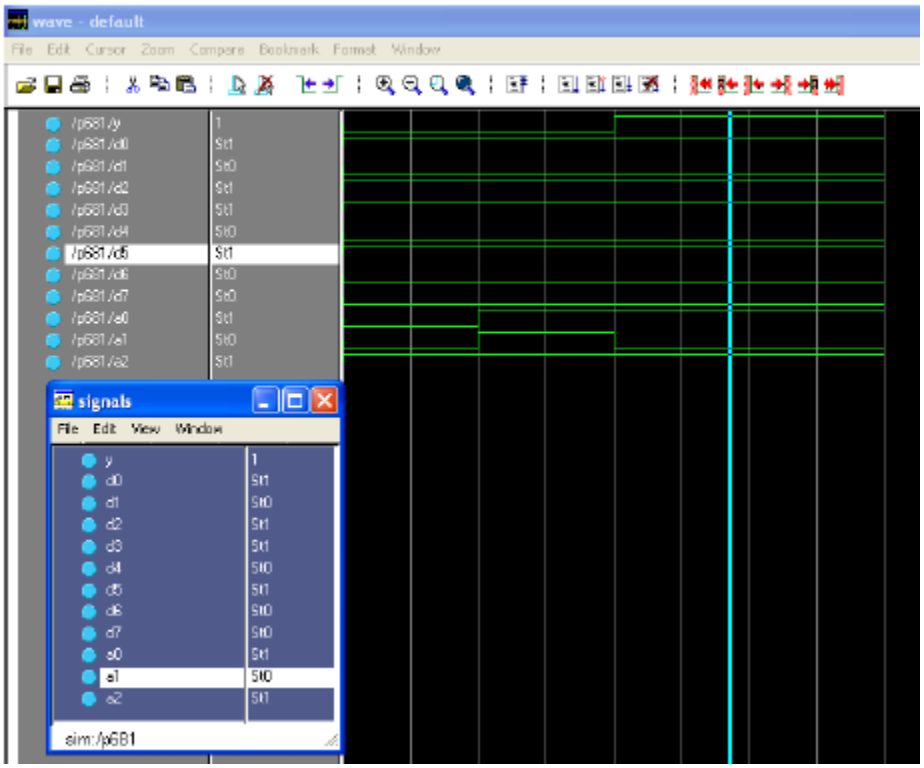


8:1 Mux



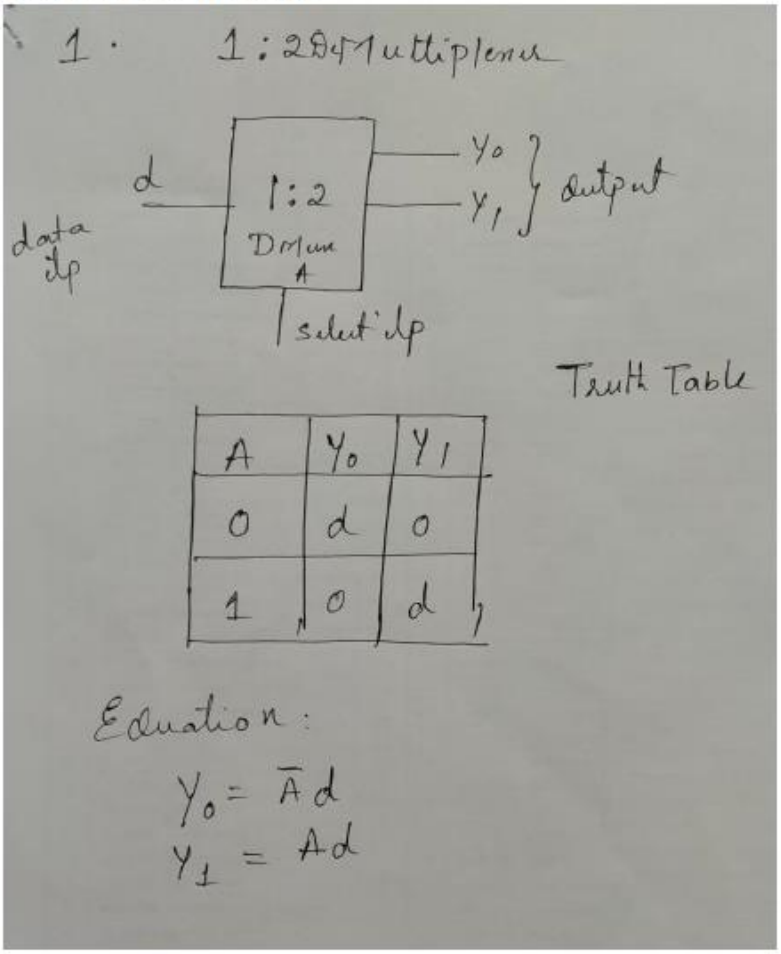
```
module p681(y,d0,d1,d2,d3,d4,d5,d6,d7,a0,a1,a2);
  output y;
  input d0;
  input d1;
  input d2;
  input d3;
  input d4;
  input d5;
  input d6;
  input d7;
  input a0;
  input a1;
  input a2;

    reg y;
    always @ (d0,d1,d2,d3,d4,d5,d6,d7,a0,a1,a2)
    begin

y=  (~a0 & ~a1 & ~a2 & d0 ) |
    (~a0 & ~a1 & a2 & d1) |
    (~a0 & a1 & a2 & d2)   |
    ( ~a0 & a1 & a2 & d3)  |
    (a0 & ~a1 & ~a2 & d4) |
    (a0 & ~a1 & a2 & d5)   |
    (a0 & a1 & ~a2 & d6)   |
    (a0 & a1 & a2 & d7) ;
    end
    endmodule
```

OUTPUT: -


wave - default
File Edit Cursor Zoom Compare Bookmark Format Window
signals
File Edit View Window
sim:/p681

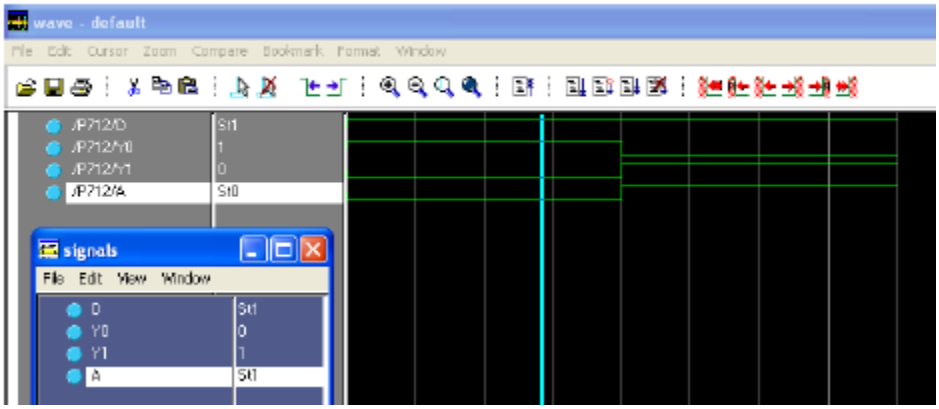
7 Design a verilog program to implement types of de-multiplxer

1:2 Demultiplexer



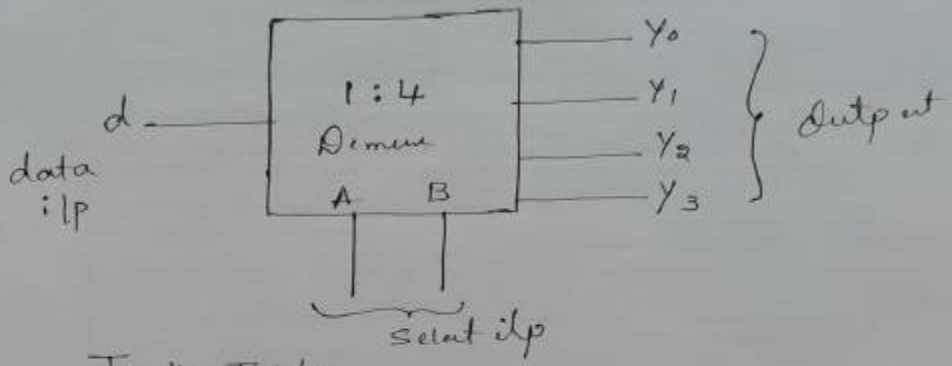

```
module P712(D,Y0,Y1,A);
input D;
output Y0;
output Y1;
input A;

    reg Y0,Y1;
    always @ (A,D)
    begin
    Y0=(~A & D);
    Y1=(A & D);
    end
    endmodule
```

OUTPUT: -



1:4 Demultiplexer

2: 1 : 4 De Multiplexer



Truth Table

A	B	Y_0	Y_1	Y_2	Y_3
0	0	d	0	0	0
0	1	0	d	0	0
1	0	0	0	d	0
1	1	0	0	0	d

Equation;

$Y_0 = \overline{A}\,\overline{B}\,d$

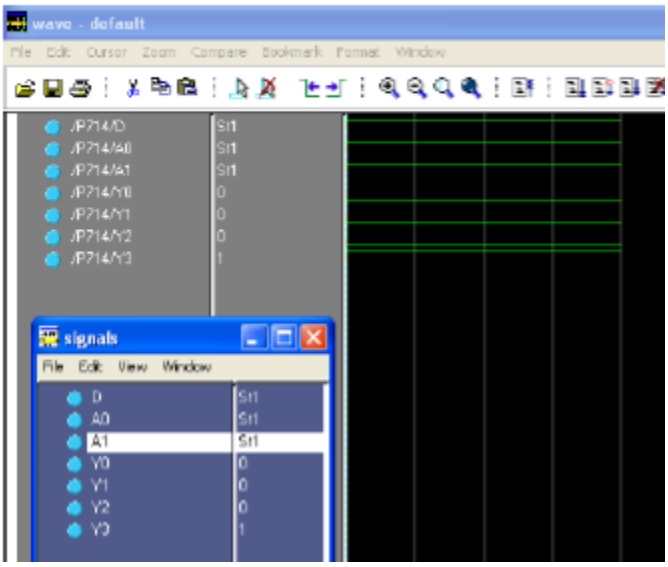
$Y_1 = \overline{A}\,B\,d$

$Y_2 = A\,\overline{B}\,d$

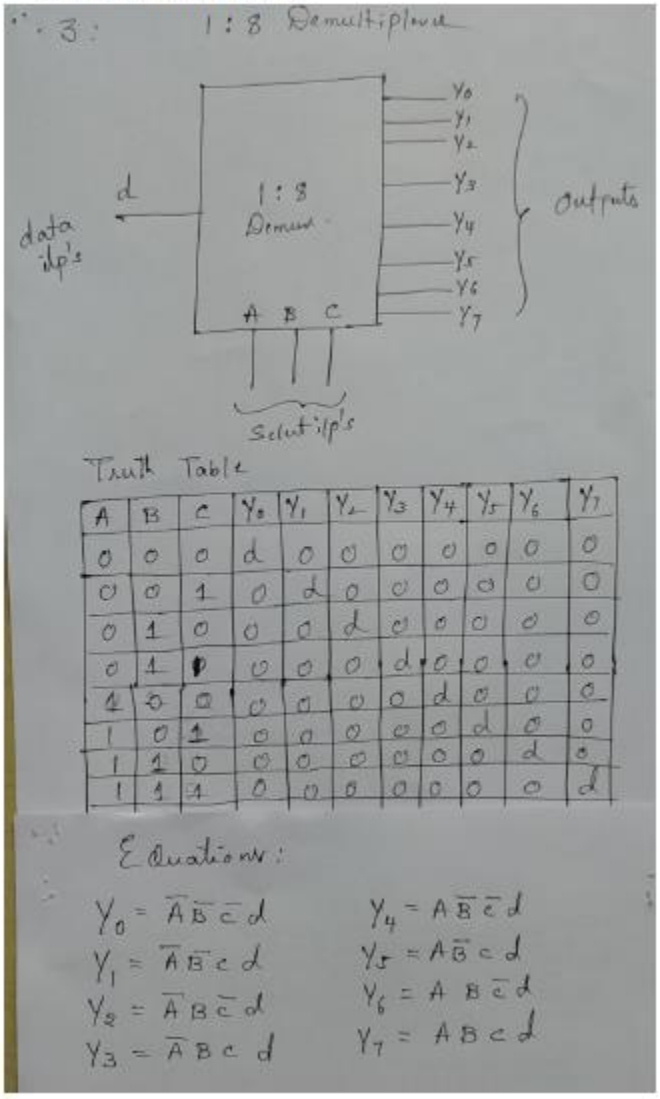
$Y_3 = A\,B\,d$

```
module P714(D,A0,A1,Y0,Y1,Y2,Y3);
  input D;
  input A0;
  input A1;
  output Y0;
  output Y1;
  output Y2;
  output Y3;

    reg Y0,Y1,Y2,Y3;
    always @ (A0,A1,D)
    begin
    Y0=(~A0 & ~A1 & D);
    Y1=(~A0 & A1 & D);
    Y2=(A0 & ~A1 & D);
    Y3=(A0 & A1 & D);
    end
    endmodule
```

OUTPUT: -



1:8 Demultiplexer

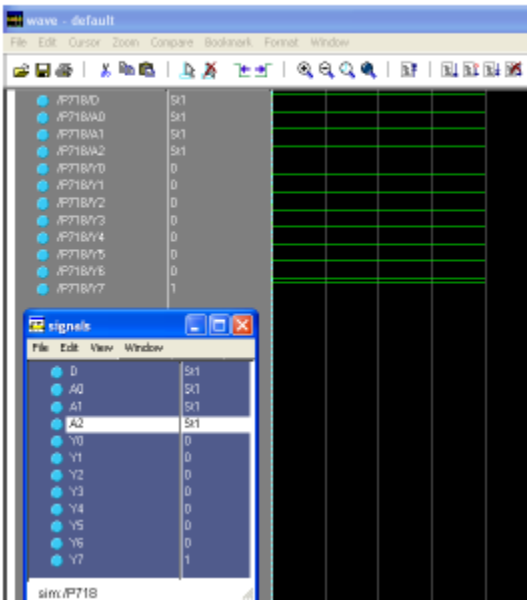


A	B	C	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
0	0	0	d	0	0	0	0	0	0	0
0	0	1	0	d	0	0	0	0	0	0
0	1	0	0	0	d	0	0	0	0	0
0	1	1	0	0	0	d	0	0	0	0
1	0	0	0	0	0	0	d	0	0	0
1	0	1	0	0	0	0	0	d	0	0
1	1	0	0	0	0	0	0	0	d	0
1	1	1	0	0	0	0	0	0	0	d

```
module P718(D,A0,A1,A2,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7);
   input D;
   input A0;
   input A1;
   input A2;
   output Y0;
   output Y1;
   output Y2;
```

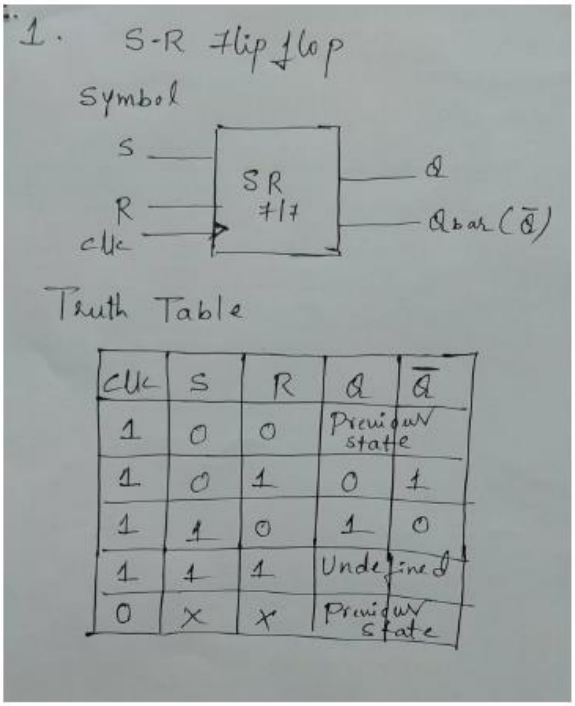
```
output Y3;
output Y4;
output Y5;
output Y6;
output Y7;

reg Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7;
always @ (A0,A1,A2,D)
begin
Y0=(~A0 & ~A1 & ~A2 & D);
Y1=(~A0 & ~A1 & A2 & D);
Y2=(~A0 & A1 & ~A2 & D);
Y3=(~A0 & A1 & A2 & D);
Y4=(A0 & ~A1 & ~A2 & D);
Y5=(A0 & ~A1 & A2 & D);
Y6=(A0 & A1 & ~A2 & D);
Y7=(A0 & A1 & A2 & D);
end
endmodule
```

OUTPUT: -



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

SR FLIP-FLOP

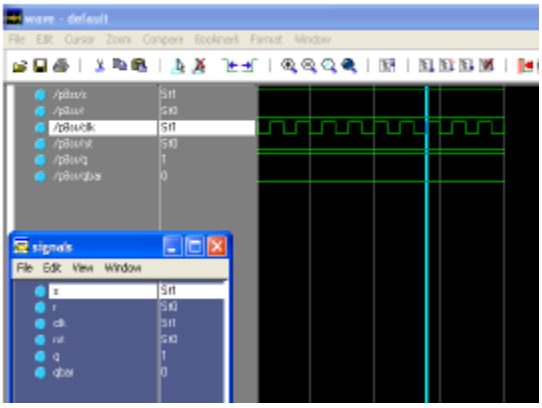


clk	S	R	Q	Q̄
1	0	0	Previous state	
1	0	1	0	1
1	1	0	1	0
1	1	1	Undefined	
0	x	x	Previous state	

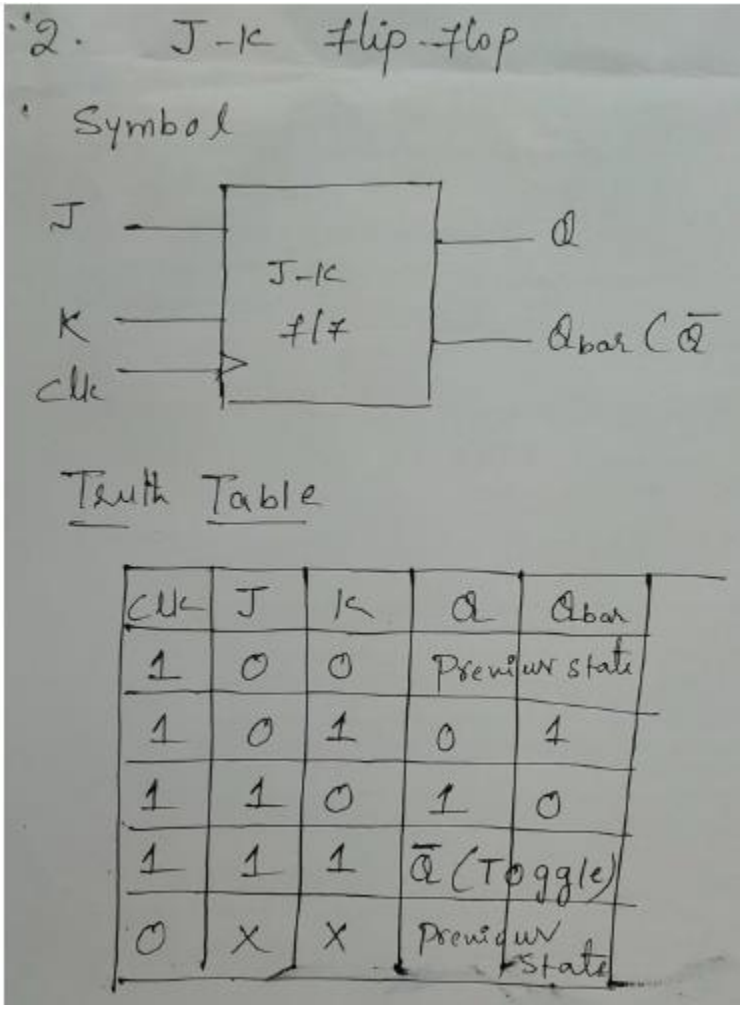
```
module p8sr(s,r,clk,rst,q,qbar);
   input s;
   input r;
   input clk;
   input rst;
   output q;
   output qbar;
```

```
reg q,qbar;
always@(posedge clk)
begin
if(rst)
q<=1'b0;
else if (s==1'b0 && r==1'b0) q<=q;
else if (s==1'b0 && r==1'b1) q<=1'b0;
else if (s==1'b1 && r==1'b0) q<=1'b1;
else if (s==1'b1 && r==1'b1) q<=1'bx;
assign qbar=~q;
end
endmodule
```

OUTPUT: -



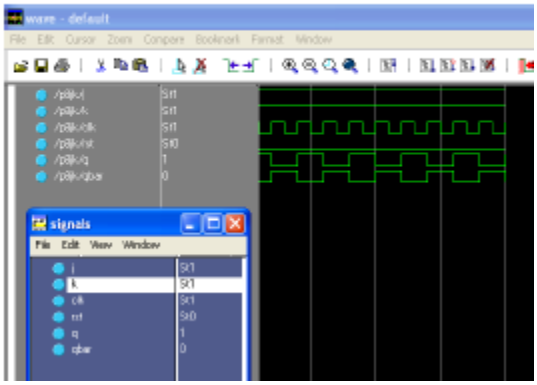
JK FLIP-FLOP

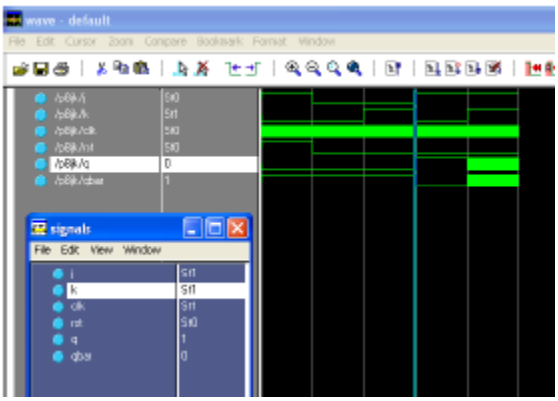



```
module p8jk(j,k,clk,rst,q,qbar);
   input j;
   input k;
   input clk;
   input rst;
   output q;
   output qbar;
```

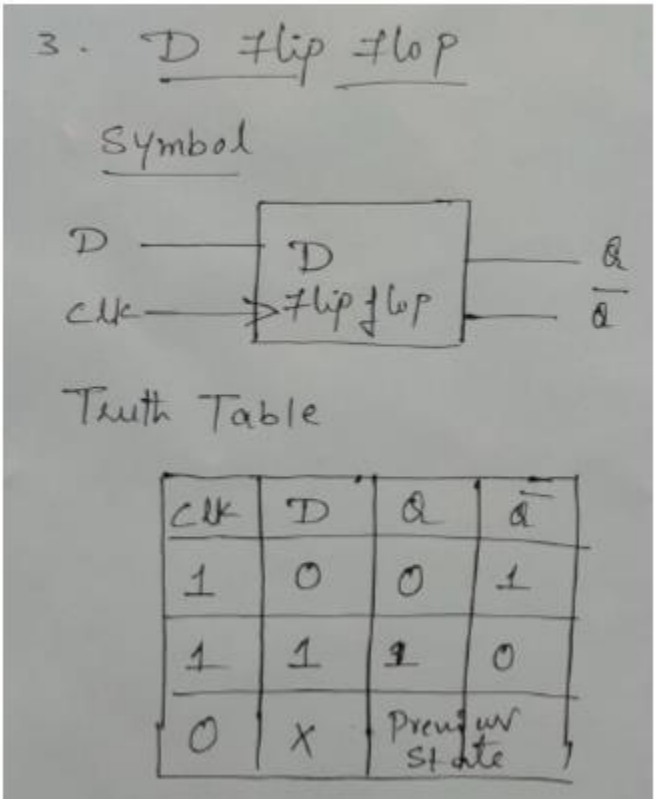
```
reg q,qbar;
always@(posedge clk)
begin
if(rst)
q<=1'b0;
else if (j==1'b0 && k==1'b0) q<=q;
else if (j==1'b0 && k==1'b1) q<=1'b0;
else if (j==1'b1 && k==1'b0) q<=1'b1;
else if (j==1'b1 && k==1'b1) q<= ~q;
assign qbar = ~q;
end
endmodule
```

OUTPUT: -

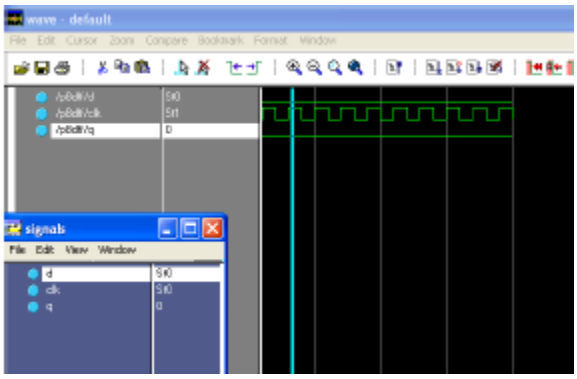




D FLIP-FLOP




```
module p8dff(d,clk,q);
  input d;
  input clk;
  output q;

reg q;
always @(posedge clk)
begin
q<=d;
end
endmodule
```


wave - default
signals