Experiment No: 1 (Software) **Date: 21-08-2022**

Verification of Logic Gates using Verilog

Register Number: 21BEC1851

Name: Rahul Karthik S

Aim: To Verify the truth table of the logic gates in Verilog using Gate, dataflow and Behavioural Modelling.

Software required: ModelSim

Procedure:

1. Click Jumpstart, create new project and type the project name and click ok.
2. Click Create New File, type the name and click Verilog from the drop-down and click ok.
3. After typing the code, click compile, compile selected from menu.
4. Click Simulate and click start simulation.
5. For input, select the input using Force or Clock and run it.

Verilog Code:

Gate Modelling

```
module gates (a,b,c,d,e,f,g,h,i);

input a,b;

output c,d,e,f,g,h,i;

and o1(c,a,b);

or o2(d,a,b);

not o3(e,a);

xor o4(f,a,b);

xnor o5(g,a,b);

nand o6(h,a,b);

nor o7(i,a,b);
```

```
endmodule
```

Data flow Modelling

```
module gates (a,b,c,d,e,f,g,h,i);
input a,b;
output c,d,e,f,g,h,I;
assign c = a&b;
assign d = a|b;
assign e = ~a;
assign f = a^b;
assign g = a~^b;
assign h = ~(a&b);
assign I = ~(a|b);
endmodule
```

Behavioral Modelling

AND Gate:

```
module gates(a,b,c);
input a,b;
output reg c;
always @ (a or b) begin
  if (a==1'b1 & b==1'b1) begin
    c=1'b1;
  end
  else
    c=1'b0;
end
```

```
endmodule
```

OR Gate:

```
module gates(a,b,c);
input a,b;
output reg c;
always @ (a or b) begin
  if (a==1'b1 | b==1'b1) begin
    c=1'b1;
  end
  else
    c=1'b0;
end
endmodule
```

NOT Gate:

```
module gates(a,c);
input a;
output reg c;
always @ (a) begin
  if (a == 1'b1) begin
    c=1'b0;
  end
  else
    c=1'b1;
end
endmodule
```

XOR Gate:

```
module gates(a,b,c);
input a,b;
output reg c;
always @ (a or b) begin
  if (a==b) begin
    c=1'b0;
  end
  else
    c=1'b1;
end
endmodule
```

XNOR Gate:

```
module gates(a,b,c);
input a,b;
output reg c;
always @ (a or b) begin
  if (a==b) begin
    c=1'b1;
  end
  else
    c=1'b1;
end
endmodule
```

NAND Gate:

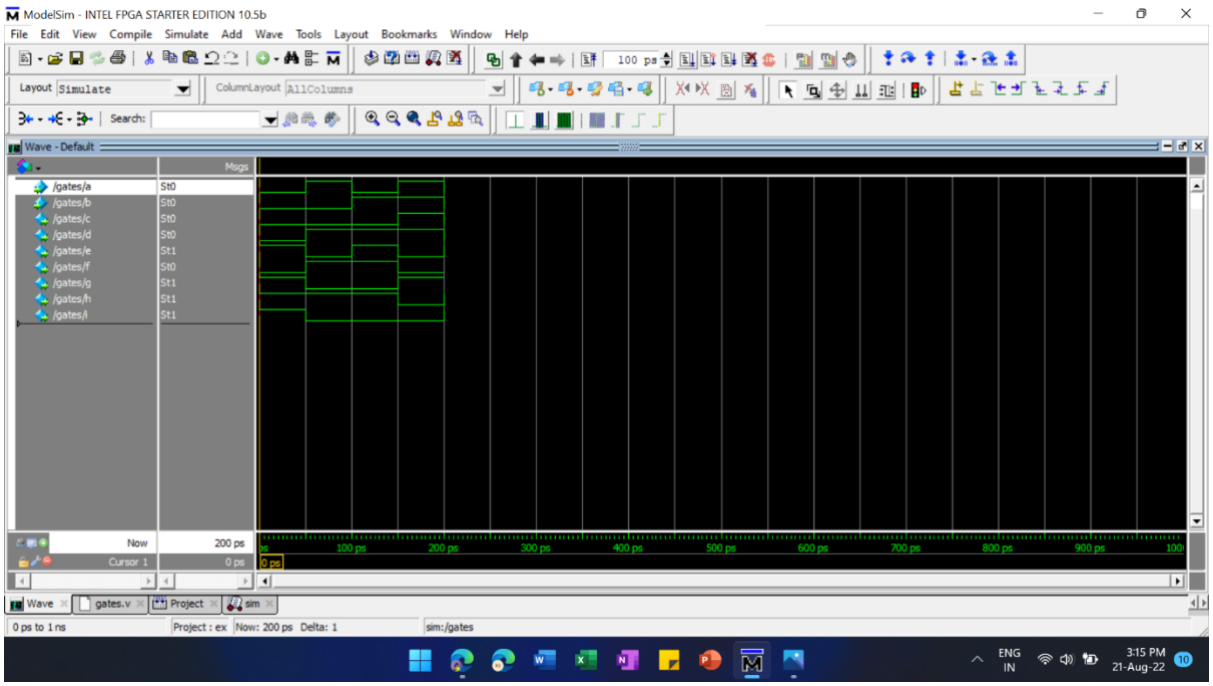
```
module gates(a,b,c);
input a,b;
output reg c;
always @ (a or b) begin
  if (a==1'b1 & b==1'b1) begin
    c=1'b0;
  end
  else
    c=1'b1;
end
endmodule
```

NOR Gate:

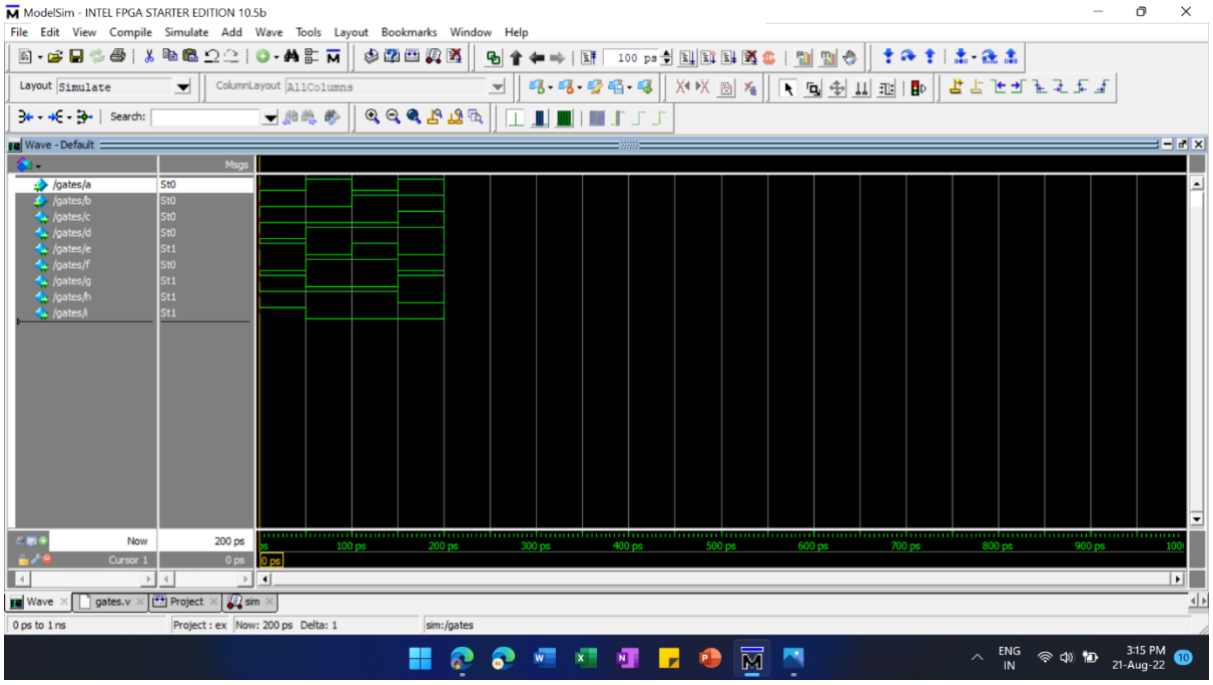
```
module gates(a,b,c);
input a,b;
output reg c;
always @ (a or b) begin
  if (a==1'b1 & b==1'b1) begin
    c=1'b0;
  end
  else
    c=1'b1;
end
endmodule
```

Verilog Output:

Gate Modelling

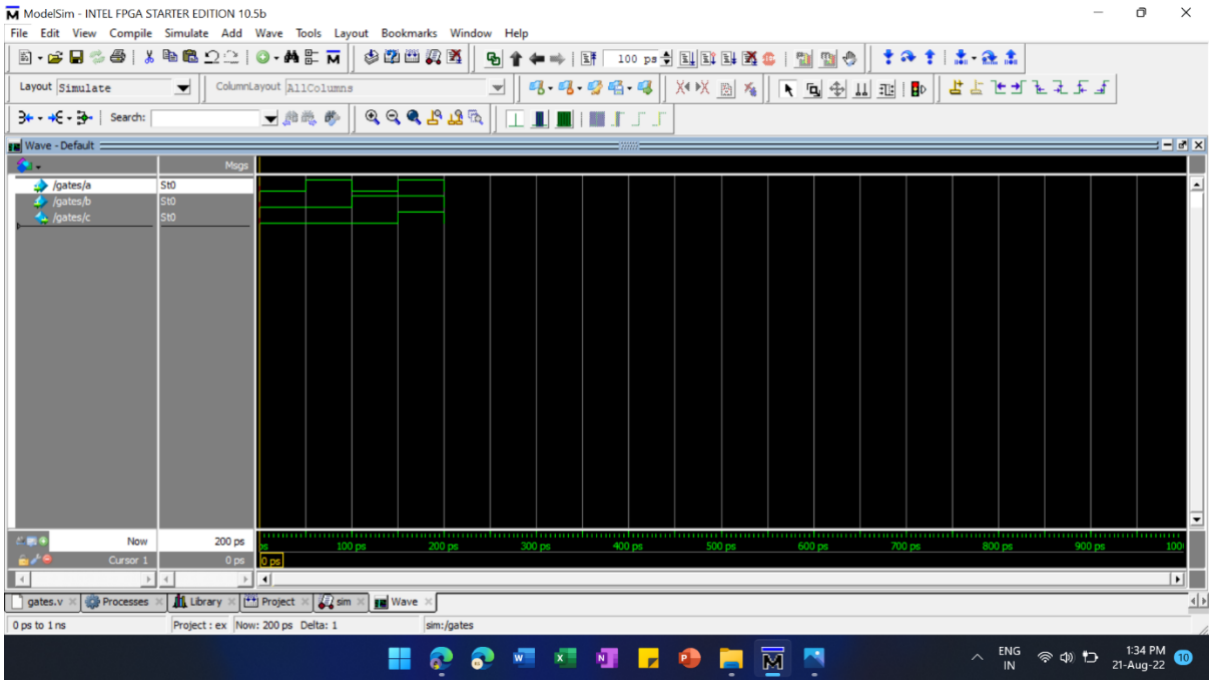


Data flow Modelling

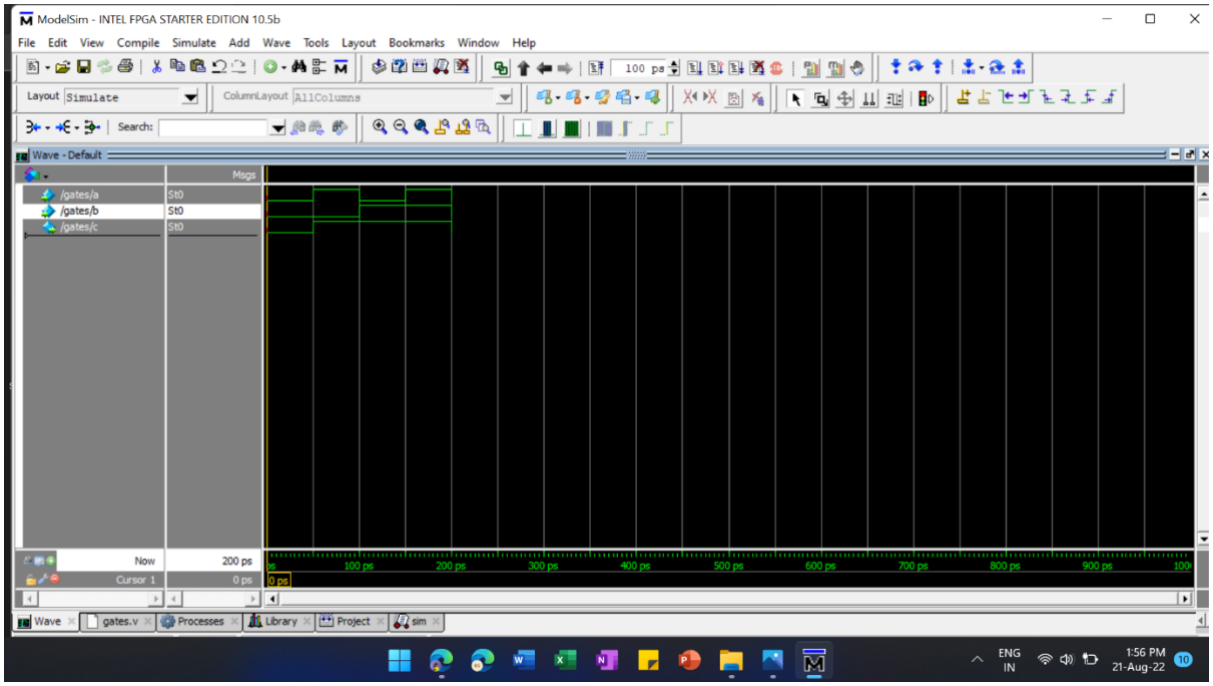


Behavioral Modelling

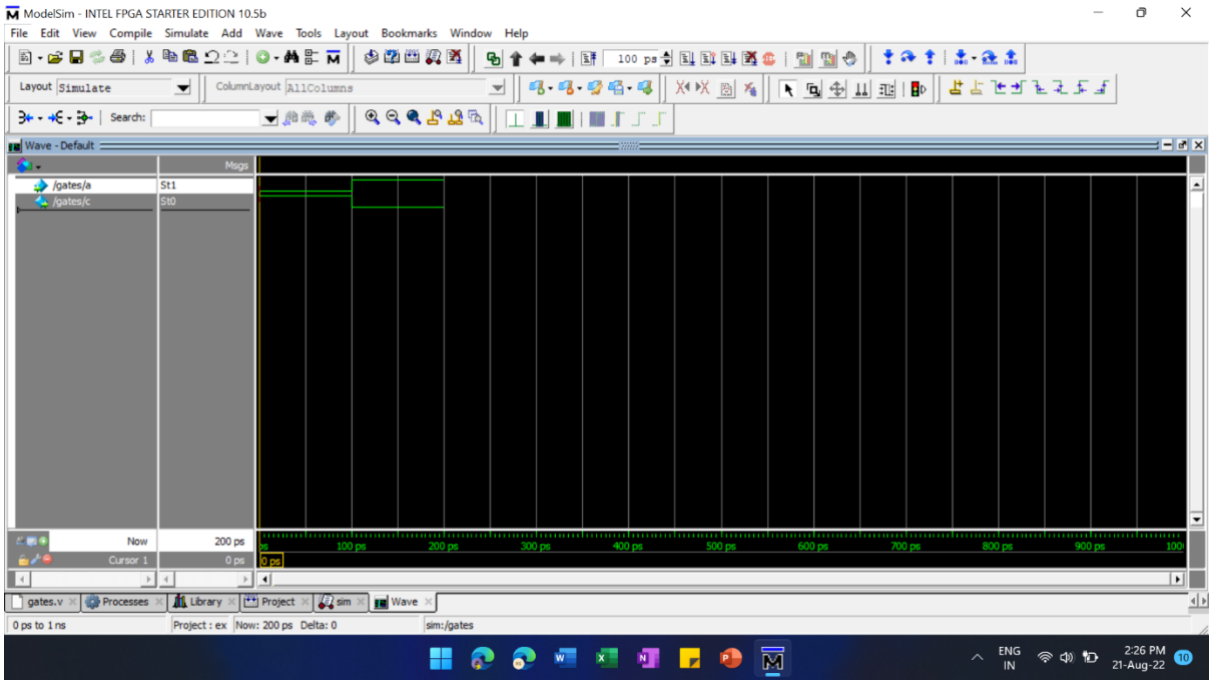
AND Gate:



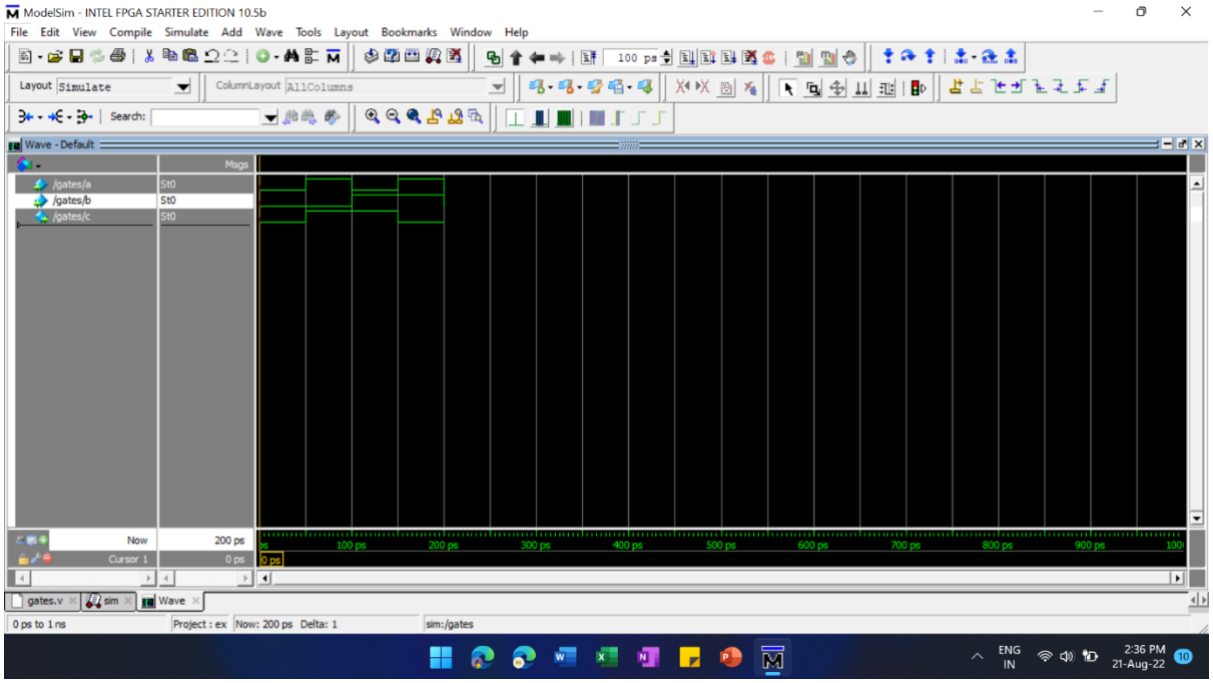
OR Gate:



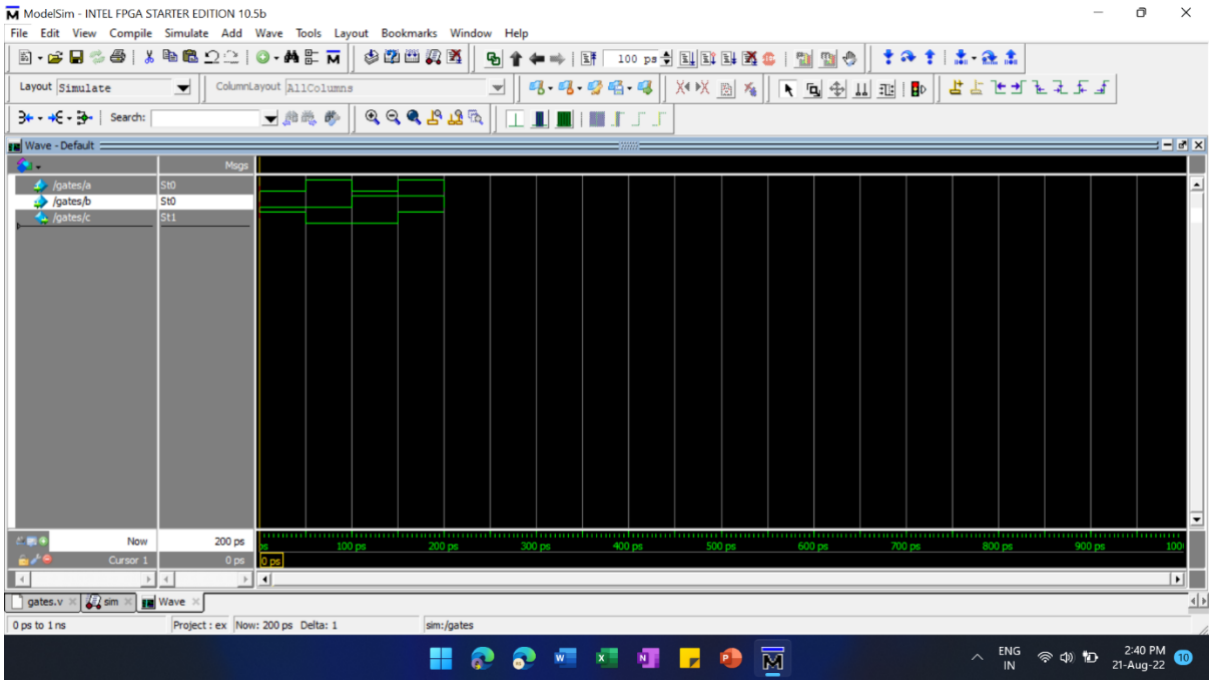
NOT Gate:



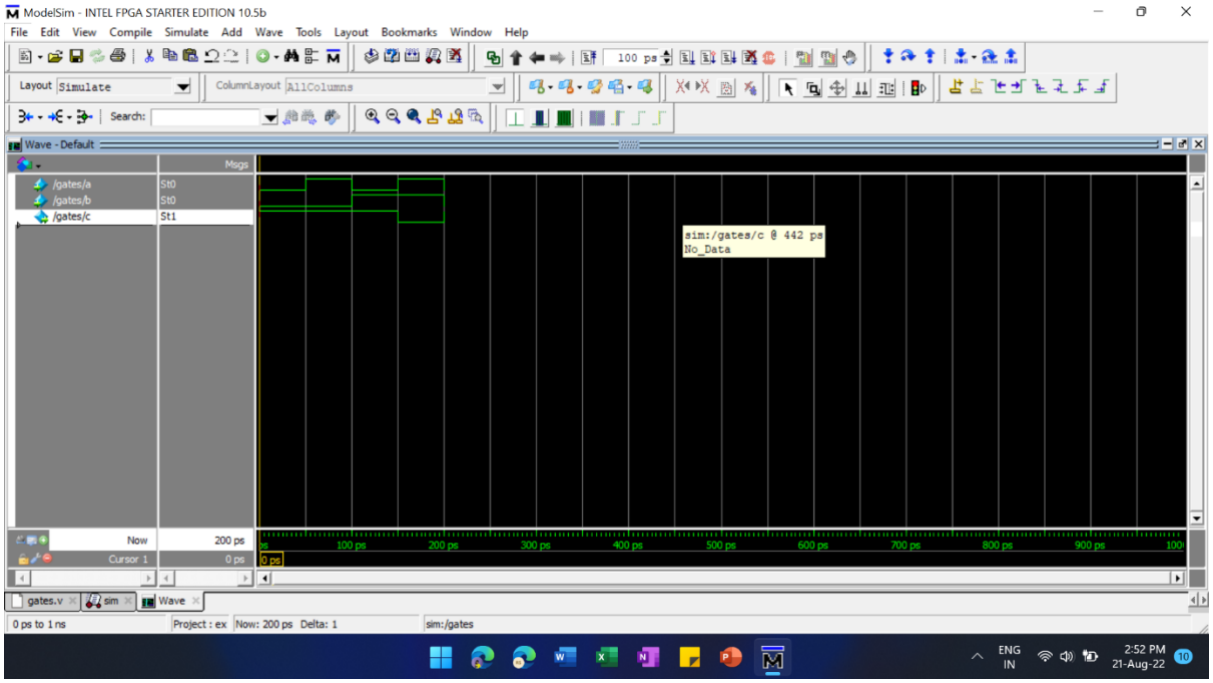
XOR Gate:



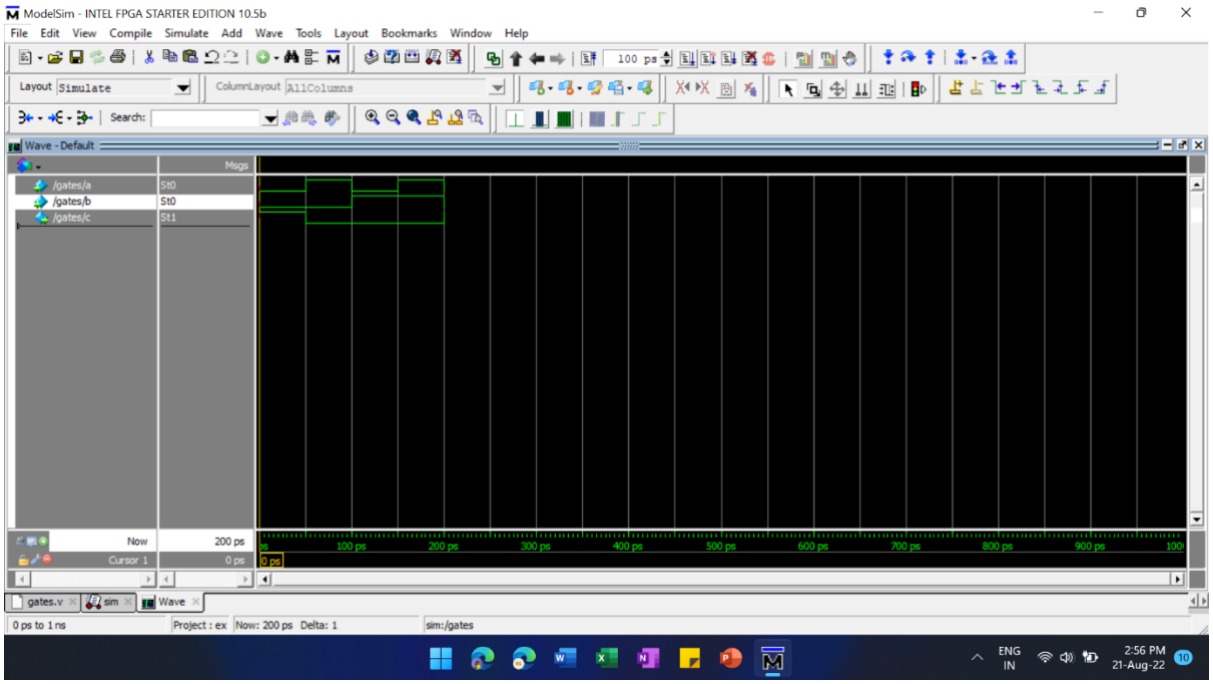
XNOR Gate:



NAND Gate:



NOR Gate:



Result/Inference:

From this experiment, the truth table of the logic gates in Verilog using Gate, dataflow and Behavioural Modelling is verified.

Experiment No. 2 (Software) **Date: 28-08-2022**

DESIGN OF ADDER AND SUBTRACTOR USING LOGIC GATES

Register Number: 21BEC1851

Name: Rahul Karthik S

Aim:

To design and construct half adder, full adder, half subtractor, full subtractor circuits using logic gates in Verilog and verify its truth table.

Software Required:

ModelSim

Theory:

Half Adder:

A half adder has two inputs for the two bits to be added and two outputs one from the sum 'S' and other from the carry 'c' into the higher adder position. Above circuit is called as a carry signal from the addition of the less significant bits sum from the X-OR Gate the carry out from the AND gate.

Full Adder:

A full adder is a combinational circuit that forms the arithmetic sum of input; it consists of three inputs and two outputs. A full adder is useful to add three bits at a time but a half adder cannot do so. In full adder sum output will be taken from X-OR Gate, carry output will be taken from OR Gate.

Half Subtractor:

The half subtractor is constructed using X-OR and AND Gate. The half subtractor has two input and two outputs. The outputs are difference and borrow. The difference can be applied using X-OR Gate, borrow output can be implemented using an AND Gate and an inverter.

Full Subtractor:

The full subtractor is a combination of X-OR, AND, OR, NOT Gates. In a full subtractor the logic circuit should have three inputs and two outputs. The two half subtractor put together gives a full subtractor. The first half subtractor will be C and AB. The output will be difference output of full subtractor. The expression AB assembles the borrow output of the half subtractor and the second term is the inverted difference output of first X-OR.

Procedure:

1. Click Jumpstart, create new project and type the project name and click ok.

2. Click Create New File, type the name and click Verilog from the drop-down and click ok.
3. After typing the code, click compile, compile selected from menu.
4. Click Simulate and click start simulation.
5. Select the variables and Click add wave.
6. For input, select the input using Force or Clock and run it.

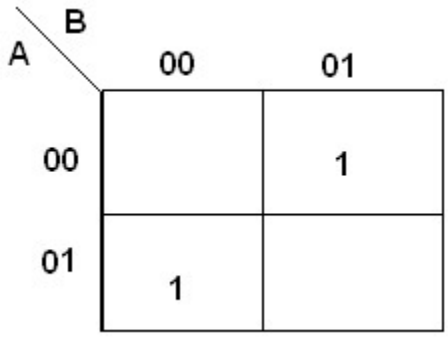
Design and Logic Diagram:

Half Adder:

Truth Table:

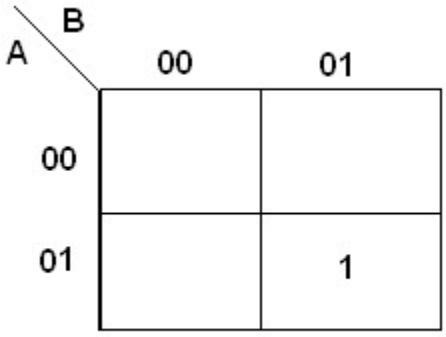
A	**B**	**CARRY**	**SUM**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

K- Map for SUM:



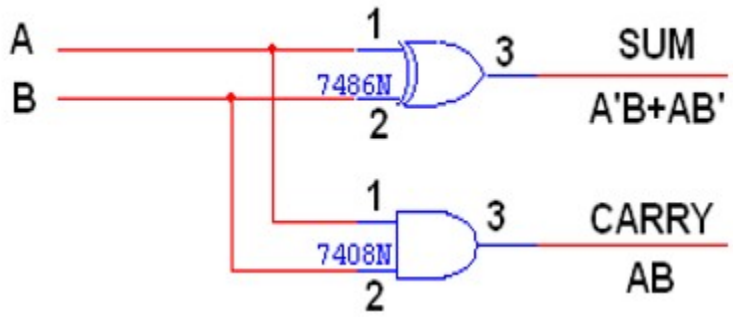
SUM = A'B + AB' = A $\oplus$ B

K- Map for CARRY:



CARRY = AB

Truth Table:

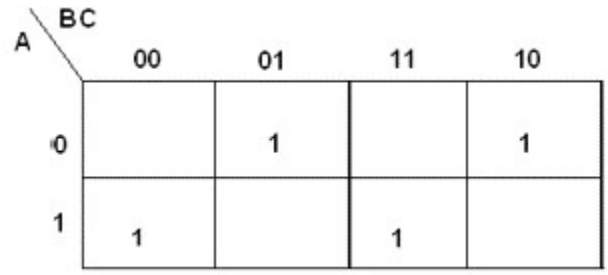


Full Adder:

Truth Table:

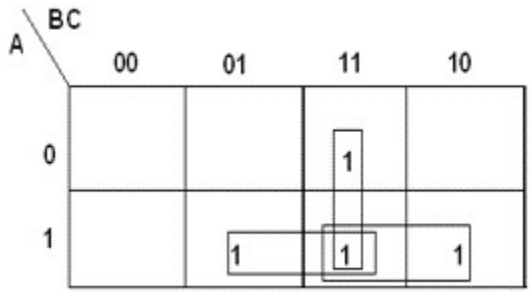
A	B	C	CARRY	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

K- Map for SUM:



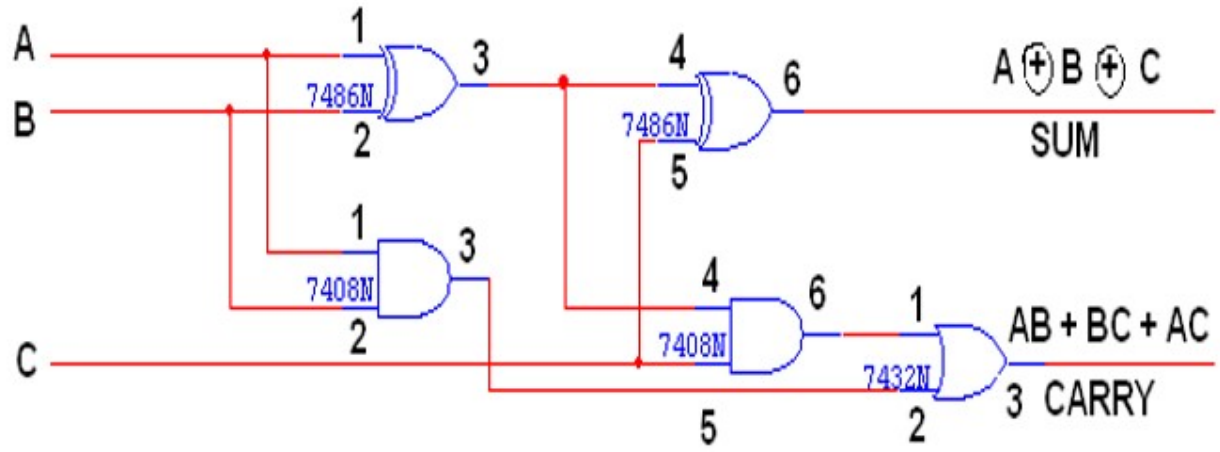
SUM = A’B’C + A’BC’ + ABC’ + ABC = A ⊕ B ⊕ C

K- Map for CARRY:



CARRY = AB + BC + AC

Truth Table:

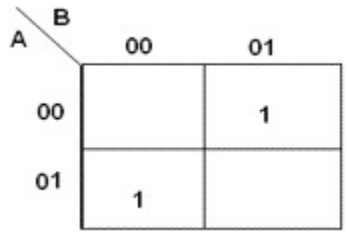


Half Subtractor:

Truth Table:

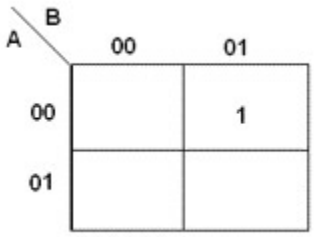
A	B	BORROW	DIFFERENCE
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

K- Map for DIFFERENCE:



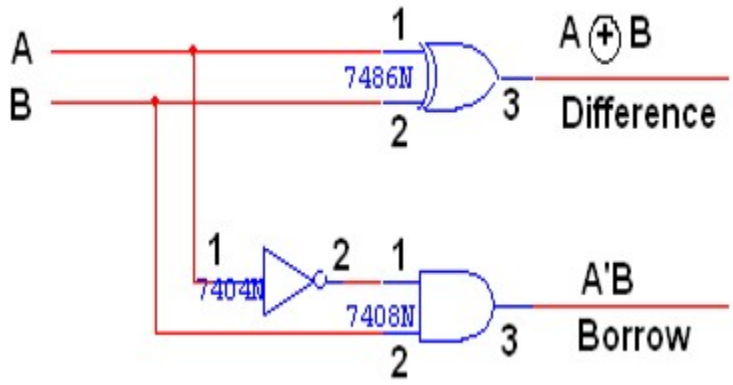
DIFFERENCE = A’B + AB’ = A $\oplus$ B

K- Map for BORROW:



BORROW = A’B

Truth Table:

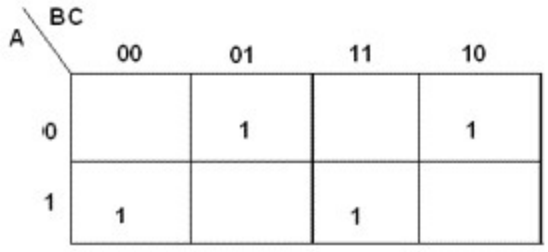


Full Subtractor:

Truth Table:

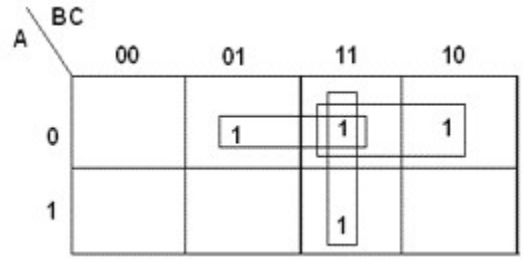
A	B	C	BORROW	DIFFERENCE
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

K- Map for DIFFERENCE:



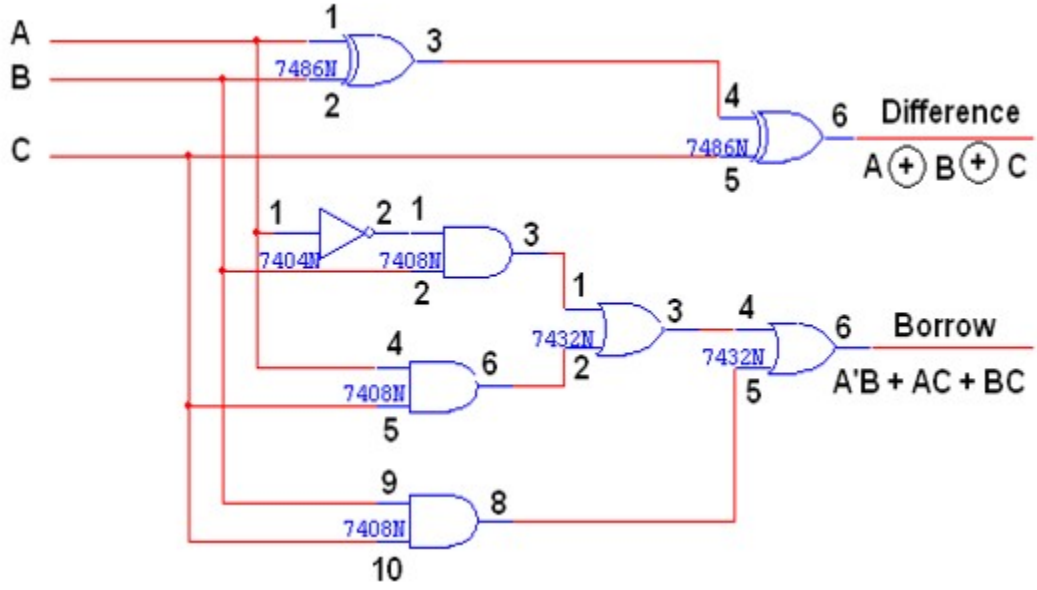
DIFFERENCE = A'B'C + A'BC' + AB'C' + ABC

K- Map for BORROW:



BORROW = A'B + BC + A'C

Truth Table:

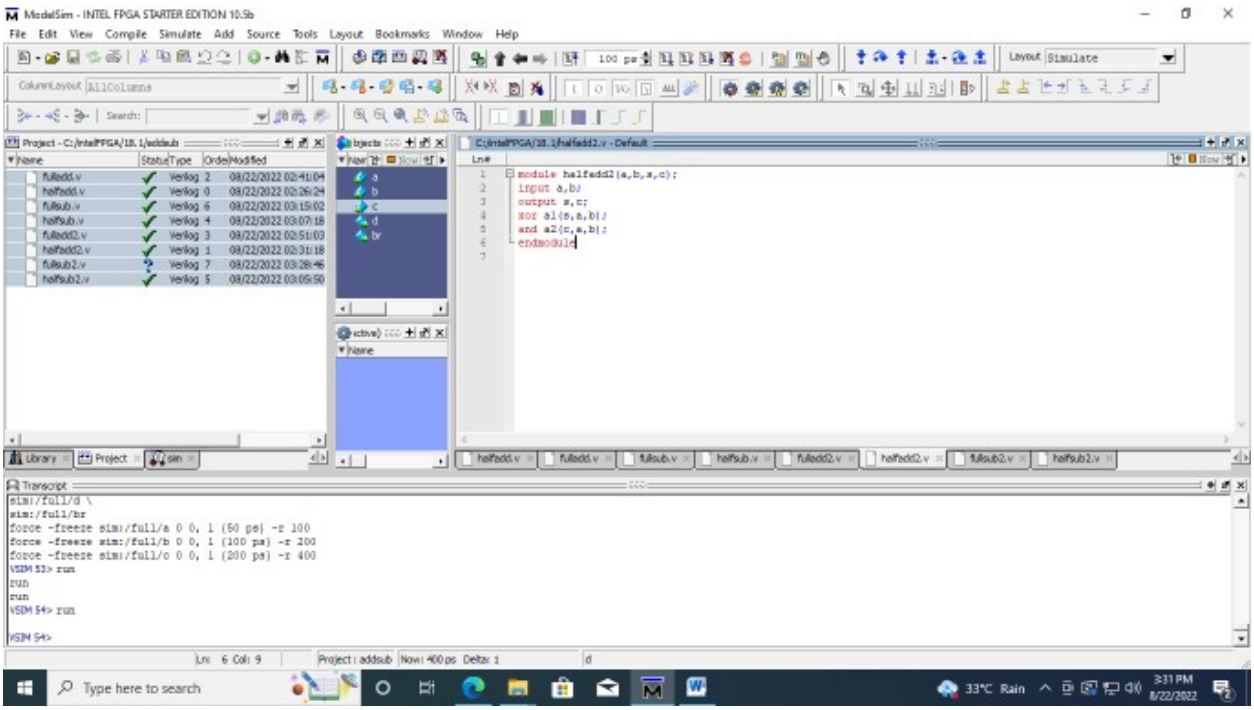


Verilog Code:

Gate Modelling:

Half Adder:

```
module halfadd2(a,b,s,c);

input a,b;

output s,c;

xor a1(s,a,b);

and a2(c,a,b);

endmodule
```

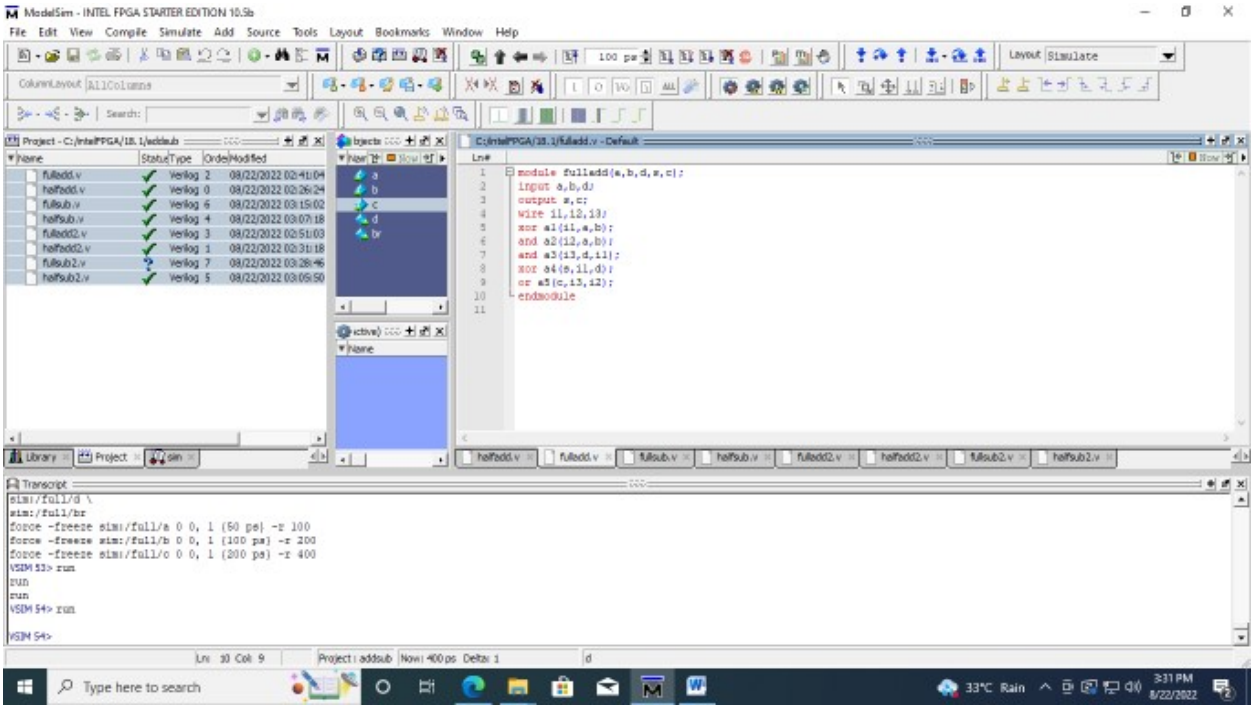


Full Adder:

```
module fulladd(a,b,d,s,c);

input a,b,d;

output s,c;

wire i1,i2,i3;

xor a1(i1,a,b);

and a2(i2,a,b);

and a3(i3,d,i1);

xor a4(s,i1,d);
```

or a5(c,i3,i2);

endmodule



Half Subtractor:

module halfsub(a,b,d,br);

input a,b;

wire i1;

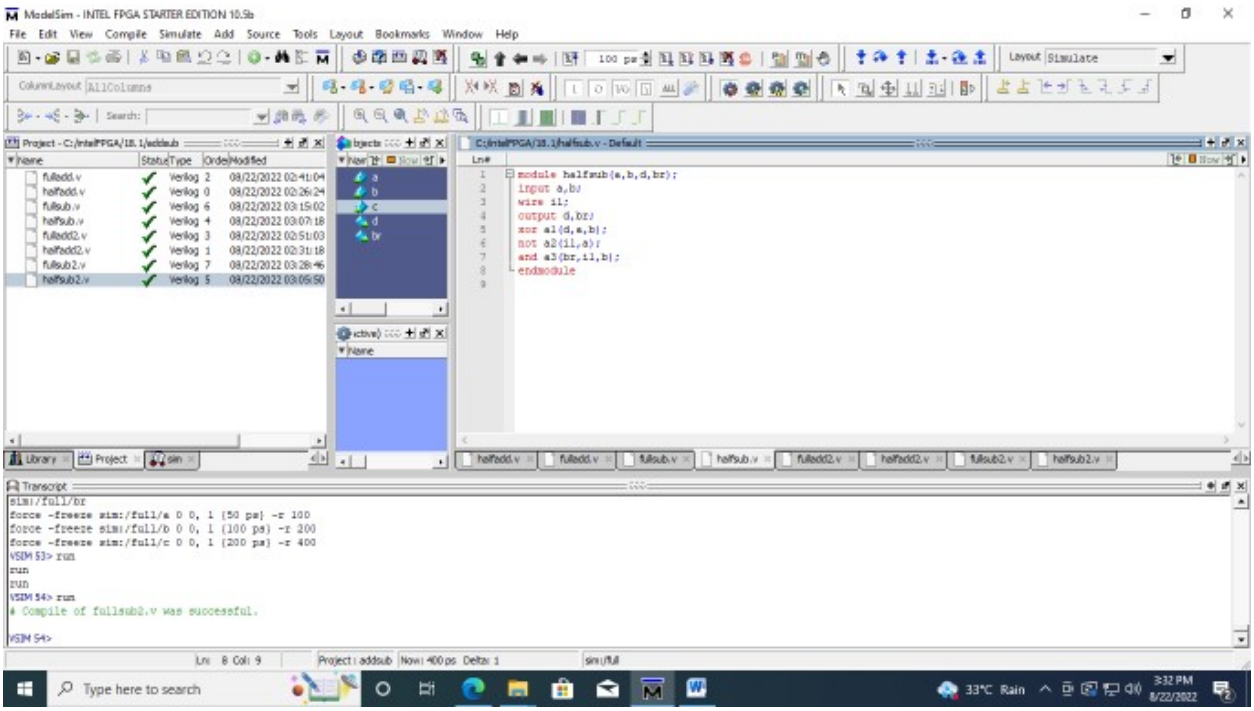
output d,br;

xor a1(d,a,b);

not a2(i1,a);

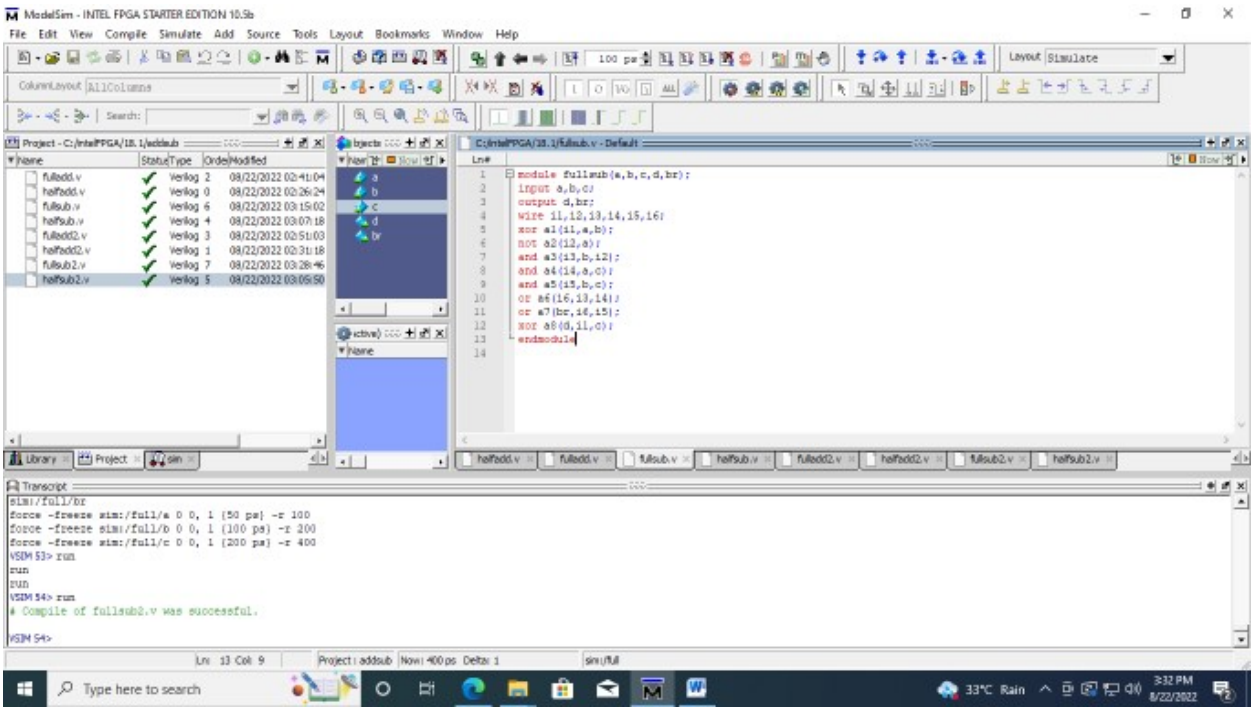
and a3(br,i1,b);

endmodule



Full Subtractor:

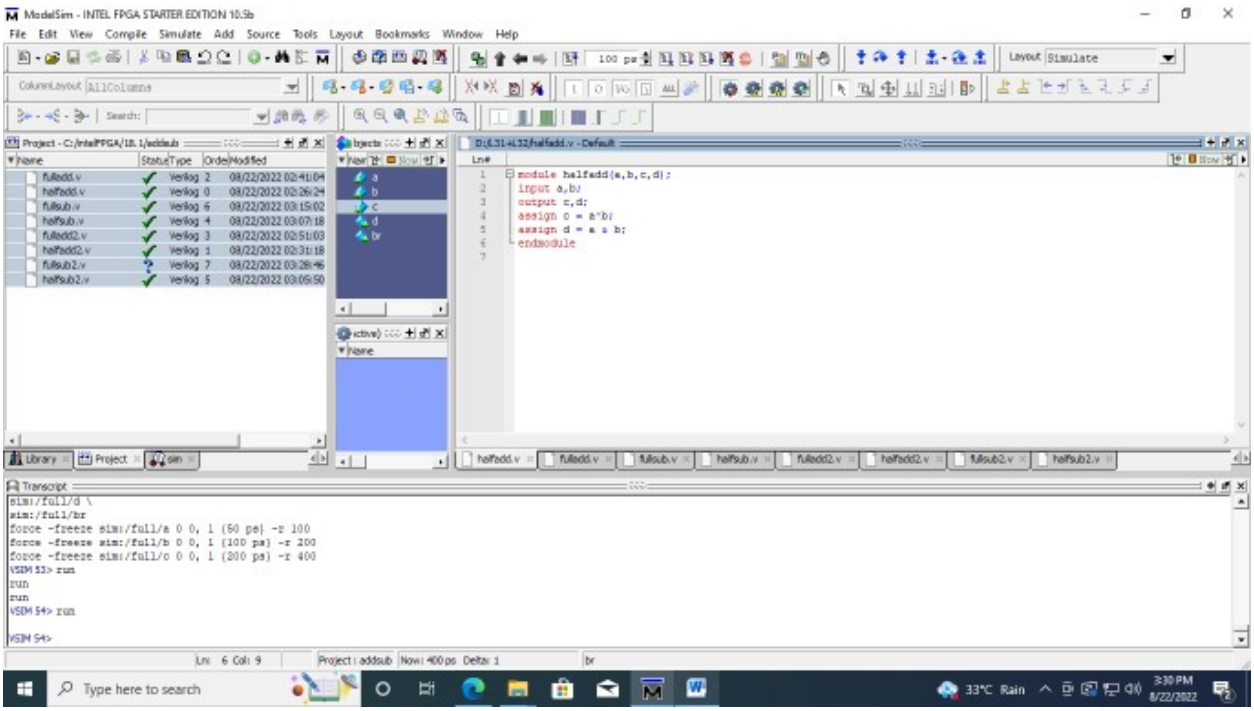
```
module fullsub(a,b,c,d,br);

input a,b,c;

output d,br;

wire i1,i2,i3,i4,i5,i6;

xor a1(i1,a,b);

not a2(i2,a);

and a3(i3,b,i2);

and a4(i4,a,c);

and a5(i5,b,c);

or a6(i6,i3,i4);

or a7(br,i6,i5);

xor a8(d,i1,c);

endmodule
```



Data Flow Modelling:

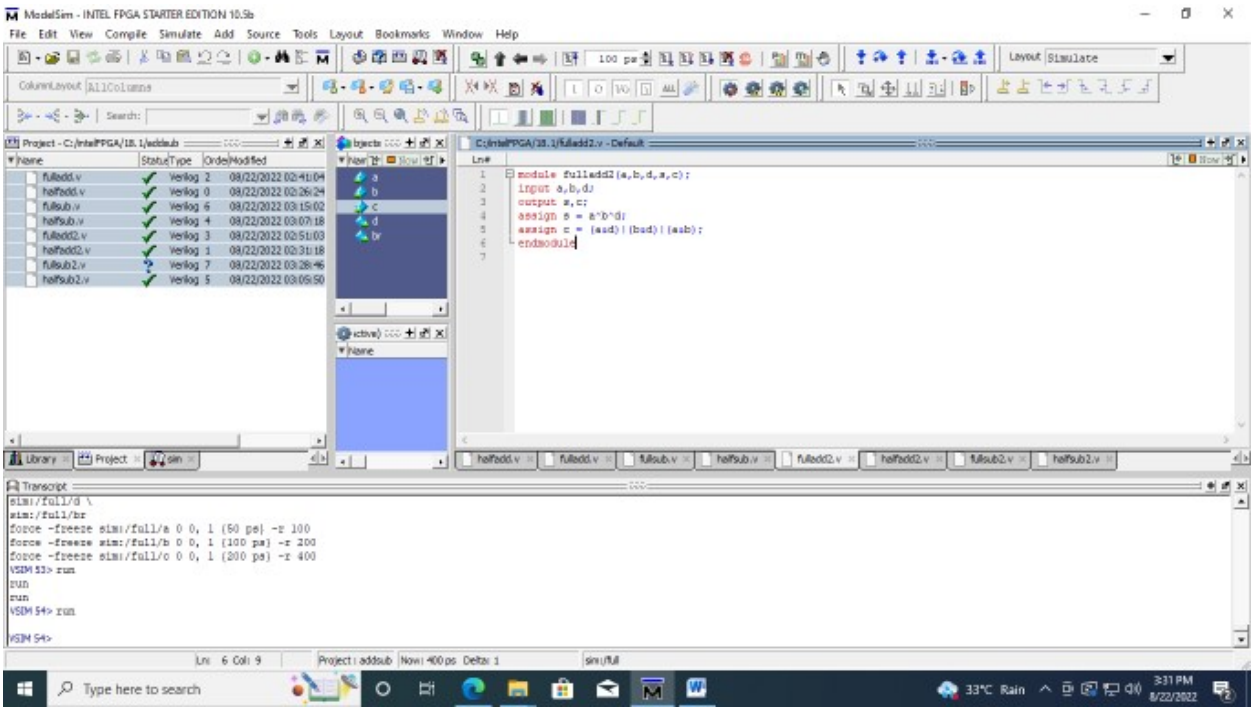
Half Adder:

```
module halfadd(a,b,c,d);

input a,b;

output c,d;

assign c = a^b;

assign d = a & b;

endmodule
```



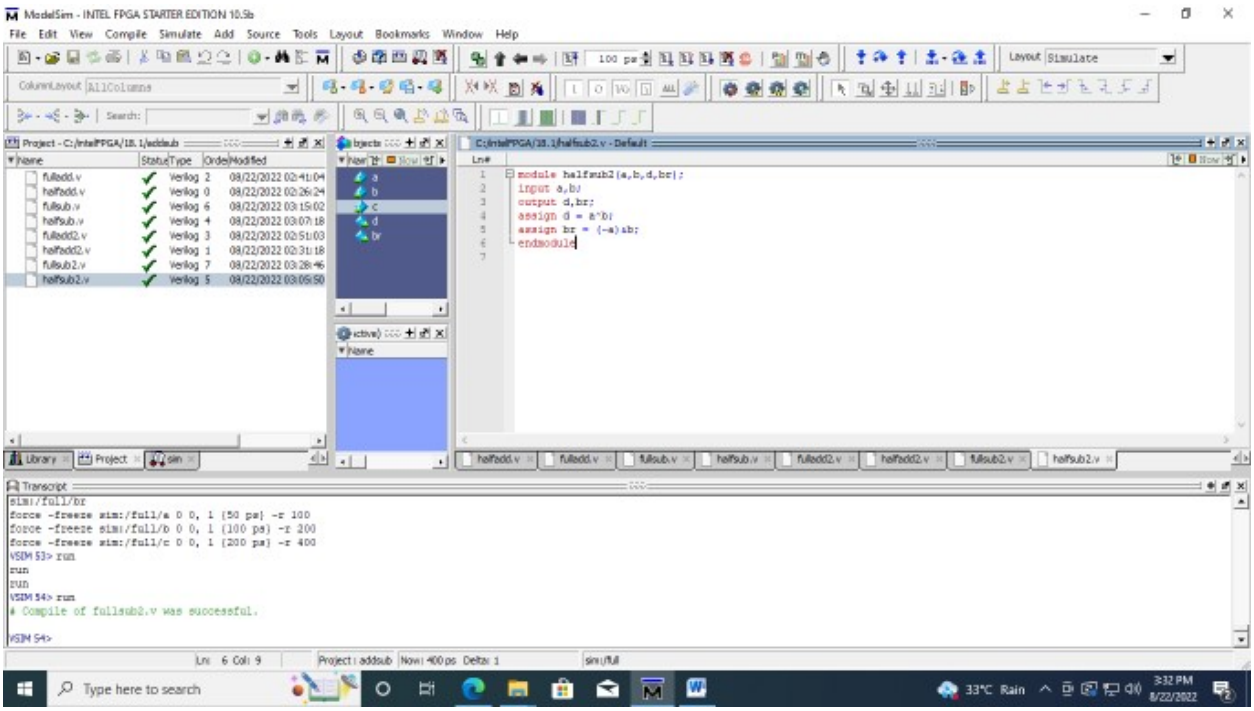
Full Adder:

```
module fulladd2(a,b,d,s,c);

input a,b,d;

output s,c;

assign s = a^b^d;

assign c = (a&d)|(b&d)|(a&b);

endmodule
```



Half Subtractor:

```
module halfsub2(a,b,d,br);

input a,b;

output d,br;

assign d = a^b;

assign br = (~a)&b;

endmodule
```



Full Subtractor:

module fullsub2(a,b,c,d,br);

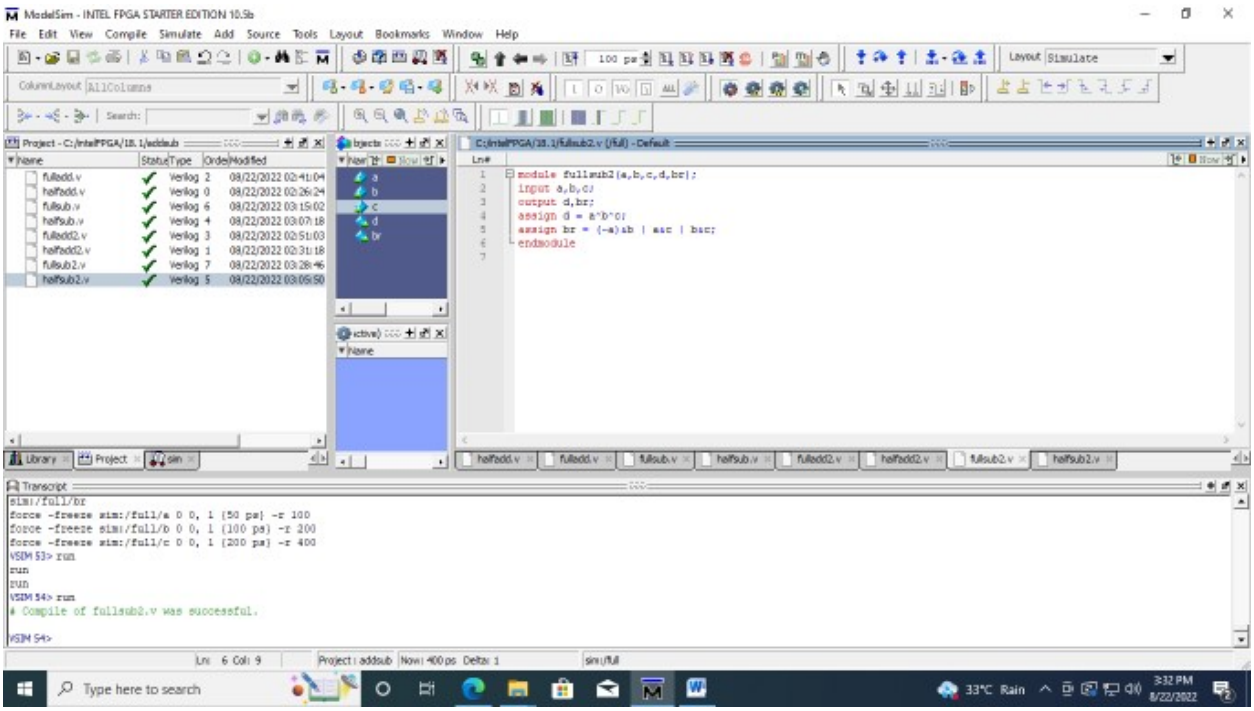
input a,b,c;

output d,br;

assign d = a^b^c;

assign br = (~a)&b | a&c | b&c;

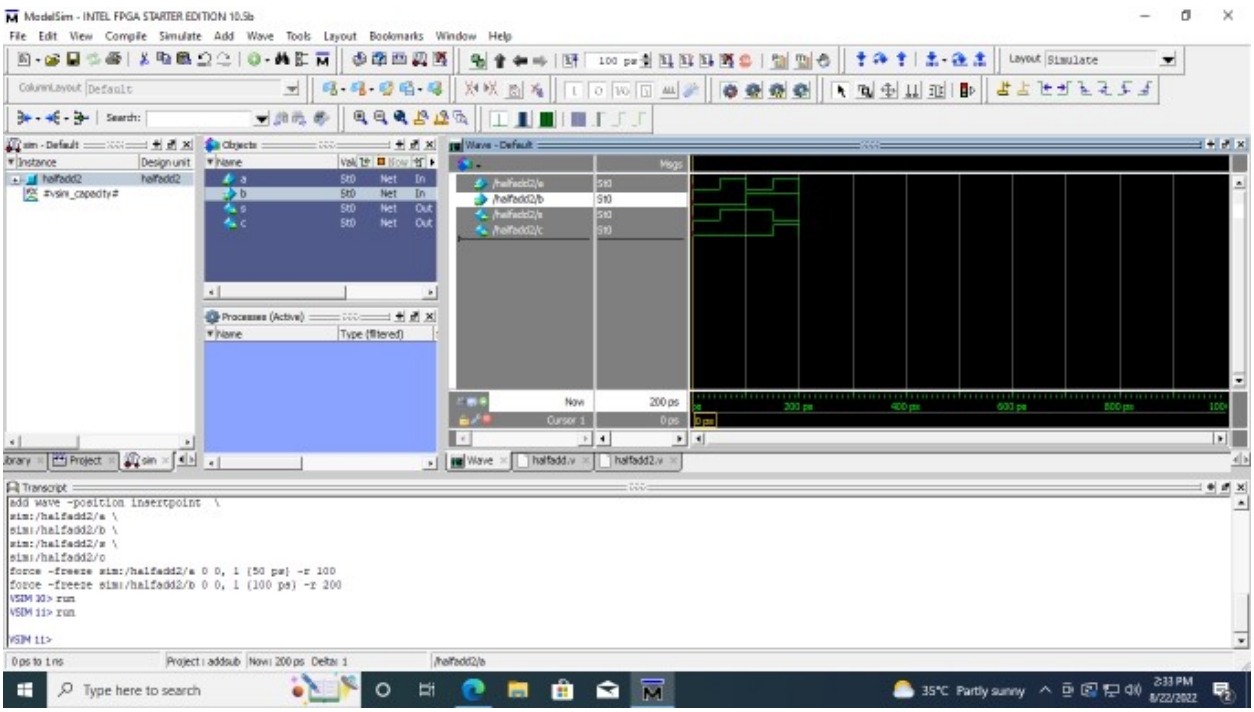
endmodule



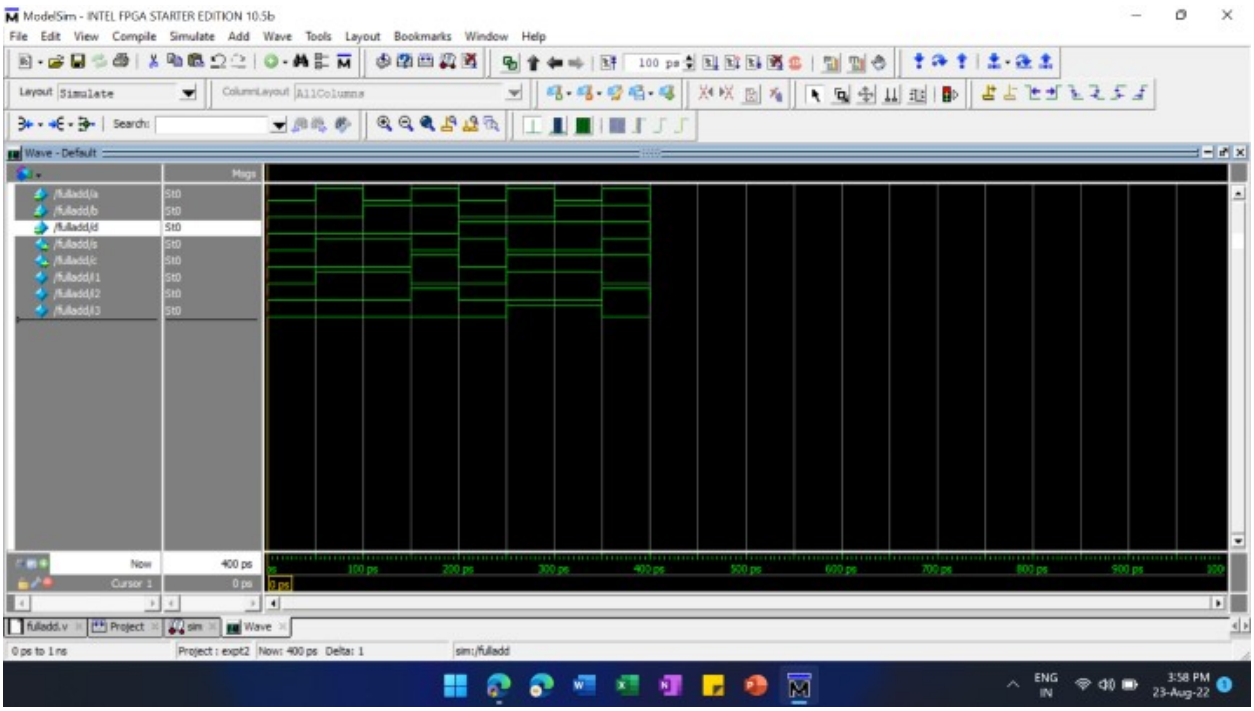
Verilog Output:

<u>Gate Modelling:</u>

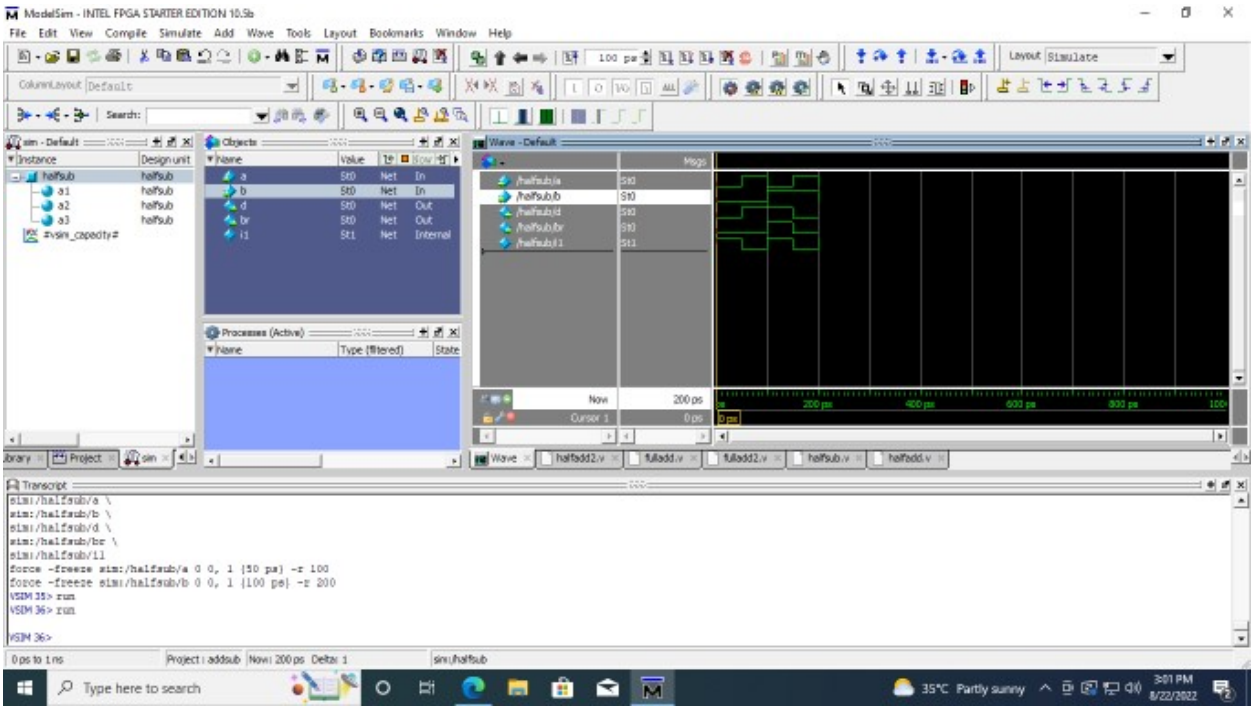
Half Adder:



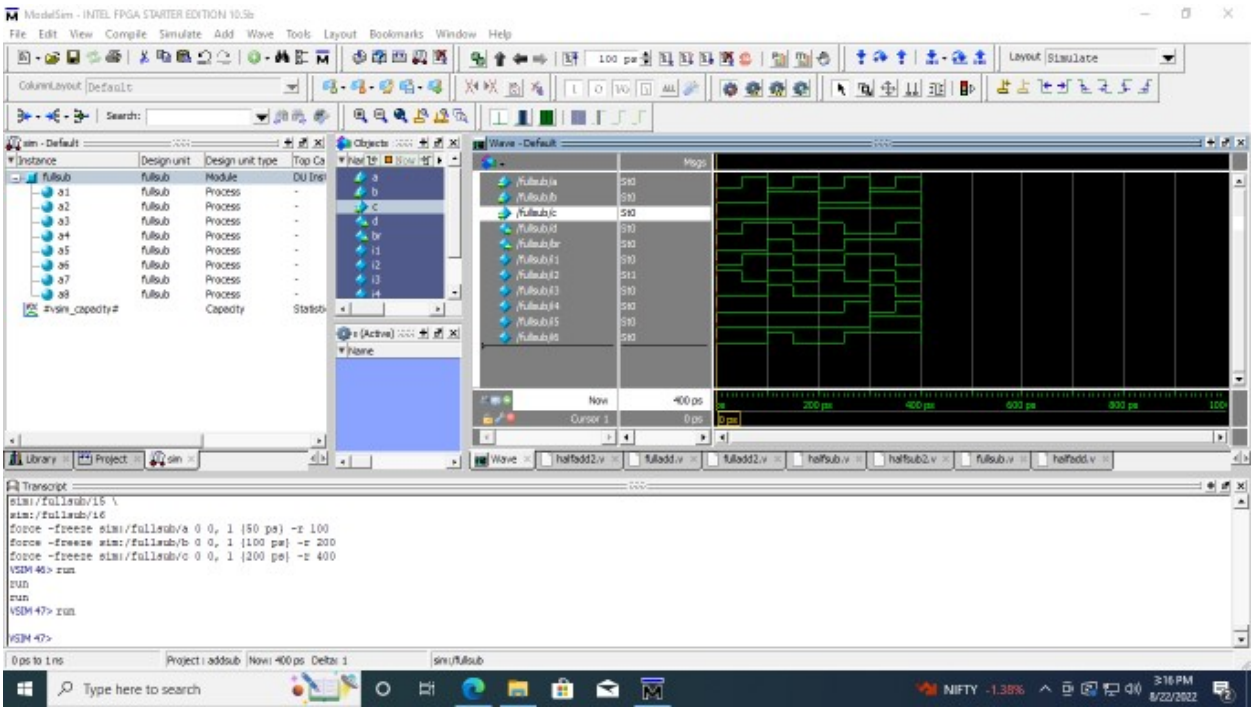
Full Adder:



Half Subtractor:

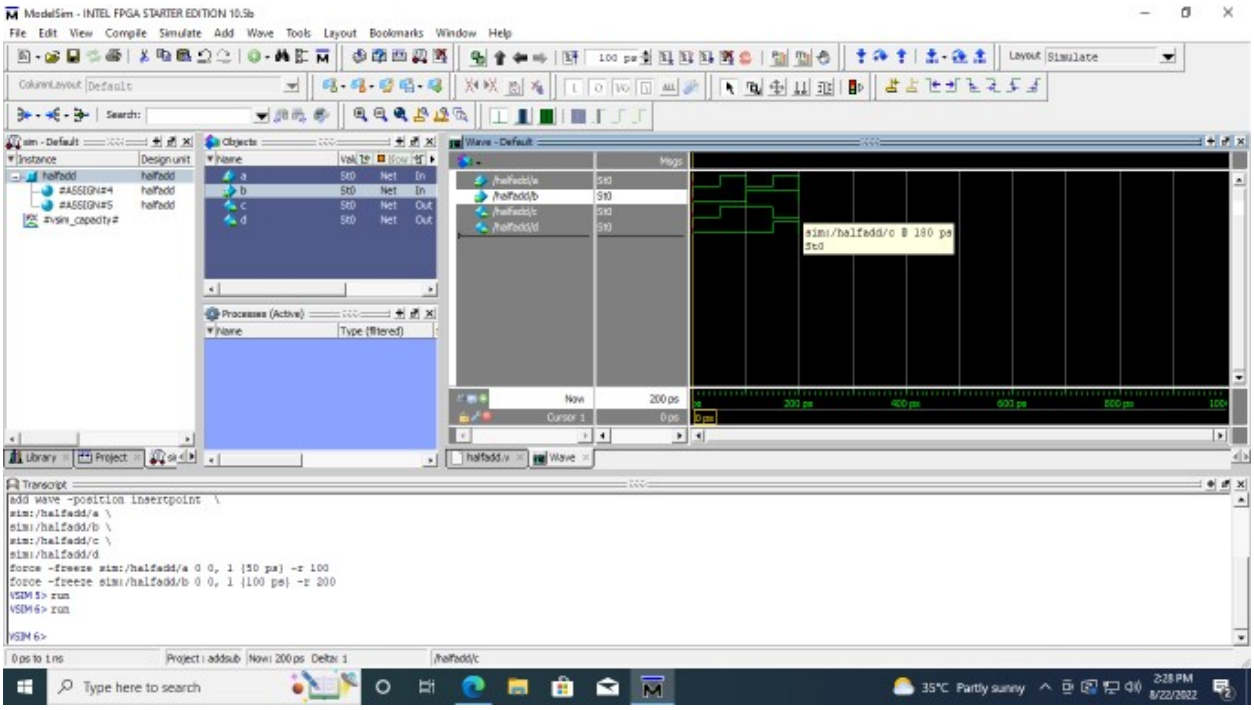


Full Subtractor:

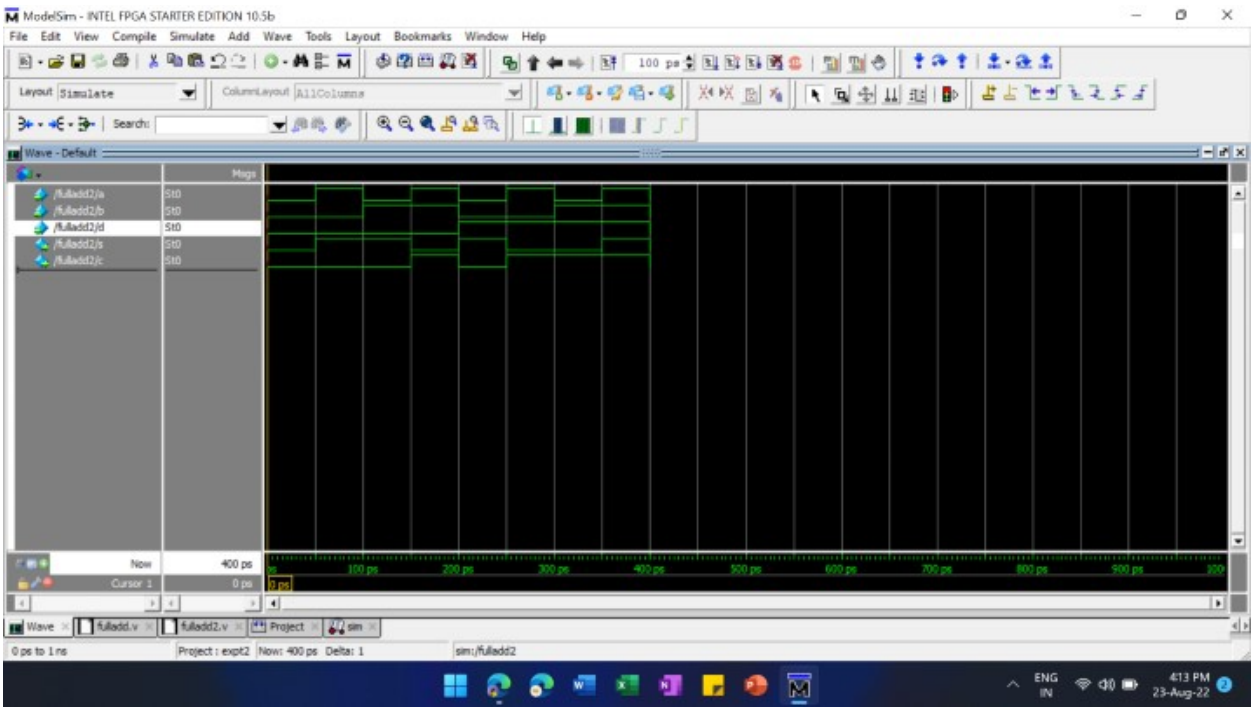


Data Flow Modelling:

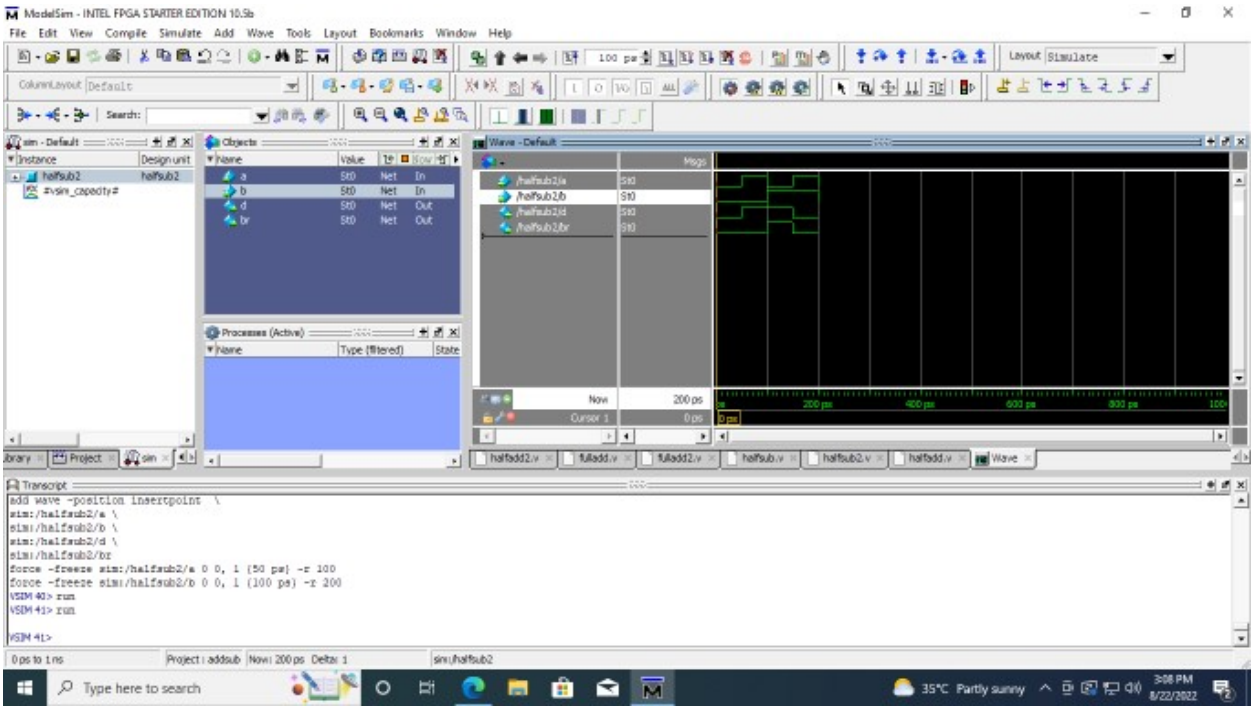
Half Adder:



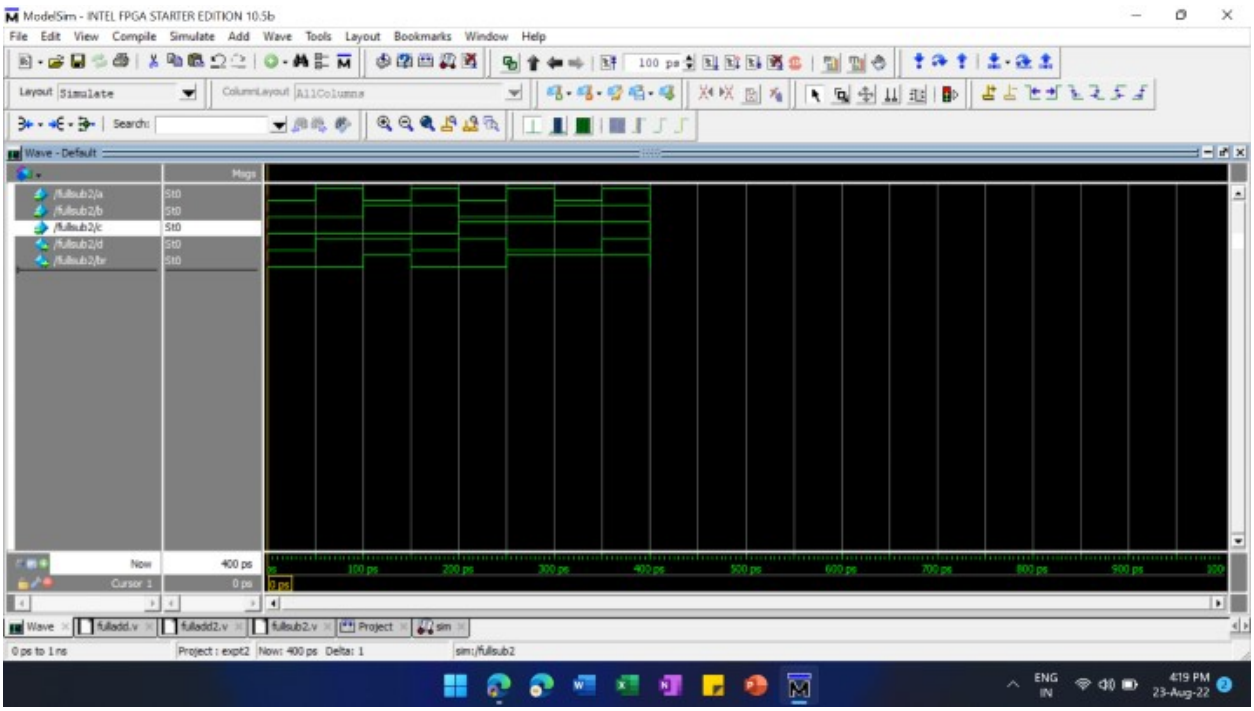
Full Adder:



Half Subtractor:



Full Subtractor:



Result:

The adder and subtractor circuit were designed and constructed using logic gates in verilog and their truth table was verified.

Experiment No: 4 (Software) **Date: 10-09-2022**

DESIGN OF ENCODER AND DECODER USING LOGIC GATES

Register Number: 21BEC1851

Name: Rahul Karthik S

Aim:

To design and construct a decoder and encoder circuit with logic gates using Verilog.

Software Required:

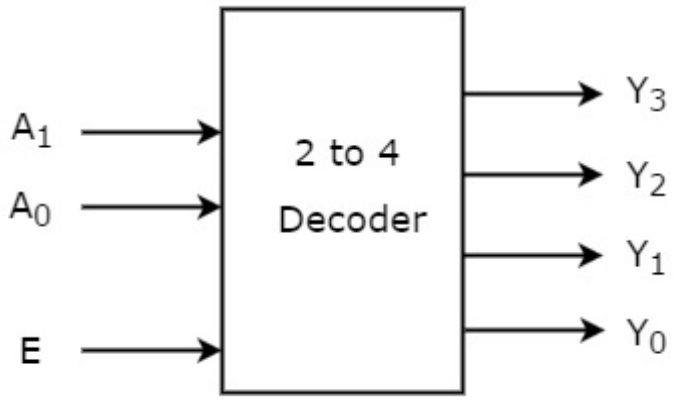
ModelSim

Theory:

Decoder:

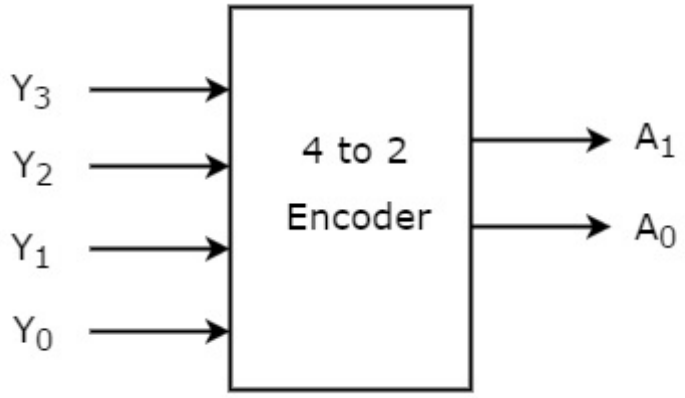
The name "Decoder" means to translate or decode coded information from one format into another, so a digital decoder transforms a set of digital input signals into an equivalent decimal code at its output.

A decoder is a combinational circuit that converts binary information from n input lines to a maximum of m=2n unique output lines.



Encoder:

An Encoder is a combinational circuit that performs the reverse operation of Decoder. It has maximum of 2^n input lines and 'n' output lines. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2n input lines with 'n' bits. It is optional to represent the enable signal in encoders.



Procedure:

1. Open ModelSim.
2. Click Jumpstart and create new project.
3. The create new file and select Verilog from the drop-down.
4. Type the code, compile it.
5. Follow (3) and do the test bench, compile it.
6. Then, simulate the test bench file and run it.
7. After that follow the same procedure for encoders.

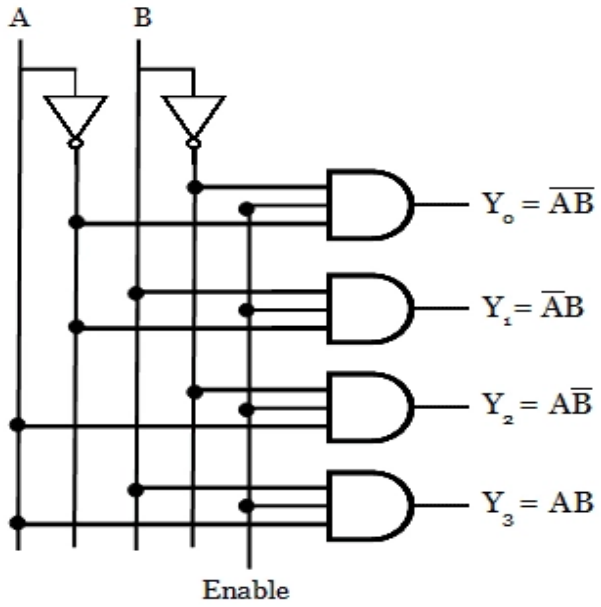
Observation:

2 to 4 Decoder:

Truth Table:

Enable	**Inputs**		**Output**			
E	**A**	**B**	**Y_3**	**Y_2**	**Y_1**	**Y_0**
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Logic Diagram:

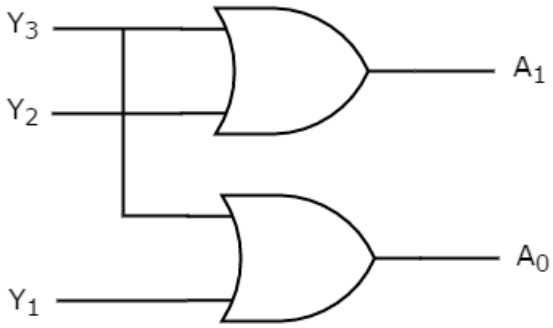


4 to 2 Encoder:

Truth Table:

Inputs				Output	
Y_3	**Y_2**	**Y_1**	**Y_0**	**A_1**	**A_0**
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Logic Diagram:



A1= Y3+Y2

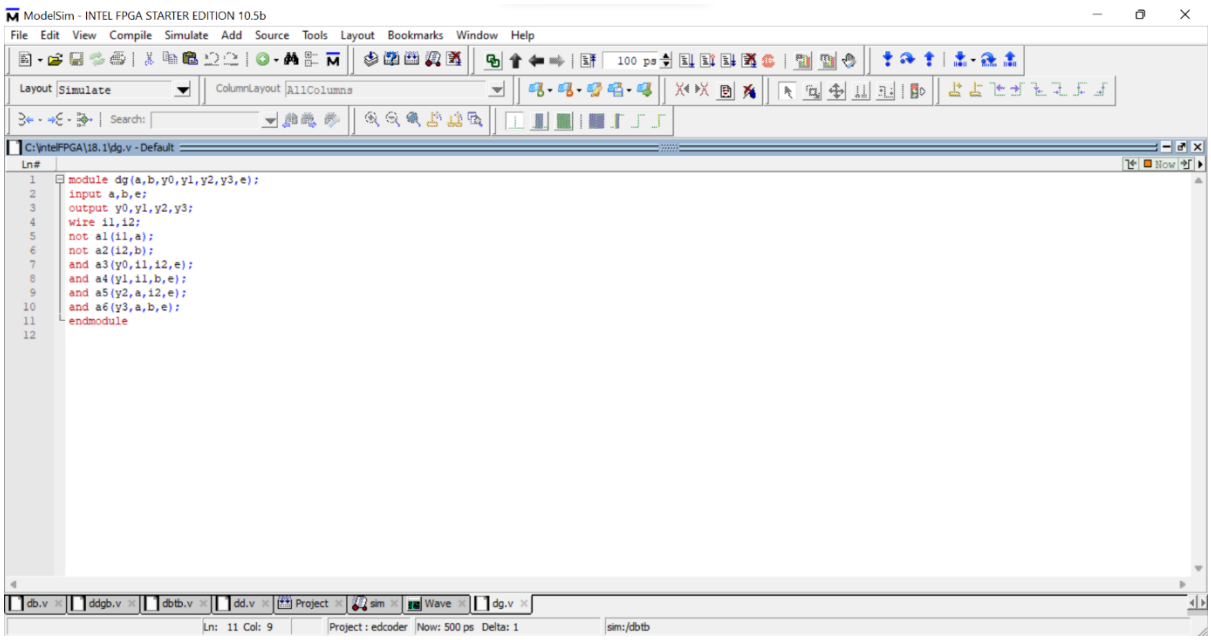
A0=Y1+Y3

Verilog Code:

2 to 4 Decoder:

Gate Modelling:

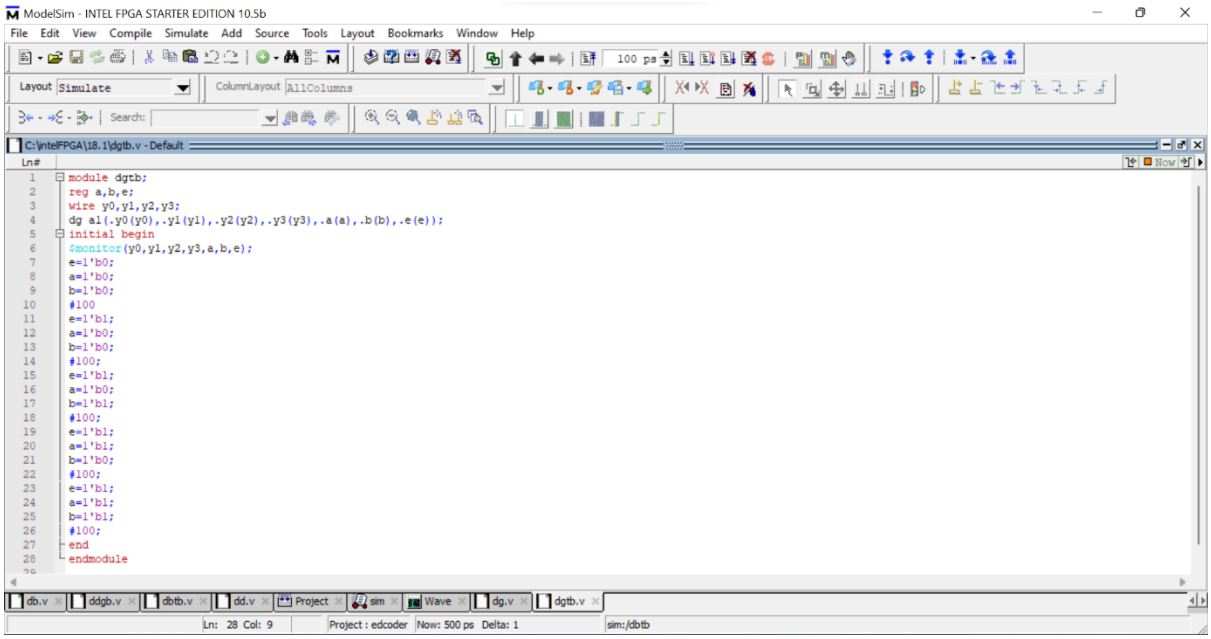
```
module dg(a,b,y0,y1,y2,y3,e);
input a,b,e;
output y0,y1,y2,y3;
wire i1,i2;
not a1(i1,a);
not a2(i2,b);
and a3(y0,i1,i2,e);
and a4(y1,i1,b,e);
and a5(y2,a,i2,e);
and a6(y3,a,b,e);
endmodule
```



Test Bench for Gate Modelling:

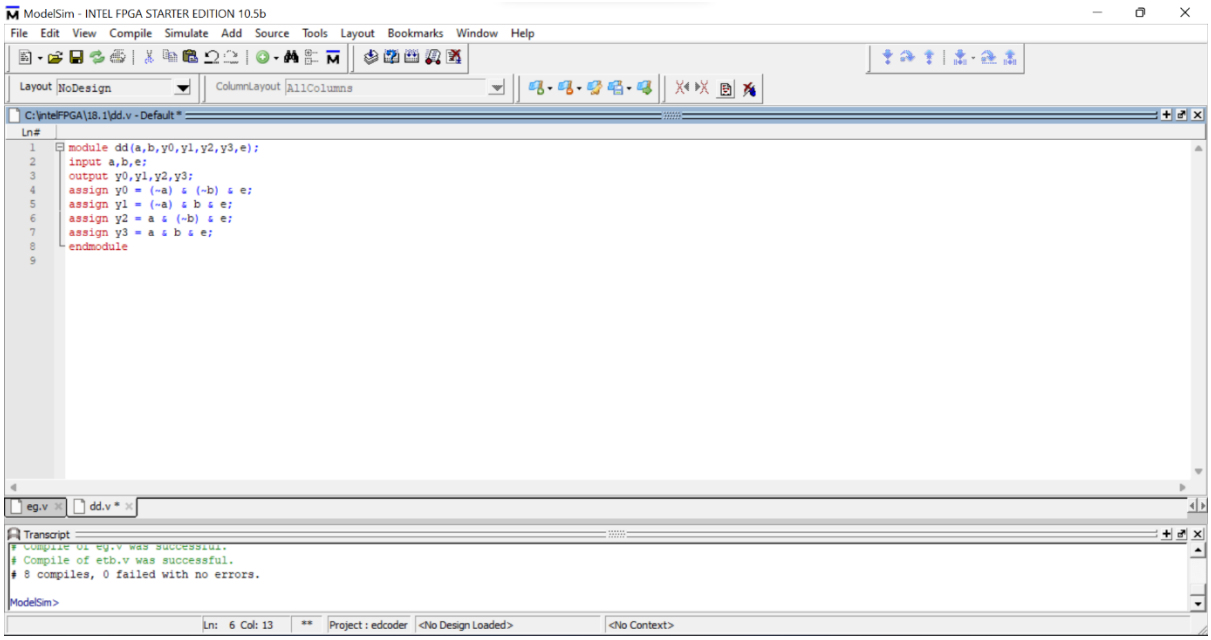
```
module dgtb;
reg a,b,e;
wire y0,y1,y2,y3;
dg a1(.y0(y0),.y1(y1),.y2(y2),.y3(y3),.a(a),.b(b),.e(e));
initial begin
$monitor(y0,y1,y2,y3,a,b,e);
```

```
e=1'b0;
a=1'b0;
b=1'b0;
#100
e=1'b1;
a=1'b0;
b=1'b0;
#100;
e=1'b1;
a=1'b0;
b=1'b1;
#100;
e=1'b1;
a=1'b1;
b=1'b0;
#100;
e=1'b1;
a=1'b1;
b=1'b1;
#100;
end
endmodule
```



Data-Flow Modelling:

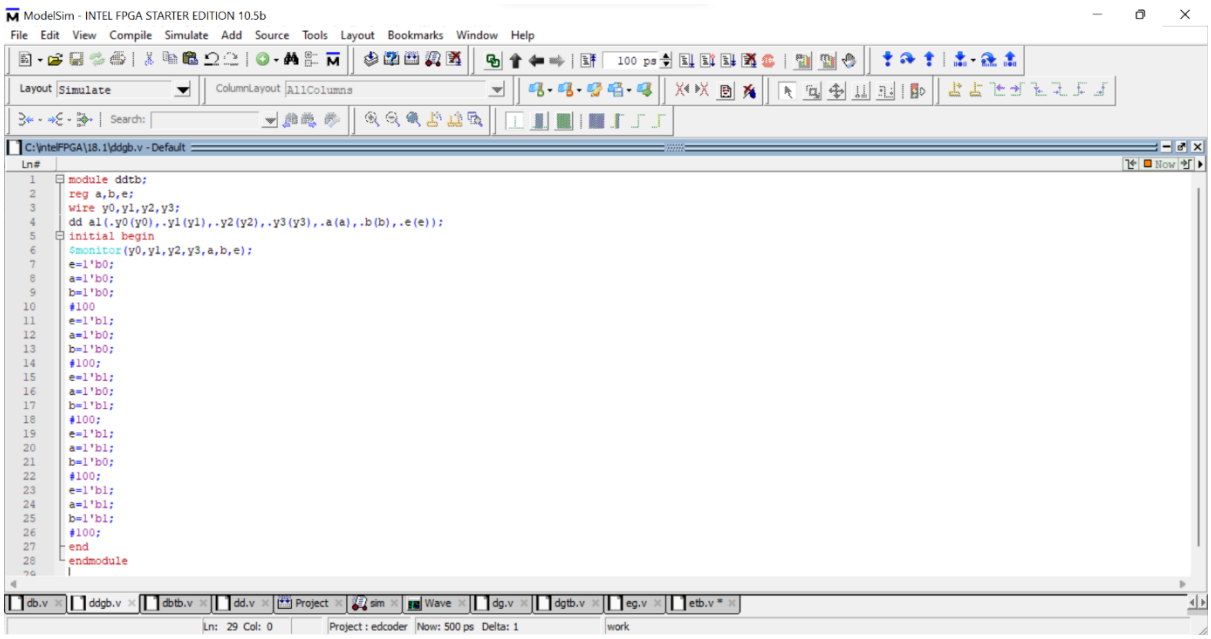
```
module dd(a,b,y0,y1,y2,y3,e);

input a,b,e;

output y0,y1,y2,y3;

assign y0 = (~a) & (~b) & e;

assign y1 = (~a) & b & e;

assign y2 = a & (~b) & e;

assign y3 = a & b & e;

endmodule
```



Test Bench for Data-Flow Modelling:

```
module ddtb;

reg a,b,e;

wire y0,y1,y2,y3;

dd a1(.y0(y0),.y1(y1),.y2(y2),.y3(y3),.a(a),.b(b),.e(e));

initial begin

$monitor(y0,y1,y2,y3,a,b,e);

e=1'b0;

a=1'b0;

b=1'b0;

#100

e=1'b1;

a=1'b0;

b=1'b0;

#100;

e=1'b1;

a=1'b0;

b=1'b1;
```

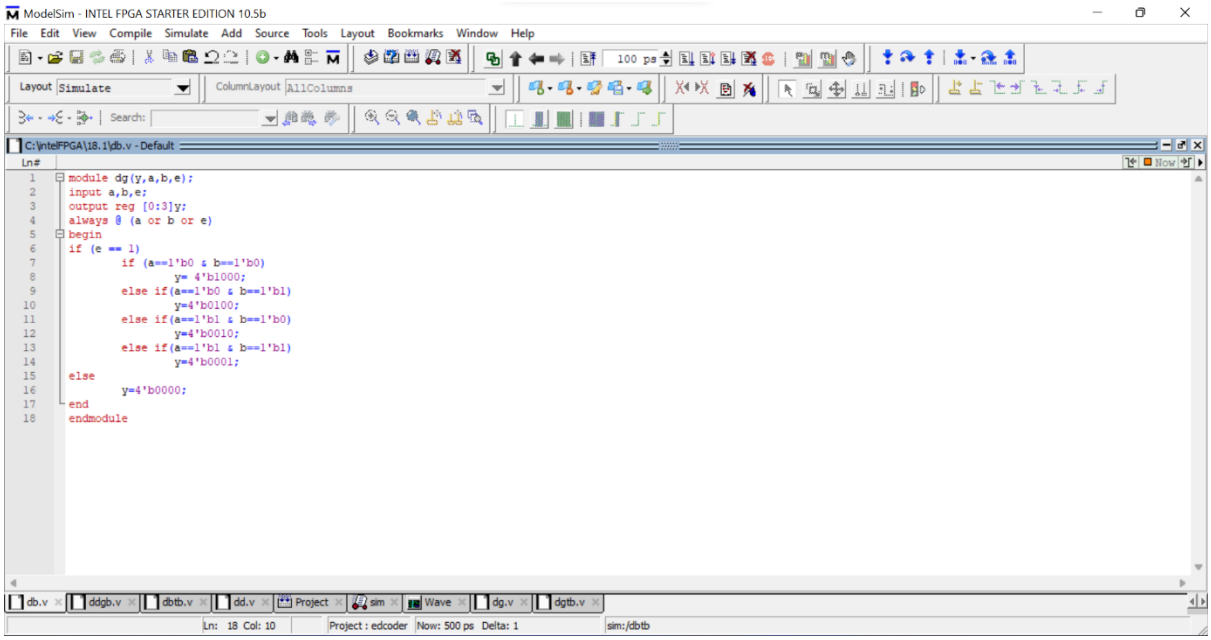
```
#100;

e=1'b1;

a=1'b1;

b=1'b0;

#100;

e=1'b1;

a=1'b1;

b=1'b1;

#100;

end

endmodule
```



Behavioral Modelling:

```
module dg(y,a,b,e);

input a,b,e;

output reg [0:3]y;

always @ (a or b or e)

begin

if (e == 1)
```

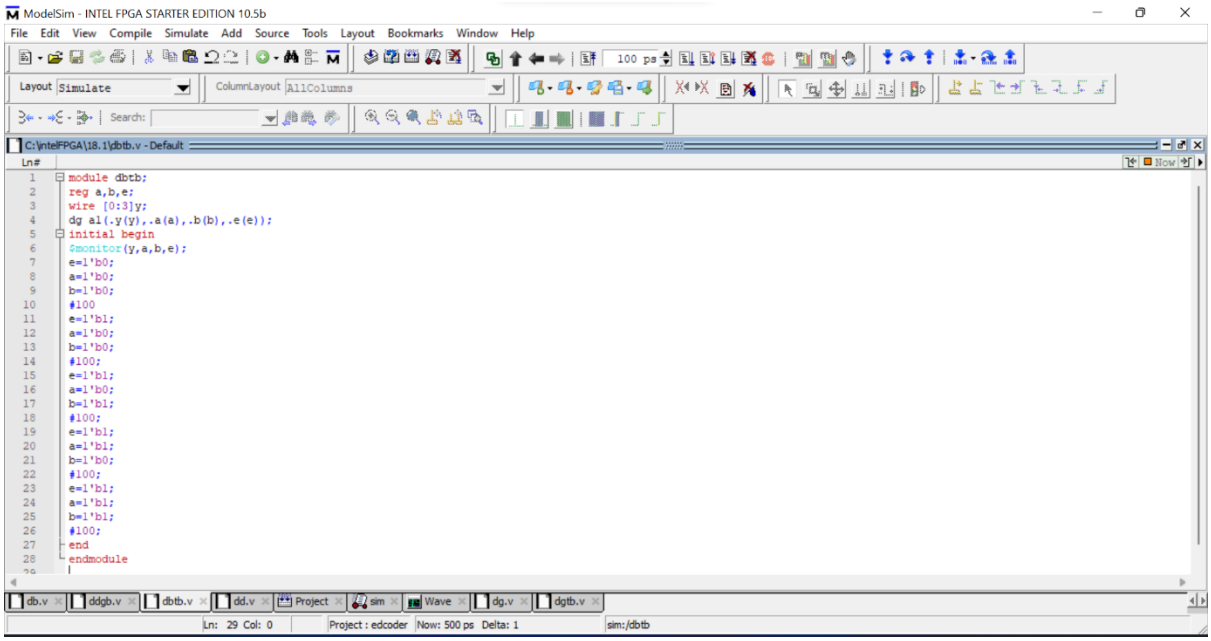
```
        if (a==1'b0 & b==1'b0)
                y= 4'b1000;
        else if(a==1'b0 & b==1'b1)
                y=4'b0100;
        else if(a==1'b1 & b==1'b0)
                y=4'b0010;
        else if(a==1'b1 & b==1'b1)
                y=4'b0001;
else
        y=4'b0000;
end
endmodule
```



Test Bench for Behavioral Modelling:

```
module dbtb;
reg a,b,e;
wire [0:3]y;
dg a1(.y(y),.a(a),.b(b),.e(e));
initial begin
```

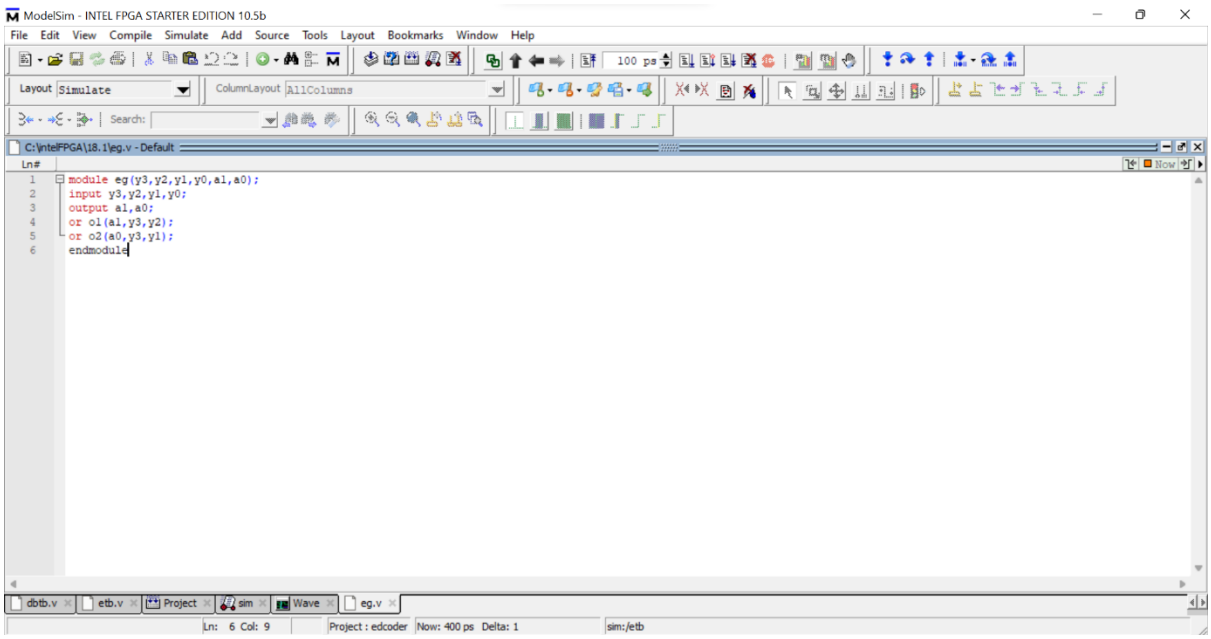
```
$monitor(y,a,b,e);
e=1'b0;
a=1'b0;
b=1'b0;
#100
e=1'b1;
a=1'b0;
b=1'b0;
#100;
e=1'b1;
a=1'b0;
b=1'b1;
#100;
e=1'b1;
a=1'b1;
b=1'b0;
#100;
e=1'b1;
a=1'b1;
b=1'b1;
#100;
end
endmodule
```



4 to 2 Encoder:

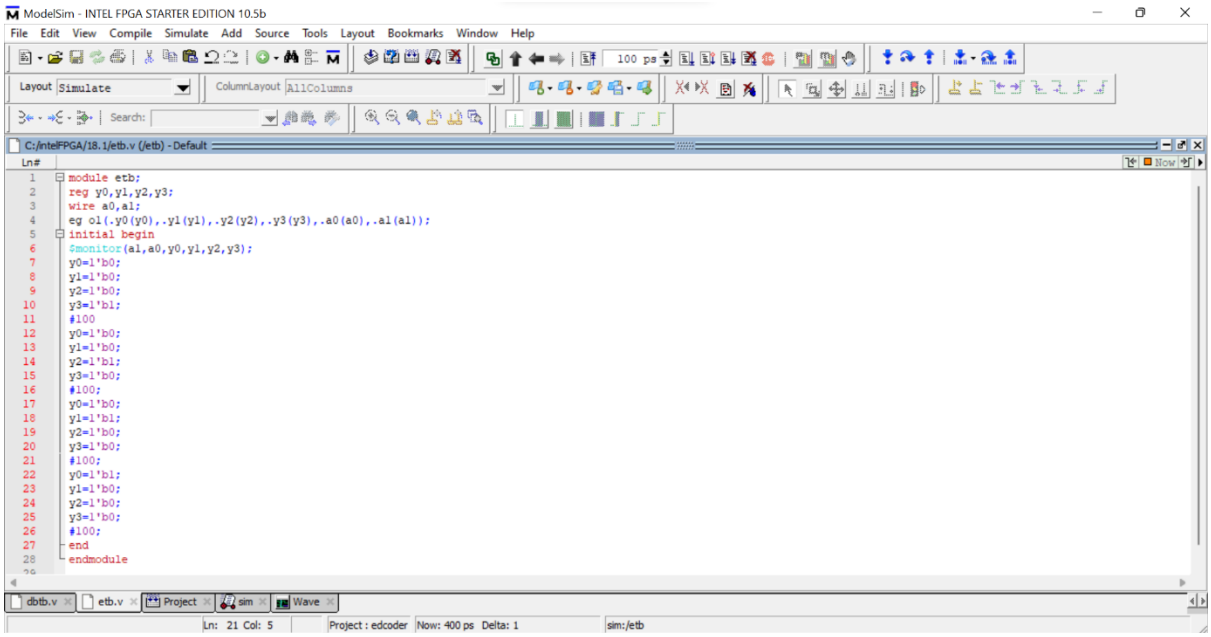
Gate Modelling:

```
module eg(y3,y2,y1,y0,a1,a0);

input y3,y2,y1,y0;

output a1,a0;

or o1(a1,y3,y2);

or o2(a0,y3,y1);

endmodule
```



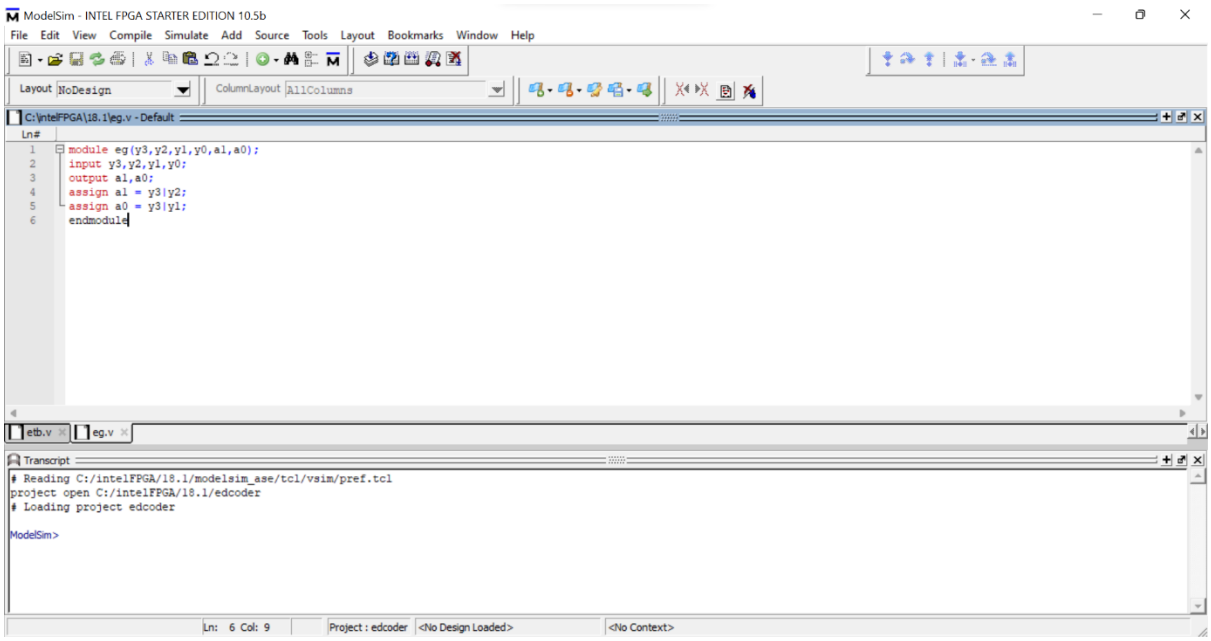
Test Bench for Gate Modelling:

```
module etb;
reg y0,y1,y2,y3;
wire a0,a1;
eg o1(.y0(y0),.y1(y1),.y2(y2),.y3(y3),.a0(a0),.a1(a1));
initial begin
$monitor(a1,a0,y0,y1,y2,y3);
y0=1'b0;
y1=1'b0;
y2=1'b0;
y3=1'b1;
#100
y0=1'b0;
y1=1'b0;
y2=1'b1;
y3=1'b0;
#100;
y0=1'b0;
y1=1'b1;
y2=1'b0;
y3=1'b0;
#100;
y0=1'b1;
y1=1'b0;
y2=1'b0;
y3=1'b0;
#100;
end
endmodule
```



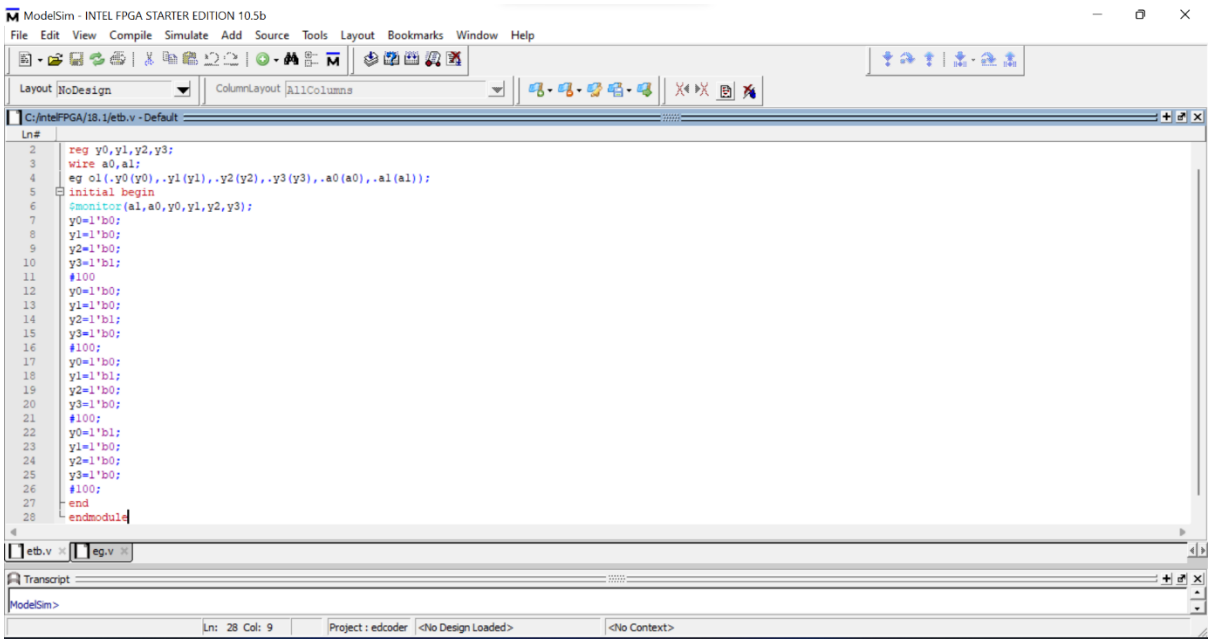
Data-Flow Modelling:

```
module eg(y3,y2,y1,y0,a1,a0);

input y3,y2,y1,y0;

output a1,a0;

assign a1 = y3|y2;

assign a0 = y3|y1;

endmodule
```



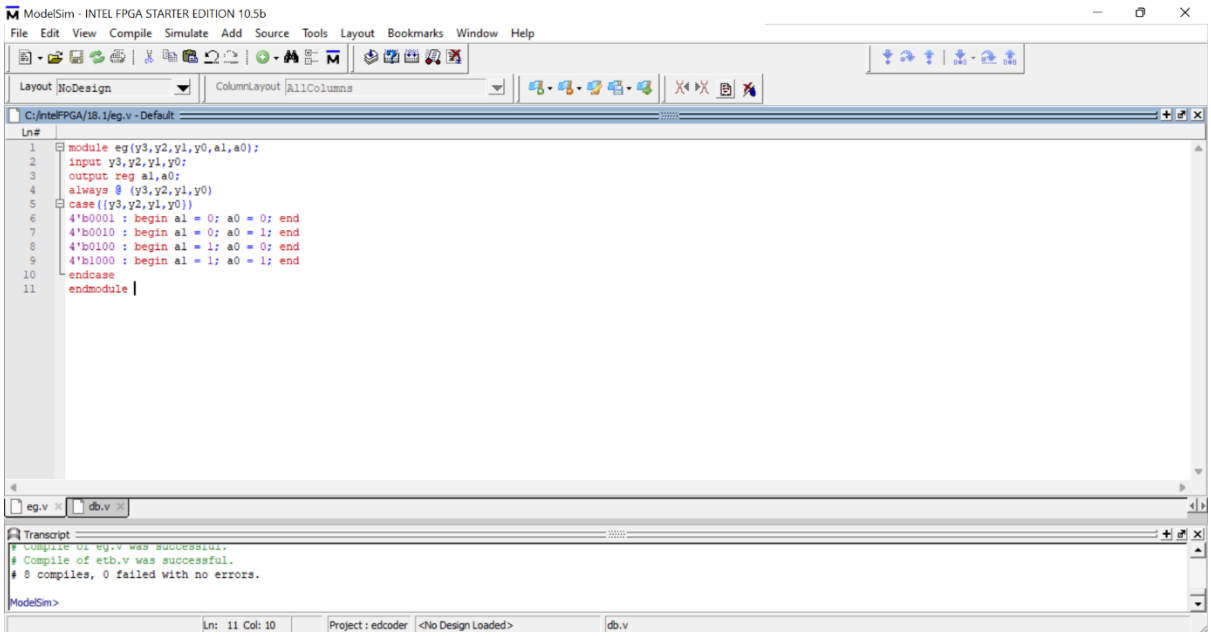
Test Bench for Data-Flow Modelling:

```
module etb;
reg y0,y1,y2,y3;
wire a0,a1;
eg o1(.y0(y0),.y1(y1),.y2(y2),.y3(y3),.a0(a0),.a1(a1));
initial begin
$monitor(a1,a0,y0,y1,y2,y3);
y0=1'b0;
y1=1'b0;
y2=1'b0;
y3=1'b1;
#100
y0=1'b0;
y1=1'b0;
y2=1'b1;
y3=1'b0;
#100;
y0=1'b0;
y1=1'b1;
y2=1'b0;
y3=1'b0;
#100;
y0=1'b1;
y1=1'b0;
y2=1'b0;
y3=1'b0;
#100;
end
endmodule
```



Behavioral Modelling:

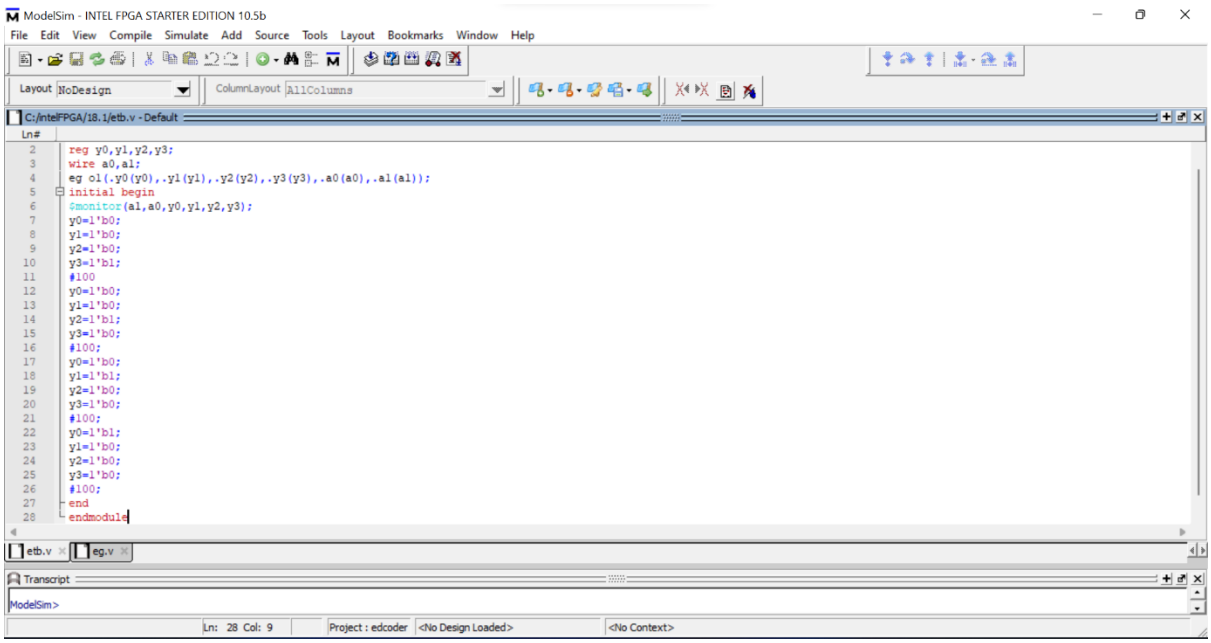
```
module eg(y3,y2,y1,y0,a1,a0);

input y3,y2,y1,y0;

output reg a1,a0;

always @ (y3,y2,y1,y0)

case({y3,y2,y1,y0})

4'b0001 : begin a1 = 0; a0 = 0; end

4'b0010 : begin a1 = 0; a0 = 1; end

4'b0100 : begin a1 = 1; a0 = 0; end

4'b1000 : begin a1 = 1; a0 = 1; end

endcase

endmodule
```



Test Bench for Behavioral Modelling:

```
module etb;

reg y0,y1,y2,y3;

wire a0,a1;

eg o1(.y0(y0),.y1(y1),.y2(y2),.y3(y3),.a0(a0),.a1(a1));

initial begin

$monitor(a1,a0,y0,y1,y2,y3);

y0=1'b0;

y1=1'b0;

y2=1'b0;

y3=1'b1;

#100

y0=1'b0;

y1=1'b0;

y2=1'b1;

y3=1'b0;

#100;

y0=1'b0;

y1=1'b1;
```

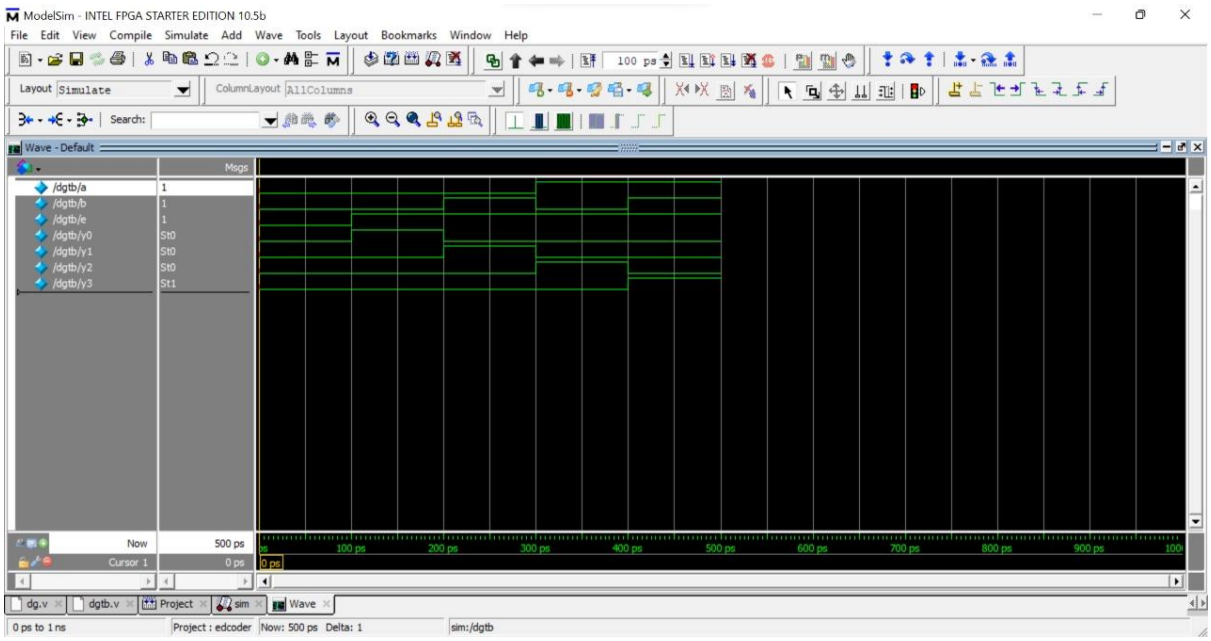
```
y2=1'b0;

y3=1'b0;

#100;

y0=1'b1;

y1=1'b0;

y2=1'b0;

y3=1'b0;

#100;

end

endmodule
```



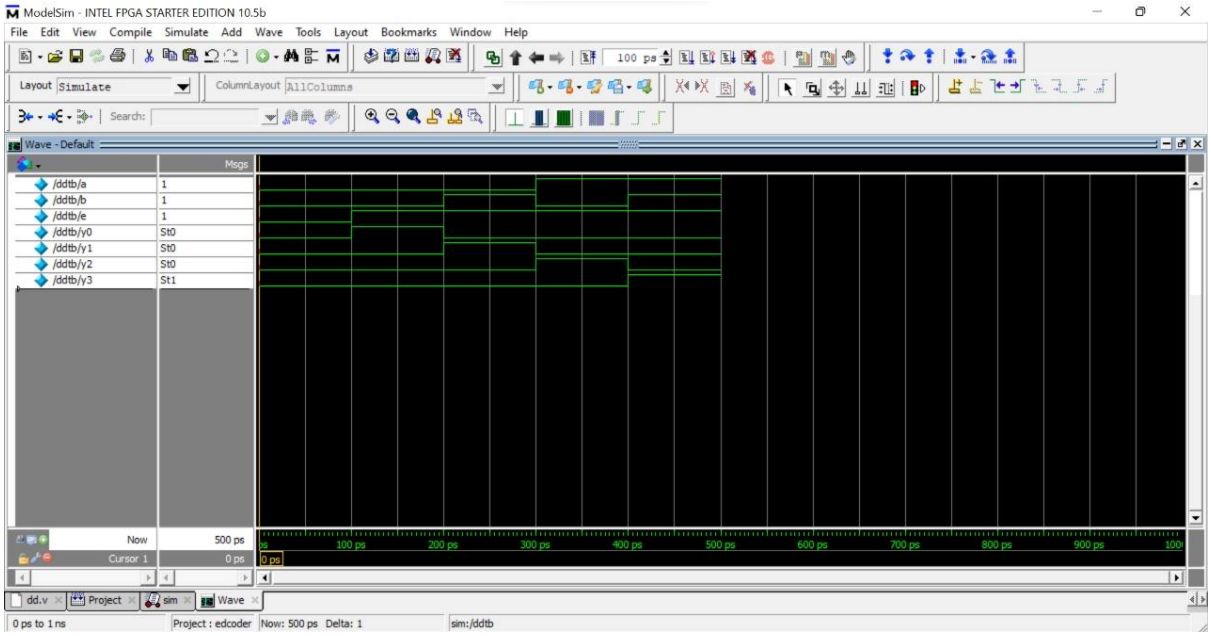
Output:

2 to 4 Decoder:

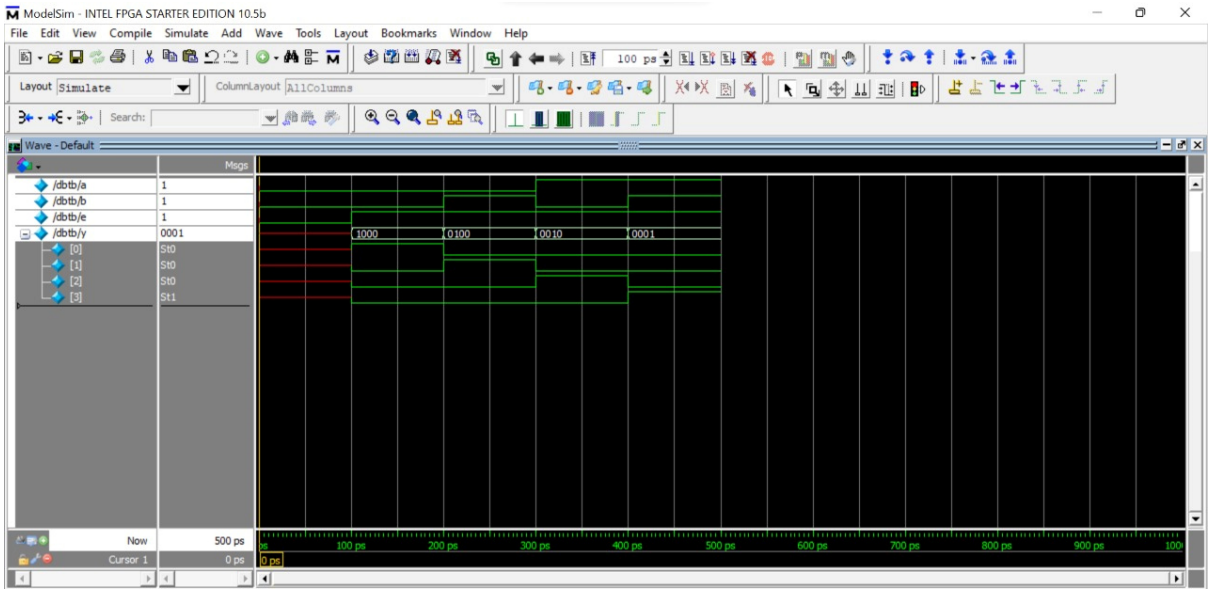
Gate Modelling:



Data Flow Modelling:

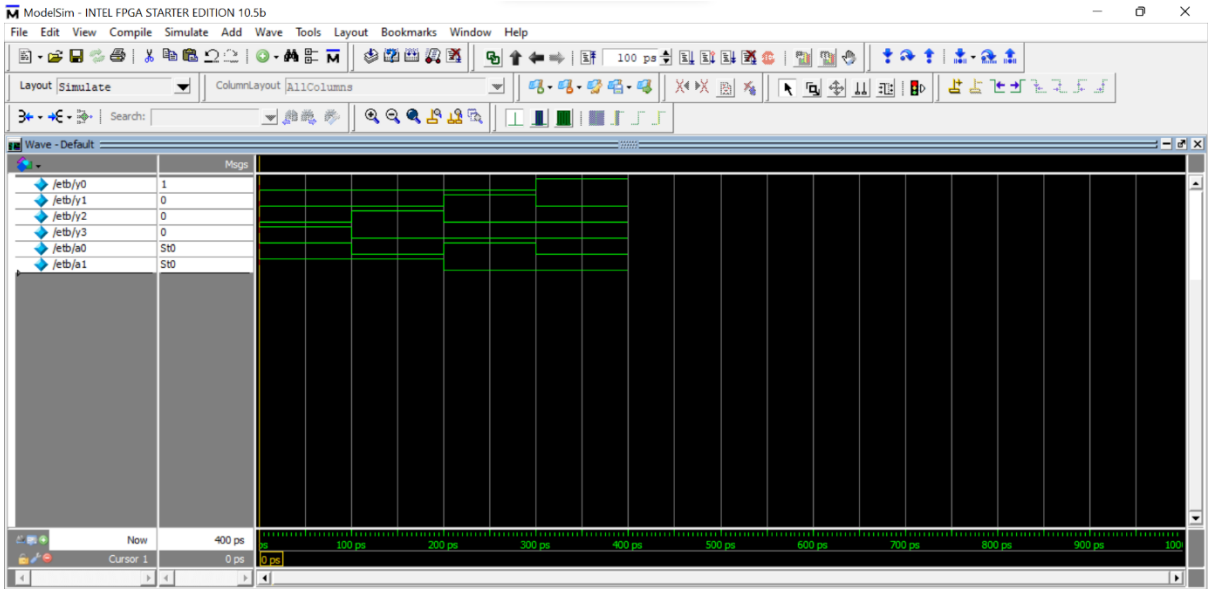


Behavioral Modelling:

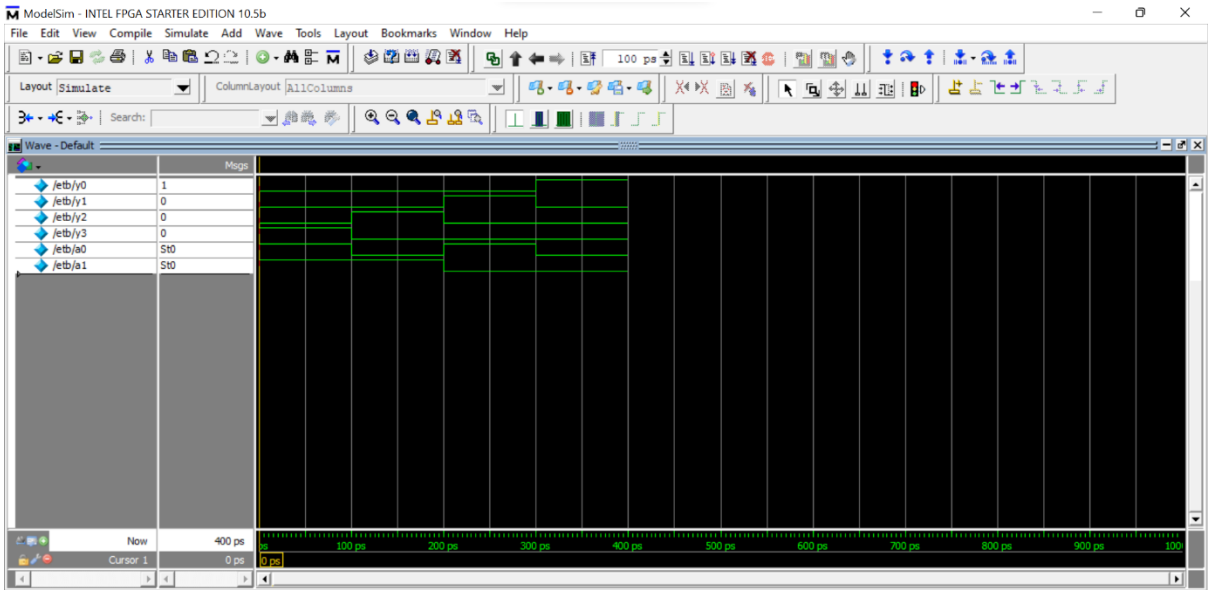


4 to 2 Encoder:

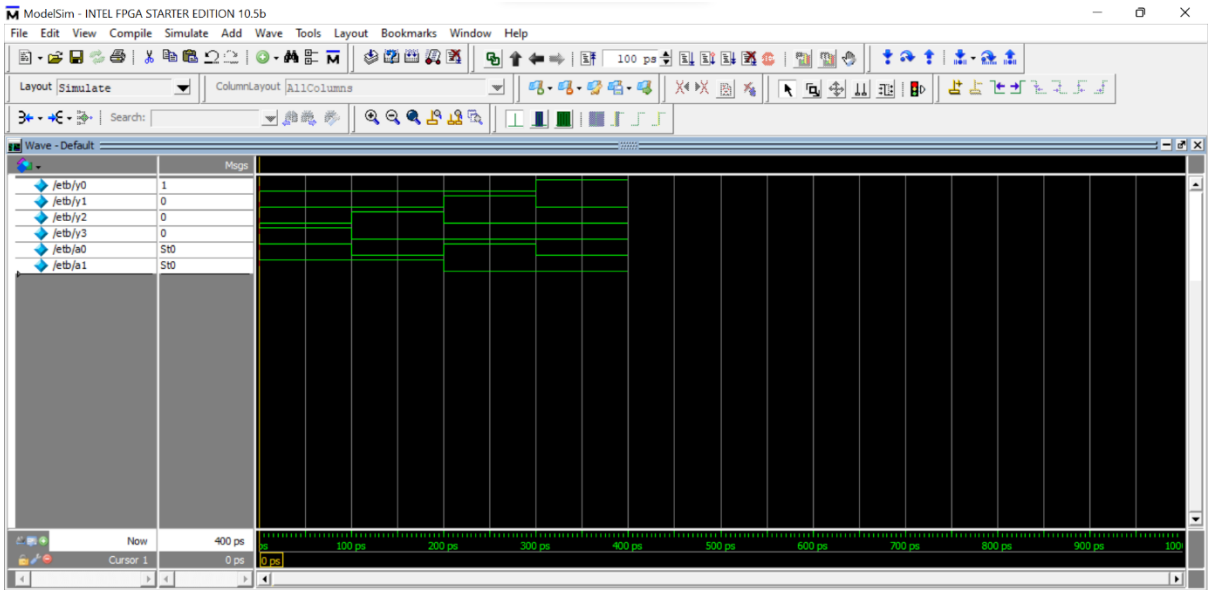
Gate Modelling:



Data Flow Modelling:



Behavioral Modelling:



Result: The encoder and decoder were designed and constructed using logic gates and their truth table was verified using Verilog.

Experiment No: 6 (Software) **Date: 02.10.2022**

Multiplexer and Demultiplexer

Aim:

To implement 4:1 mux and 1:4 demux using logic gates in verilog programming.

Software Required: ModelSim

Procedure:

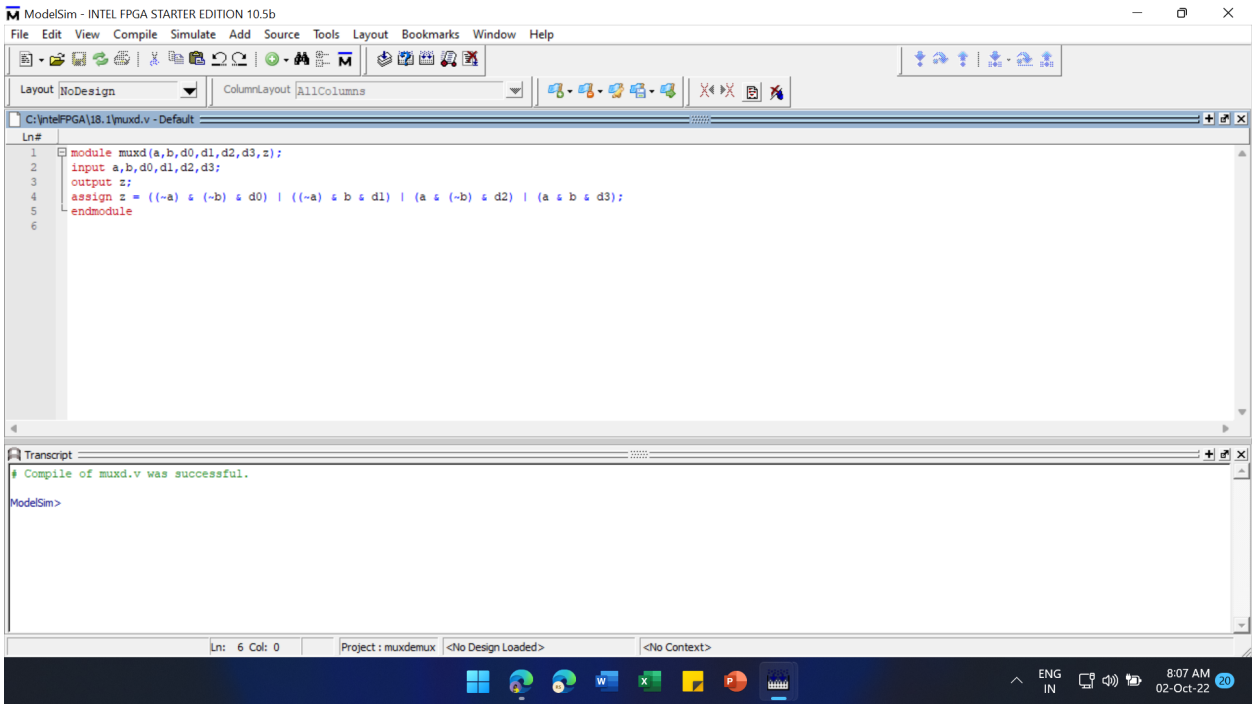
1. Open ModelSim.
2. Create New Project, then create a new file by selecting the verilog in the drop-down.
3. Type the code, then compile it.
4. Type the test bench, compile and simulate it.
5. Click add wave in the window and then run it.

Verilog Code:

Multiplexer:

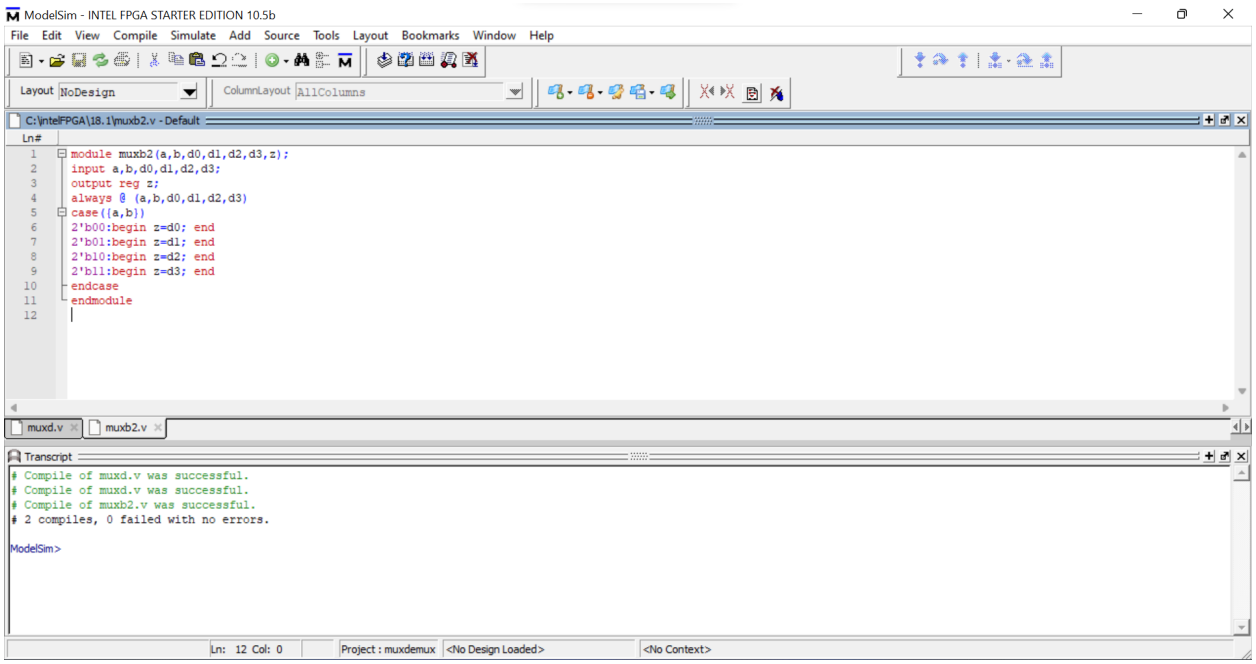
Data-Flow Modeling:

```
module muxd(a,b,d0,d1,d2,d3,z);
input a,b,d0,d1,d2,d3;
output z;
assign z = ((~a) & (~b) & d0) | ((~a) & b & d1) | (a & (~b) & d2) | (a & b & d3);
endmodule
```



Behavioral Modeling:

```
module muxb2(a,b,d0,d1,d2,d3,z);
input a,b,d0,d1,d2,d3;
output reg z;
always @ (a,b,d0,d1,d2,d3)
case({a,b})
2'b00:begin z=d0; end
2'b01:begin z=d1; end
2'b10:begin z=d2; end
2'b11:begin z=d3; end
endcase
endmodule
```

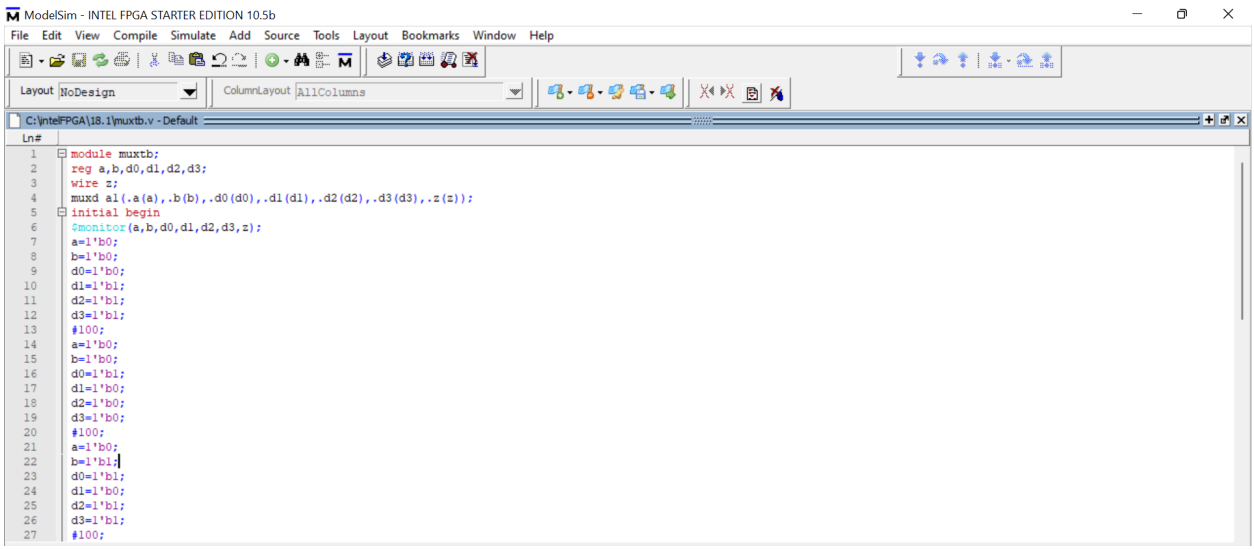


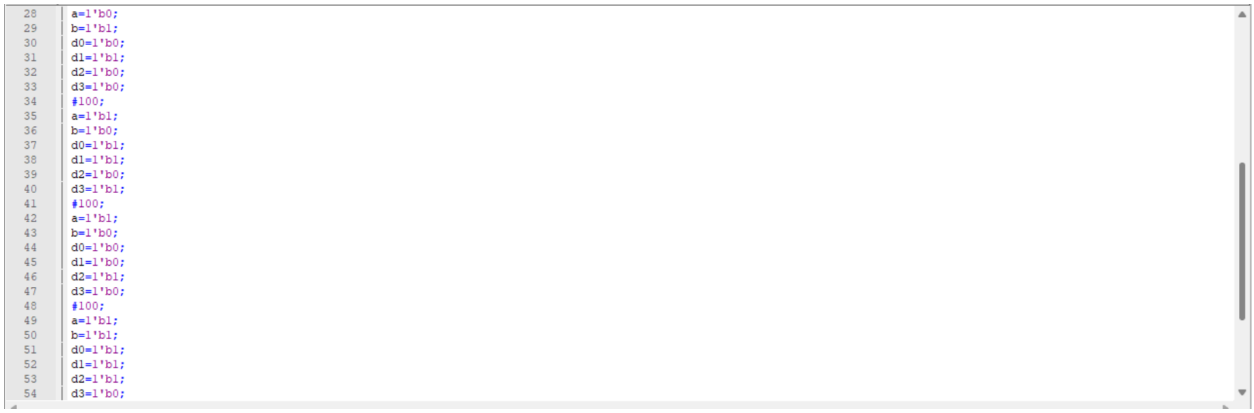
Test Bench:

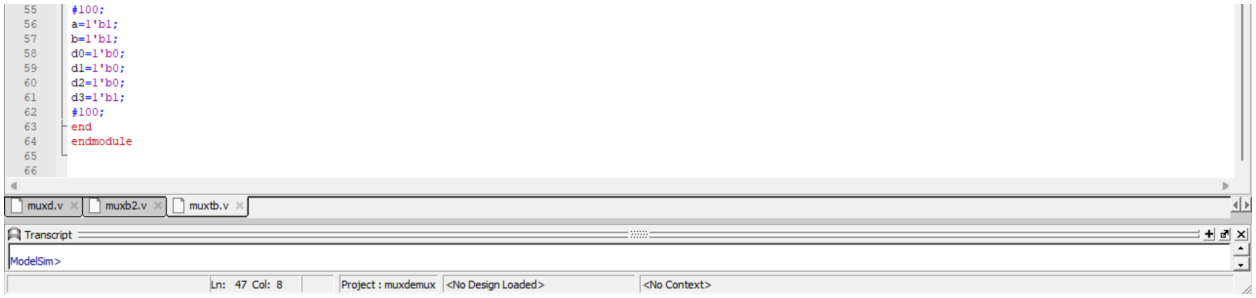
```
module muxtb;
reg a,b,d0,d1,d2,d3;
wire z;
muxd a1(.a(a),.b(b),.d0(d0),.d1(d1),.d2(d2),.d3(d3),.z(z));
initial begin
$monitor(a,b,d0,d1,d2,d3,z);
a=1'b0;
b=1'b0;
d0=1'b0;
d1=1'b1;
d2=1'b1;
d3=1'b1;
#100;
a=1'b0;
b=1'b0;
d0=1'b1;
d1=1'b0;
d2=1'b0;
d3=1'b0;
```

```
#100;
a=1'b0;
b=1'b1;
d0=1'b1;
d1=1'b0;
d2=1'b1;
d3=1'b1;
#100;
a=1'b0;
b=1'b1;
d0=1'b0;
d1=1'b1;
d2=1'b0;
d3=1'b0;
#100;
a=1'b1;
b=1'b0;
d0=1'b1;
d1=1'b1;
d2=1'b0;
d3=1'b1;
#100;
a=1'b1;
b=1'b0;
d0=1'b0;
d1=1'b0;
d2=1'b1;
d3=1'b0;
#100;
a=1'b1;
b=1'b1;
d0=1'b1;
d1=1'b1;
d2=1'b1;
```

```
d3=1'b0;
#100;
a=1'b1;
b=1'b1;
d0=1'b0;
d1=1'b0;
d2=1'b0;
d3=1'b1;
#100;
end
endmodule
```



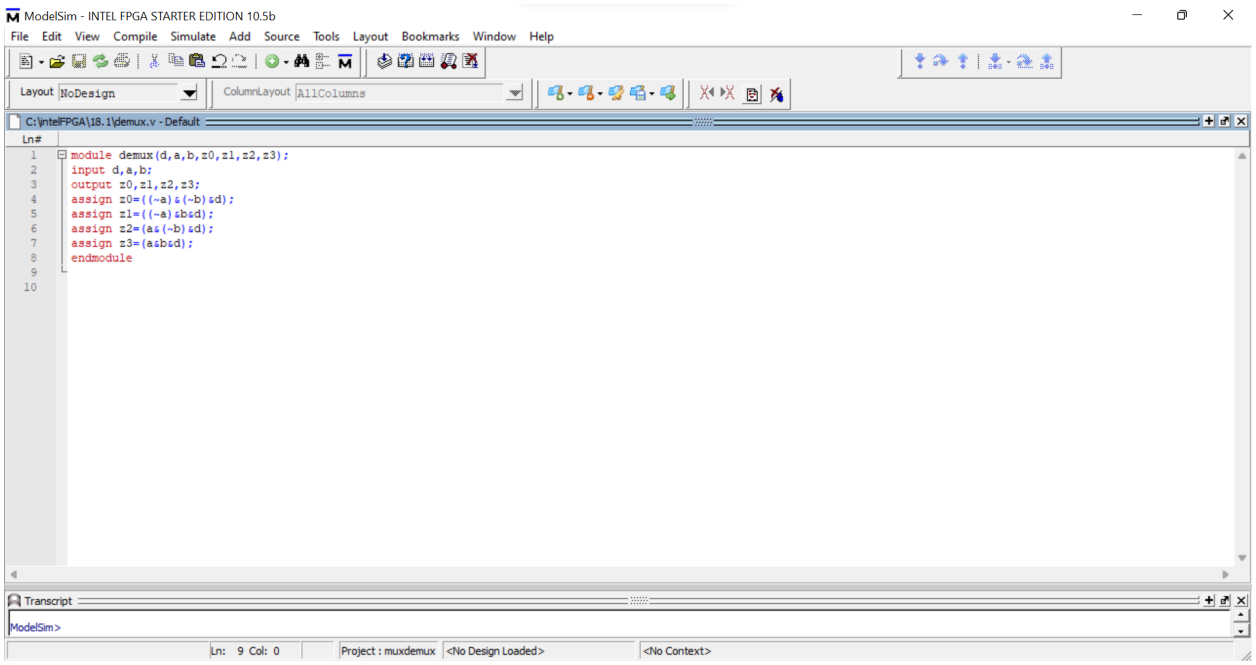




Demultiplexer:

Data-Flow Modeling:

```
module demux(d,a,b,z0,z1,z2,z3);
input d,a,b;
output z0,z1,z2,z3;
assign z0=((~a)&(~b)&d);
assign z1=((~a)&b&d);
assign z2=(a&(~b)&d);
assign z3=(a&b&d);
endmodule
```



Behavioral Modelling:

```
module demuxb(a,b,d,z0,z1,z2,z3);
input a,b,d;
output reg z0,z1,z2,z3;
```

always @ (a,b,d)

case({a,b})

2'b00:begin z0=d; z1=0; z2=0; z3=0; end

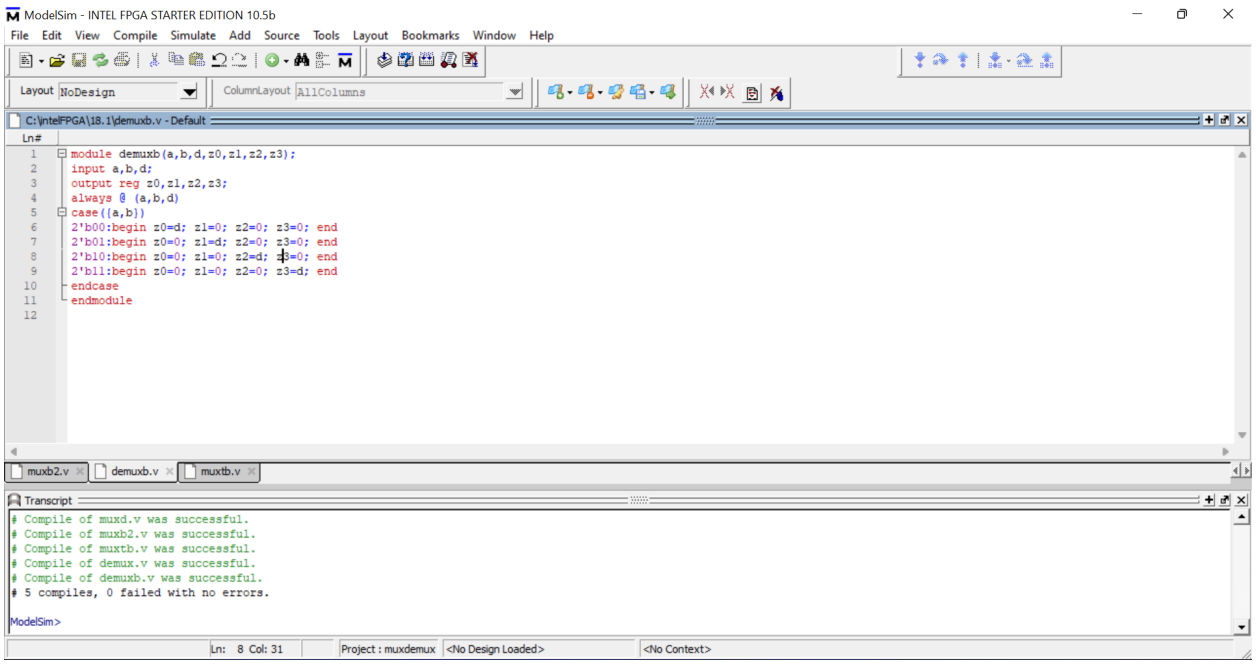
2'b01:begin z0=0; z1=d; z2=0; z3=0; end

2'b10:begin z0=0; z1=0; z2=d; z3=0; end

2'b11:begin z0=0; z1=0; z2=0; z3=d; end

endcase

endmodule



Test Bench:

module demuxtb;

reg a,b,d;

wire z1,z2,z3,z0;

demuxb a1(.a(a),.b(b),.z0(z0),.z1(z1),.z2(z2),.z3(z3),.d(d));

initial begin

$monitor(a,b,z0,z1,z2,z3,d);

a=1'b0;

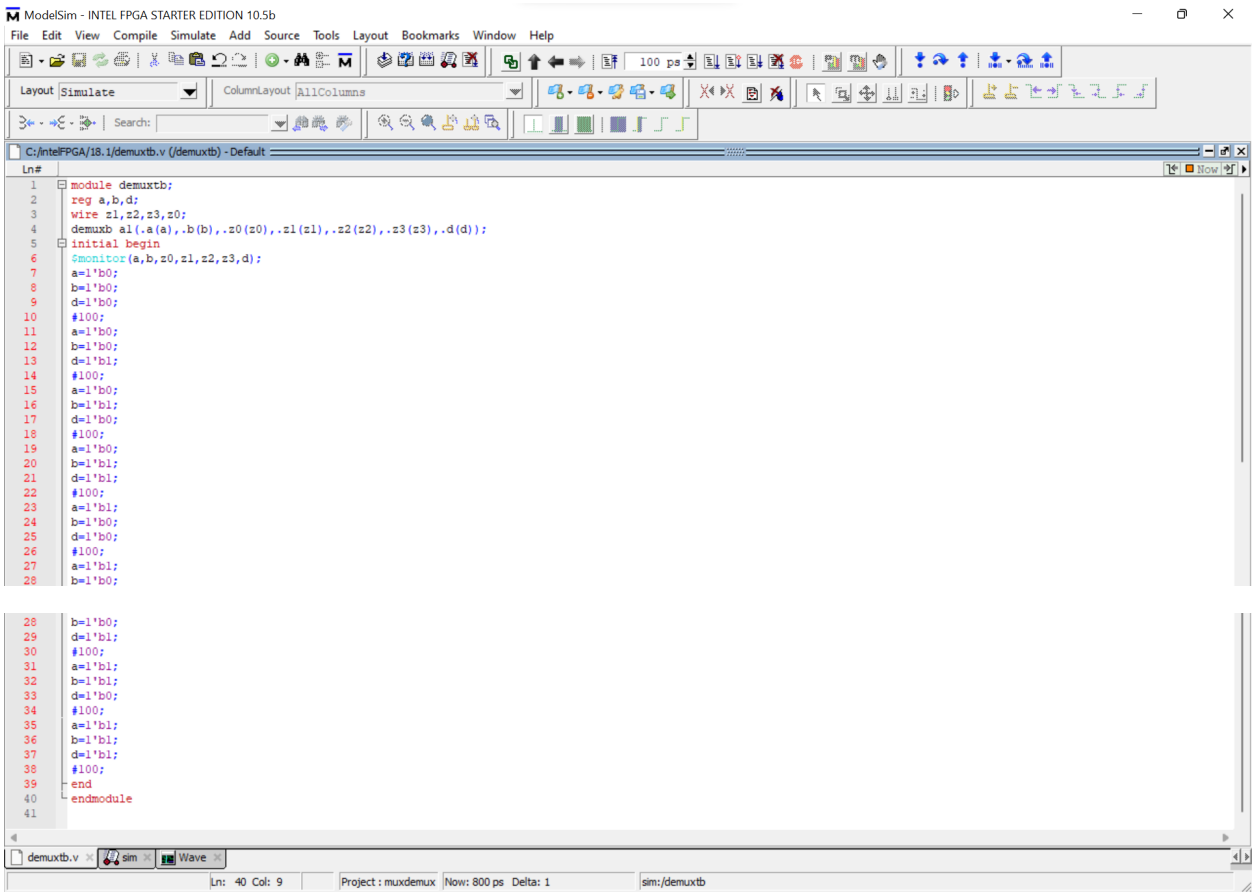
b=1'b0;

d=1'b0;

#100;

a=1'b0;

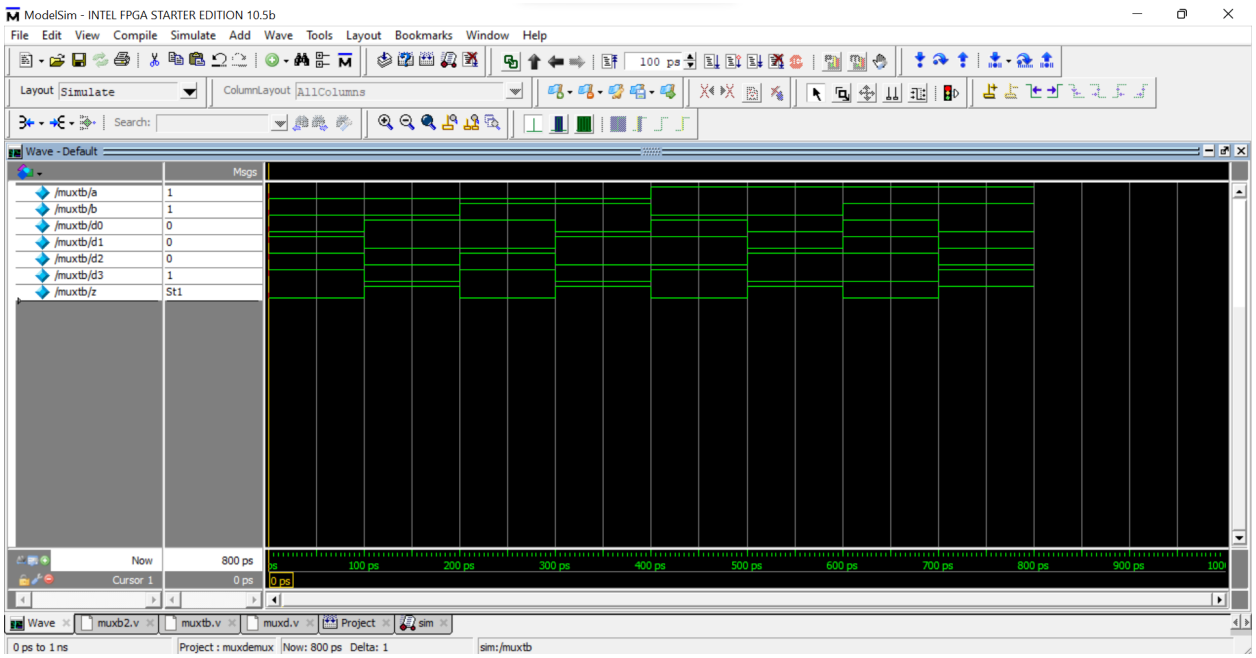
```
b=1'b0;
d=1'b1;
#100;
a=1'b0;
b=1'b1;
d=1'b0;
#100;
a=1'b0;
b=1'b1;
d=1'b1;
#100;
a=1'b1;
b=1'b0;
d=1'b0;
#100;
a=1'b1;
b=1'b0;
d=1'b1;
#100;
a=1'b1;
b=1'b1;
d=1'b0;
#100;
a=1'b1;
b=1'b1;
d=1'b1;
#100;
end
endmodule
```



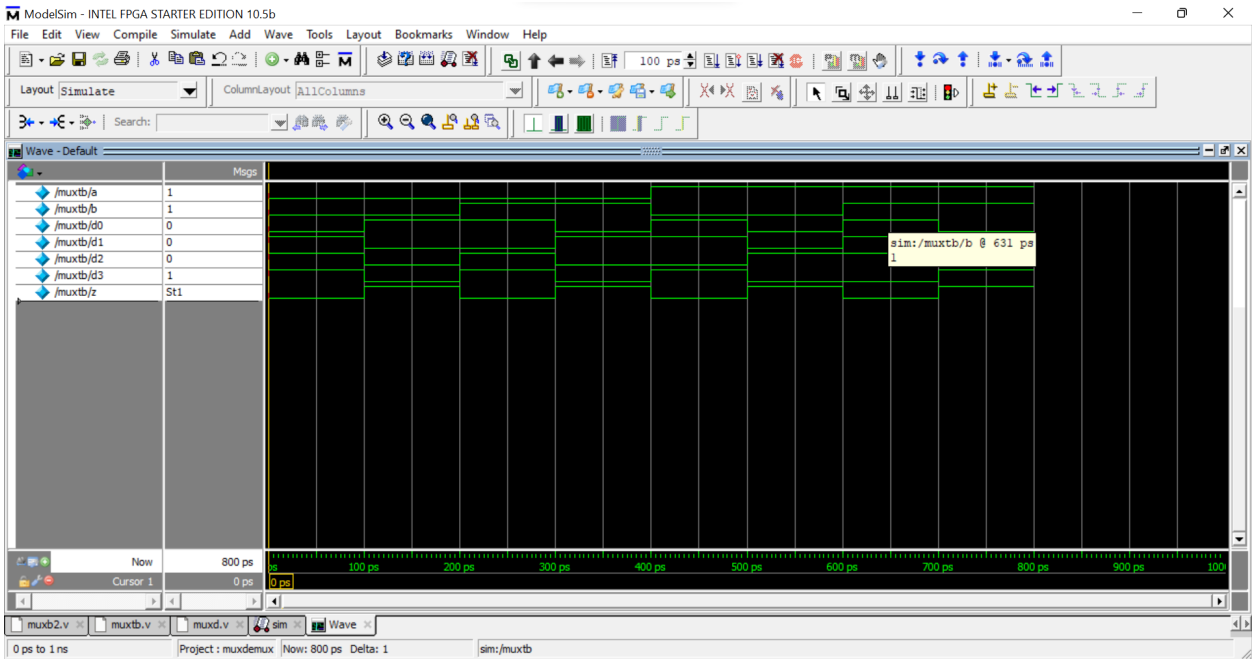
Output:

Multiplexer:

Data-Flow Modeling:

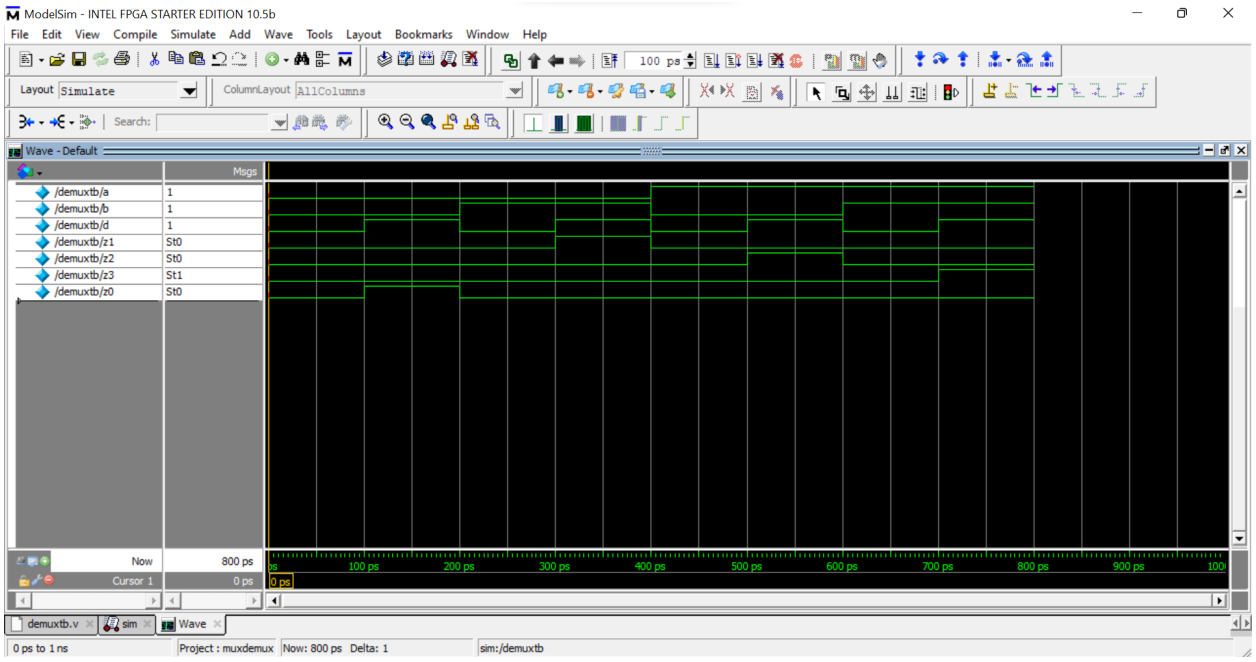


Behavioral Modeling:

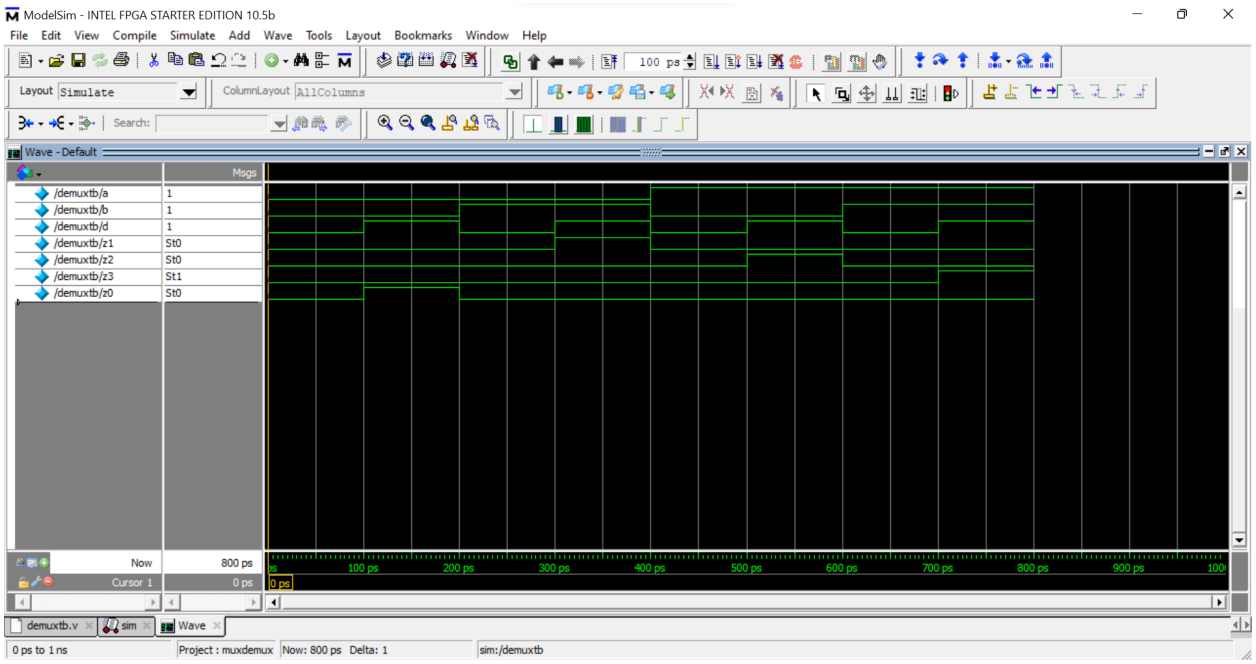


Demultiplexer:

Data-Flow Modeling:



Behavioral Modeling:



Result/ Inference:

From this experiment, the multiplexer and demultiplexer combinational circuits are verified using data-flow modeling and behavioral modeling using verilog programming.

Experiment No. 7 (Software) **Date: 12.11.2022**

Counters

Aim:

To design and verify up-down, up, down counters using Verilog Programming.

Software Required:

ModelSim

Procedure:

1. Open ModelSim.
2. Create New Project, then create a new file by selecting the verilog in the drop-down.
3. Type the code, then compile it.
4. Type the test bench, compile and simulate it.
5. Click add wave in the window and then run it.

Program Code:

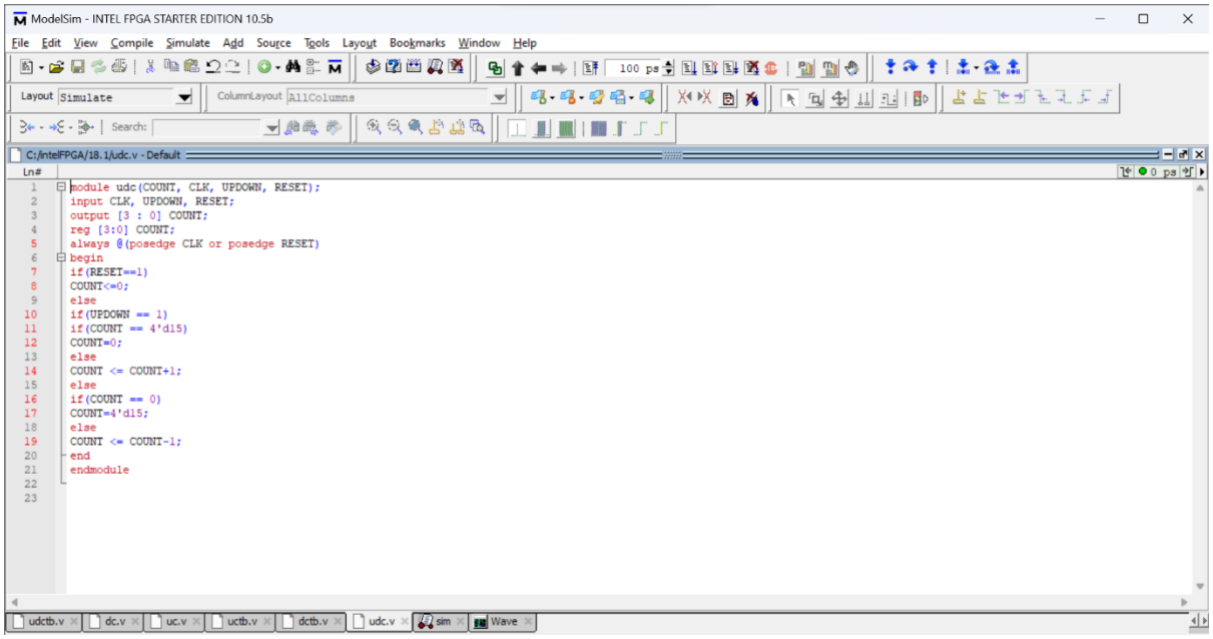
Up-Down Counter:

```
module udc(COUNT, CLK, UPDOWN, RESET);
input CLK, UPDOWN, RESET;
output [3 : 0] COUNT;
reg [3:0] COUNT;
always @(posedge CLK or posedge RESET)
begin
if(RESET==1)
COUNT<=0;
else
if(UPDOWN == 1)     //Up mode selected
if(COUNT == 4'd15)
COUNT=0;
else
COUNT <= COUNT+1;
else
if(COUNT == 0)
COUNT=4'd15;
else
```

COUNT <= COUNT-1;

end

endmodule



Test Bench:

module udctb;

reg CLK, UPDOWN, RESET;

wire [3 : 0] COUNT;

udc uut(COUNT, CLK, UPDOWN, RESET);

always #5 CLK=~CLK;

initial

begin

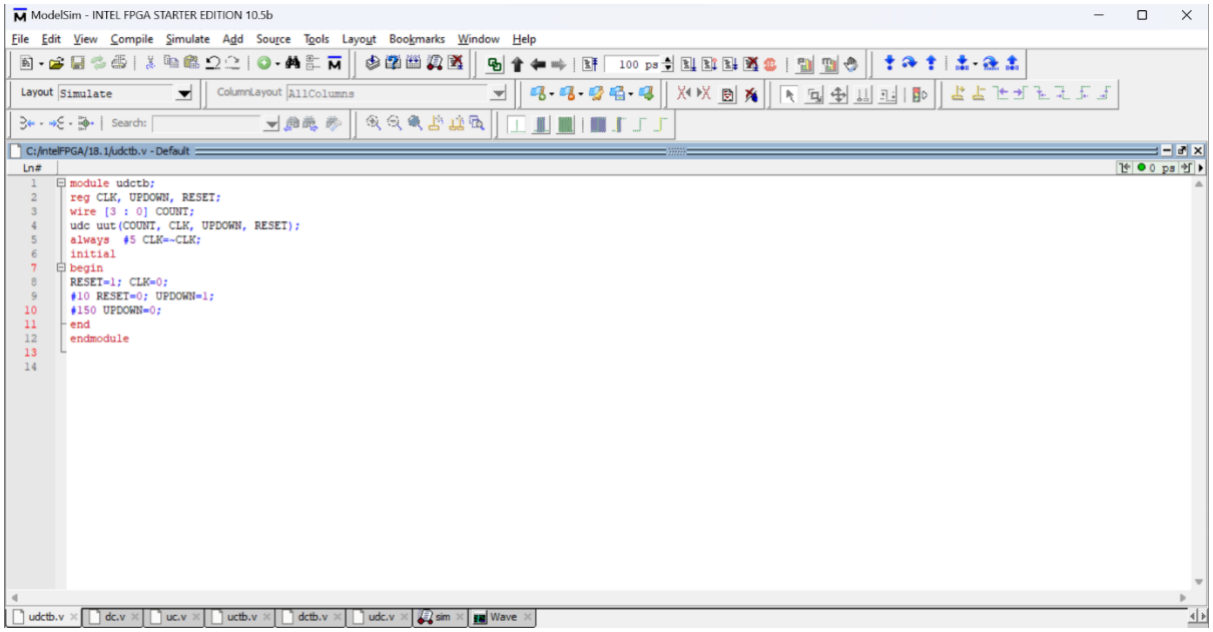
RESET=1; CLK=0;

#10 RESET=0; UPDOWN=1;

#150 UPDOWN=0;

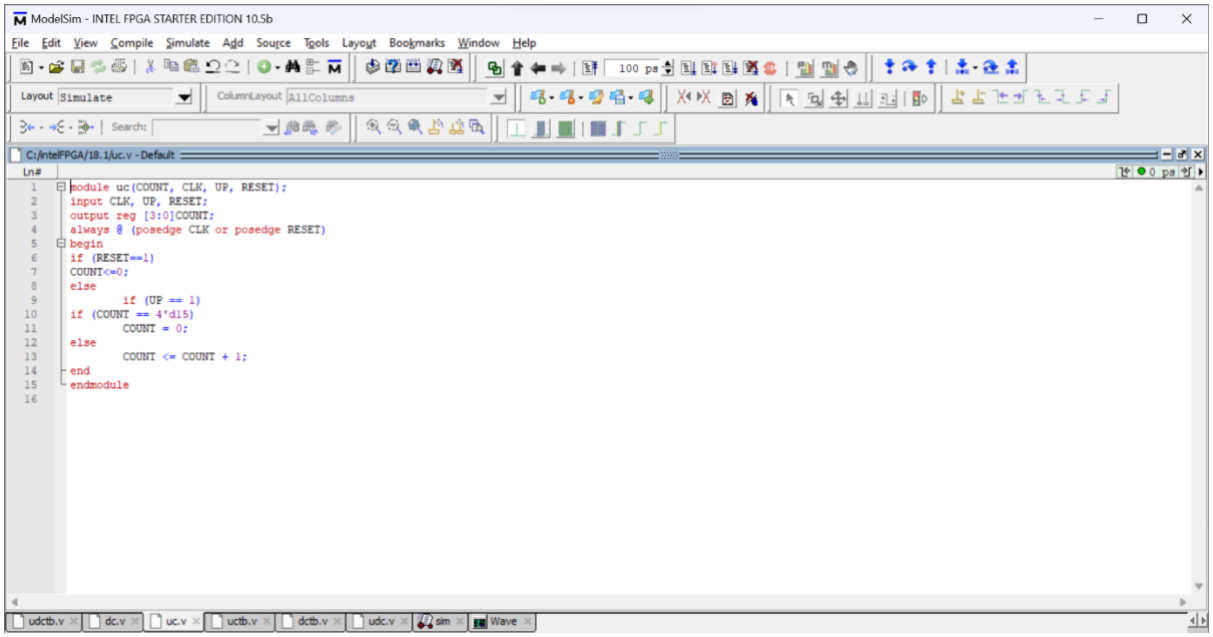
end

endmodule



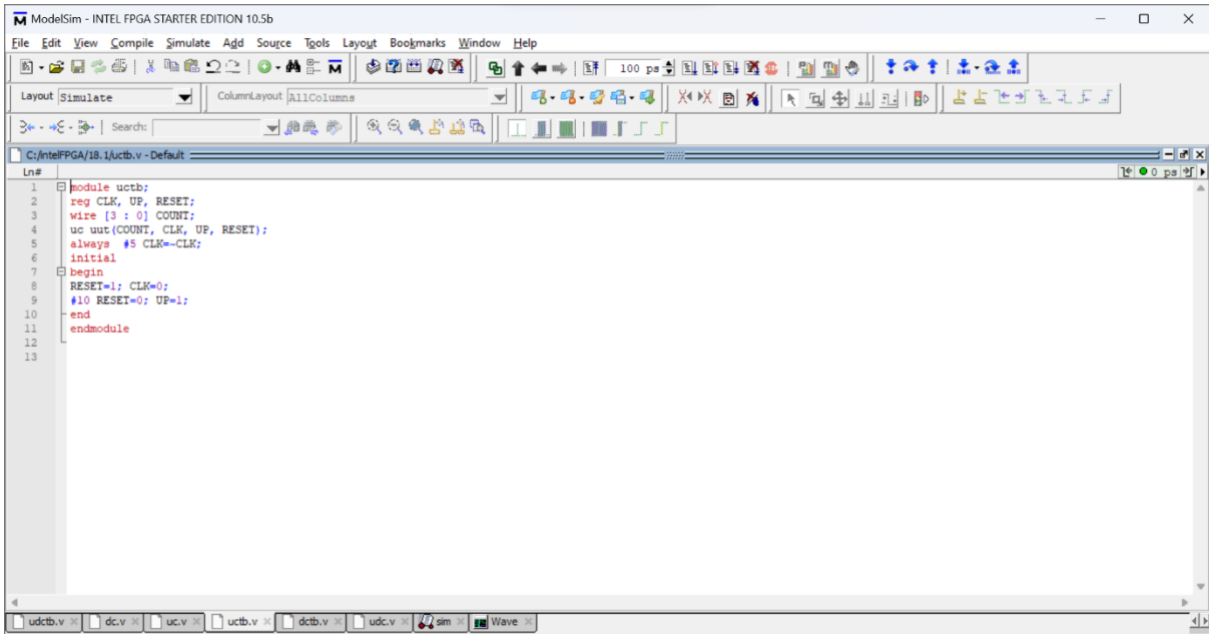
Up Counter:

```
module uc(COUNT, CLK, UP, RESET);
input CLK, UP, RESET;
output [3 : 0] COUNT;
reg [3:0] COUNT;
always @(posedge CLK or posedge RESET)
begin
if(RESET==1)
COUNT<=0;
else
        if(UP == 1)
if(COUNT == 4'd15)
        COUNT=0;
else
COUNT <= COUNT+1;
end
endmodule
```



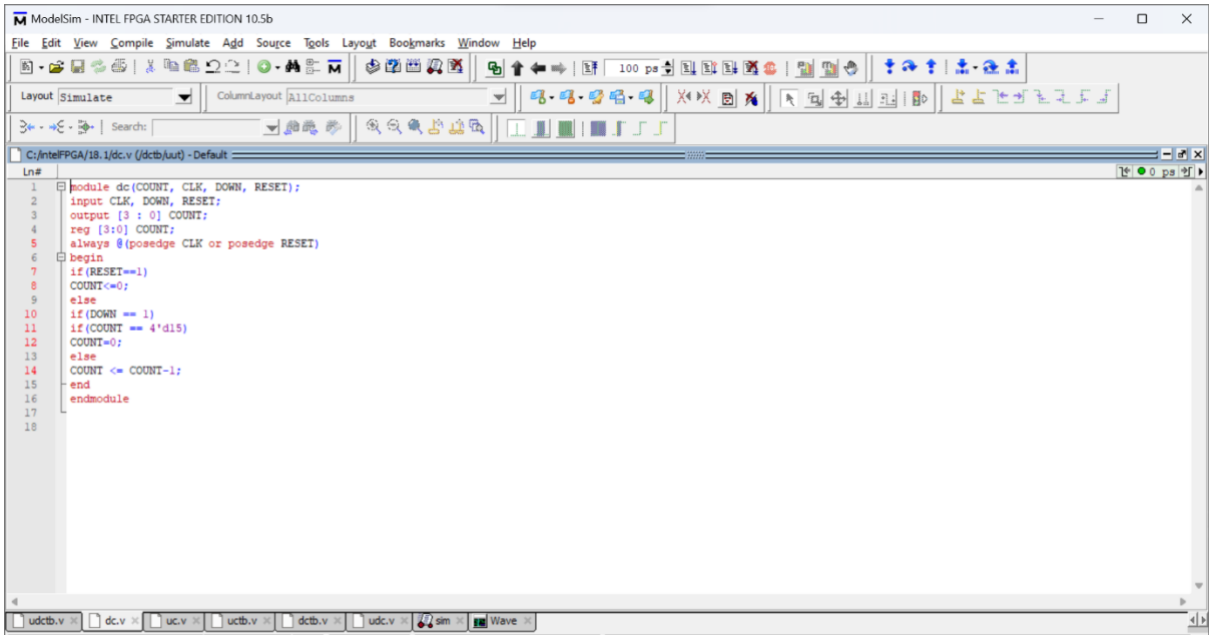
Test Bench:

```
module uctb;
reg CLK, UP, RESET;
wire [3 : 0] COUNT;
uc uut(COUNT, CLK, UP, RESET);
always  #5 CLK=~CLK;
initial
begin
RESET=1; CLK=0;
#10 RESET=0; UP=1;
end
endmodule
```



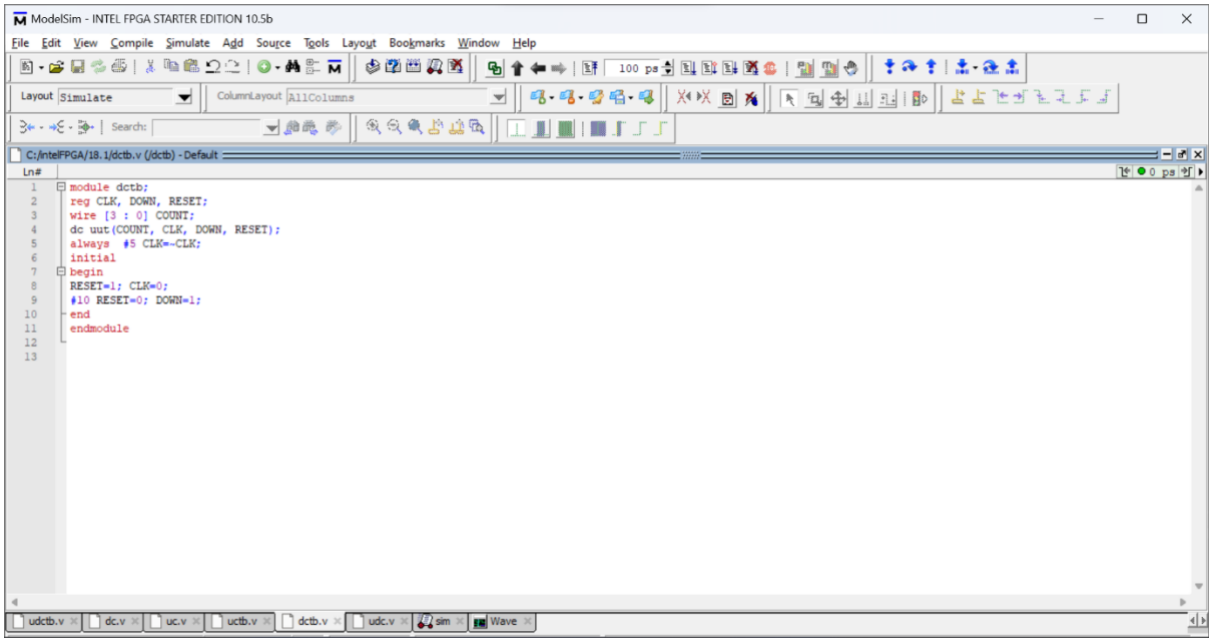
Down Counter:

```
module dc(COUNT, CLK, DOWN, RESET);
input CLK, DOWN, RESET;
output [3 : 0] COUNT;
reg [3:0] COUNT;
always @(posedge CLK or posedge RESET)
begin
if(RESET==1)
COUNT<=0;
else
if(DOWN == 1)
if(COUNT == 4'd15)
COUNT=0;
else
COUNT <= COUNT-1;
end
endmodule
```



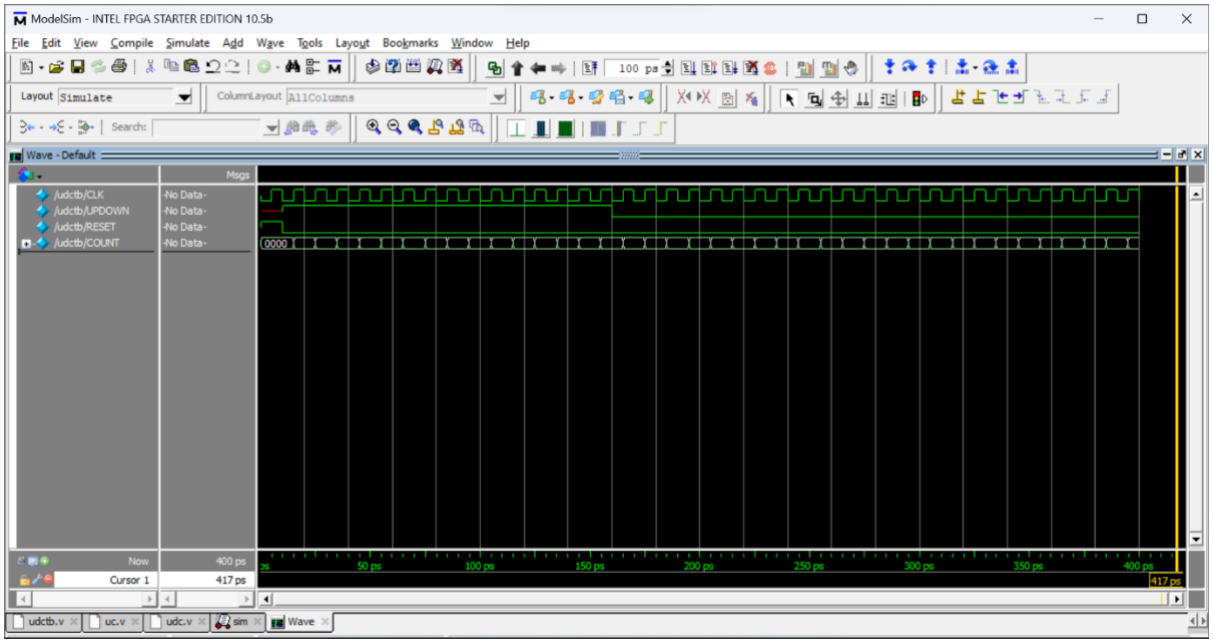
Test Bench:

```
module dctb;
reg CLK, DOWN, RESET;
wire [3 : 0] COUNT;
dc uut(COUNT, CLK, DOWN, RESET);
always  #5 CLK=~CLK;
initial
begin
RESET=1; CLK=0;
#10 RESET=0; DOWN=1;
end
endmodule
```

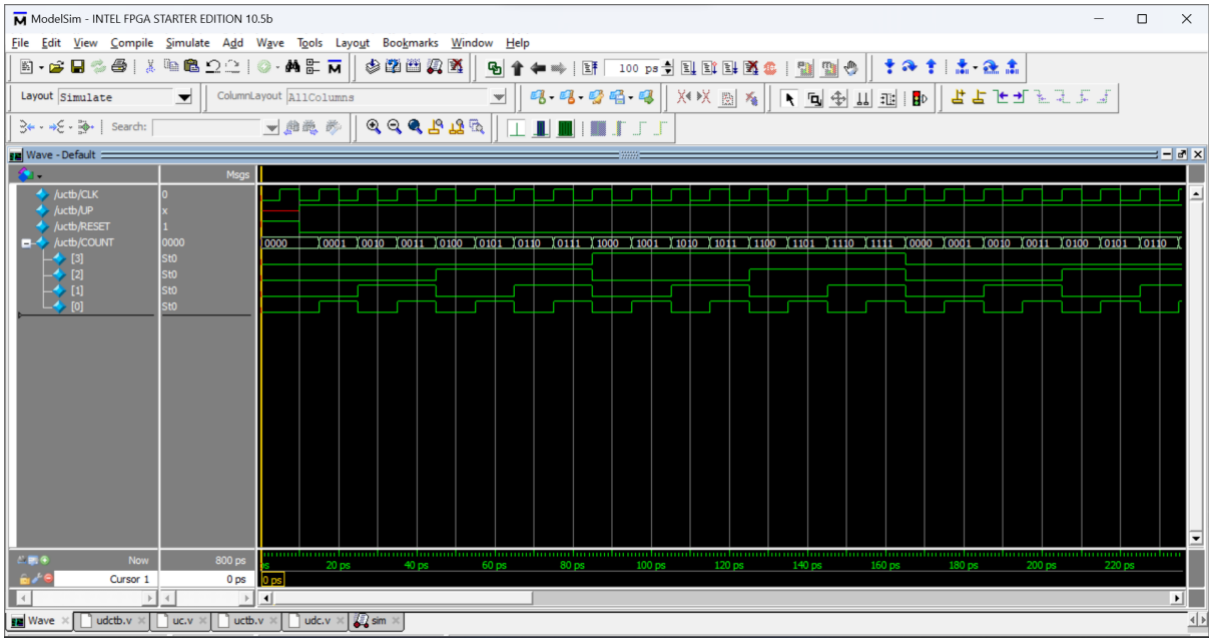


Output:

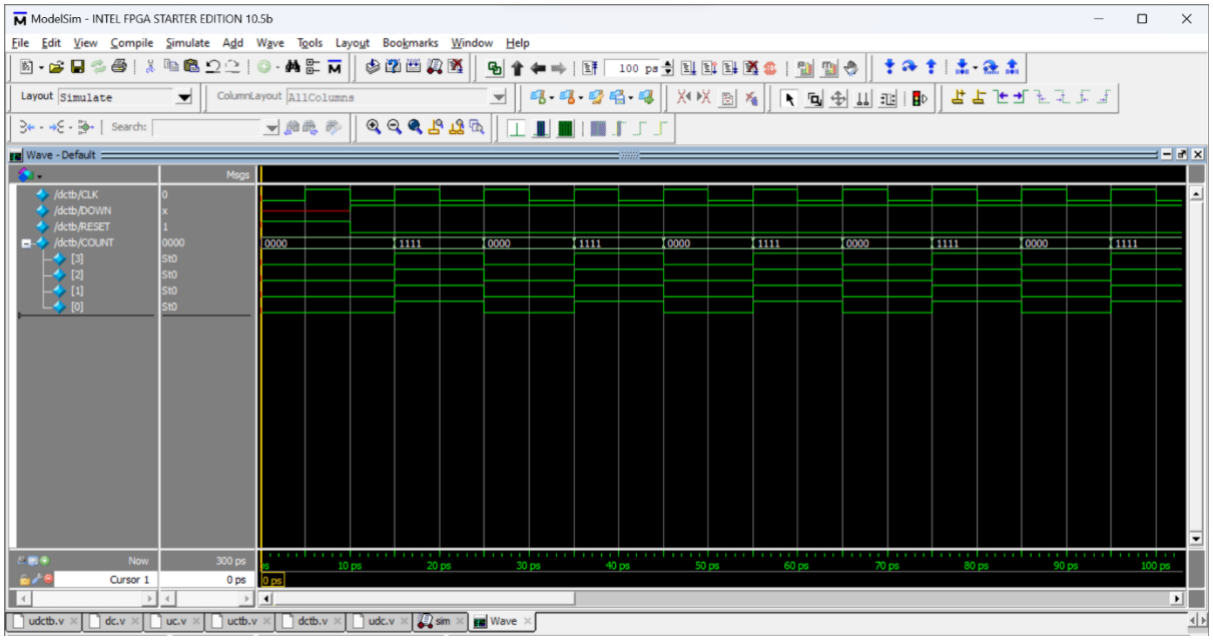
Up-Down Counter:



Up Counter:



Down Counter:



Result/ Inference:

The up-down, up and down counters were designed and verified using Verilog Programming.

Experiment No. 6 (Software) **Date: 06.12.2022**

Shift Registers

Aim:

To design shift registers using Verilog programming.

Software Required:

ModelSim

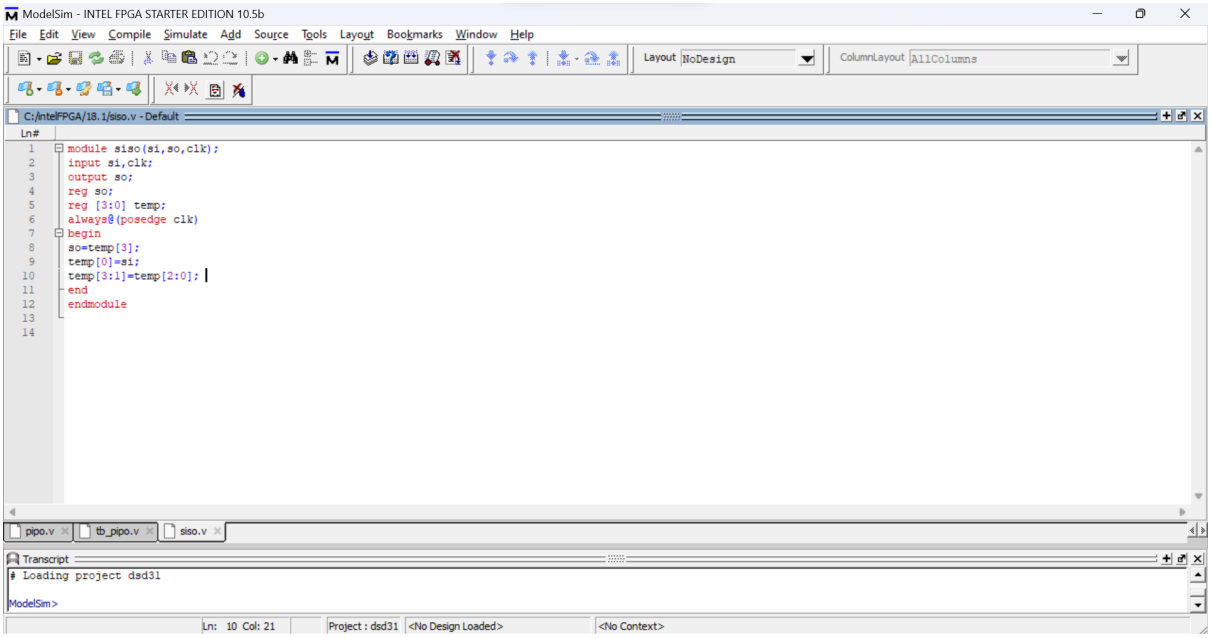
Procedure:

1. Open ModelSim.
2. Create New Project, then create a new file by selecting the verilog in the drop-down.
3. Type the code, then compile it.
4. Type the test bench, compile and simulate it.
5. Click add wave in the window and then run it.

Program Code:

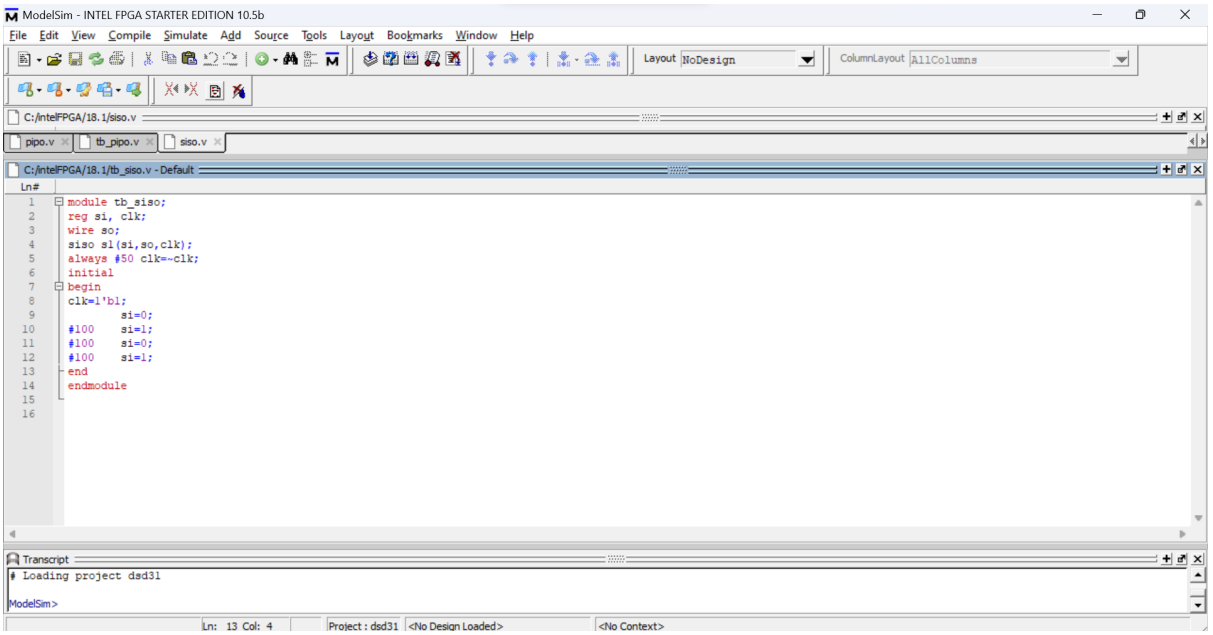
SISO:

```
module siso(si,so,clk);
input si,clk;
output so;
reg so;
reg [3:0] temp;
always@(posedge clk)
begin
so=temp[3];
temp[0]=si;
temp[3:1]=temp[2:0];
end
endmodule
```



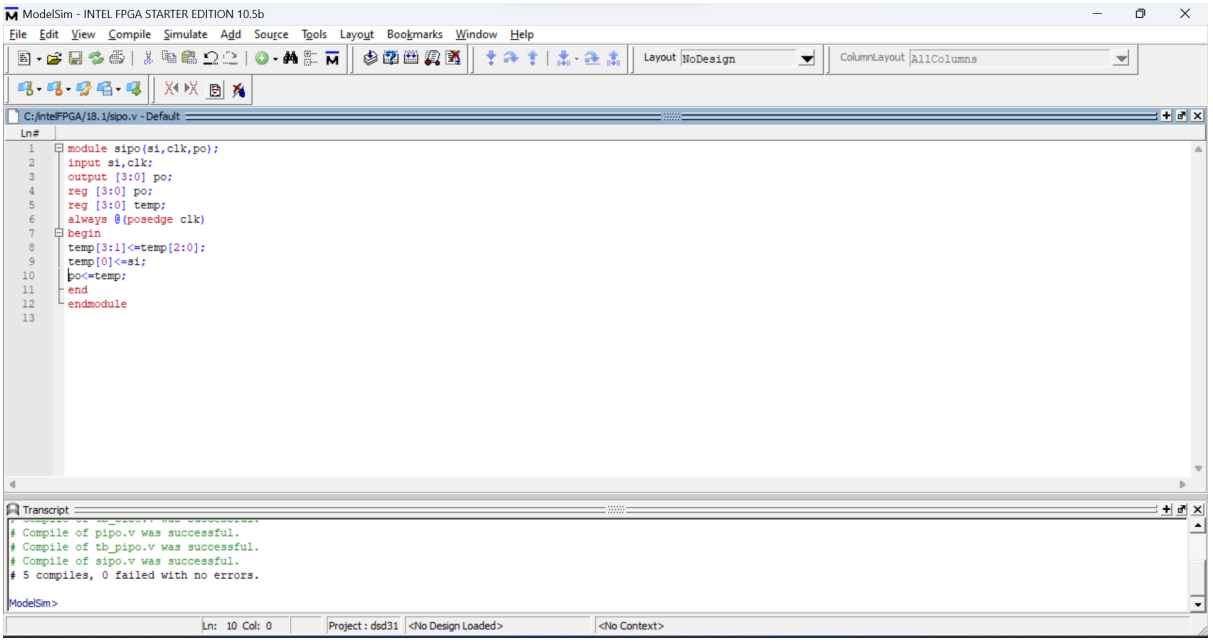
Test Bench for SISO:

```
module tb_siso;
reg si, clk;
wire so;
siso s1(si,so,clk);
always #50 clk=~clk;
initial
begin
clk=1'b1;
          si=0;
#100    si=1;
#100    si=0;
#100    si=1;
end
endmodule
```



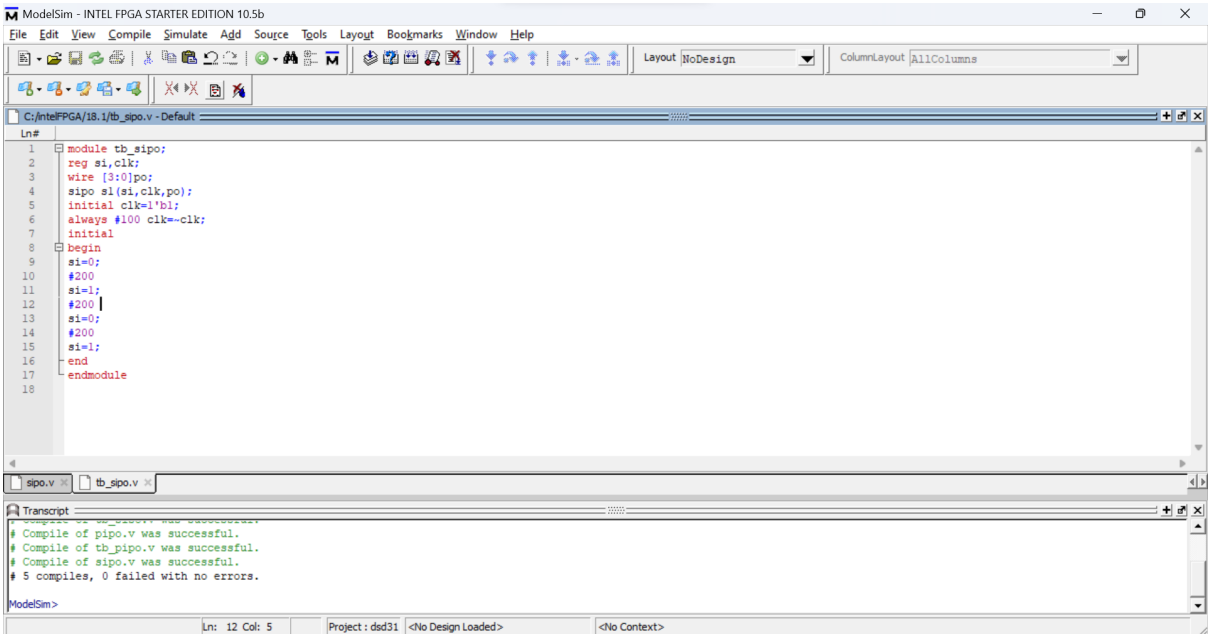
SIPO:

```
module sipo(si,clk,po);
input si,clk;
output [3:0] po;
reg [3:0] po;
reg [3:0] temp;
always @(posedge clk)
begin
temp[3:1]<=temp[2:0];
temp[0]<=si;
po<=temp;
end
endmodule
```



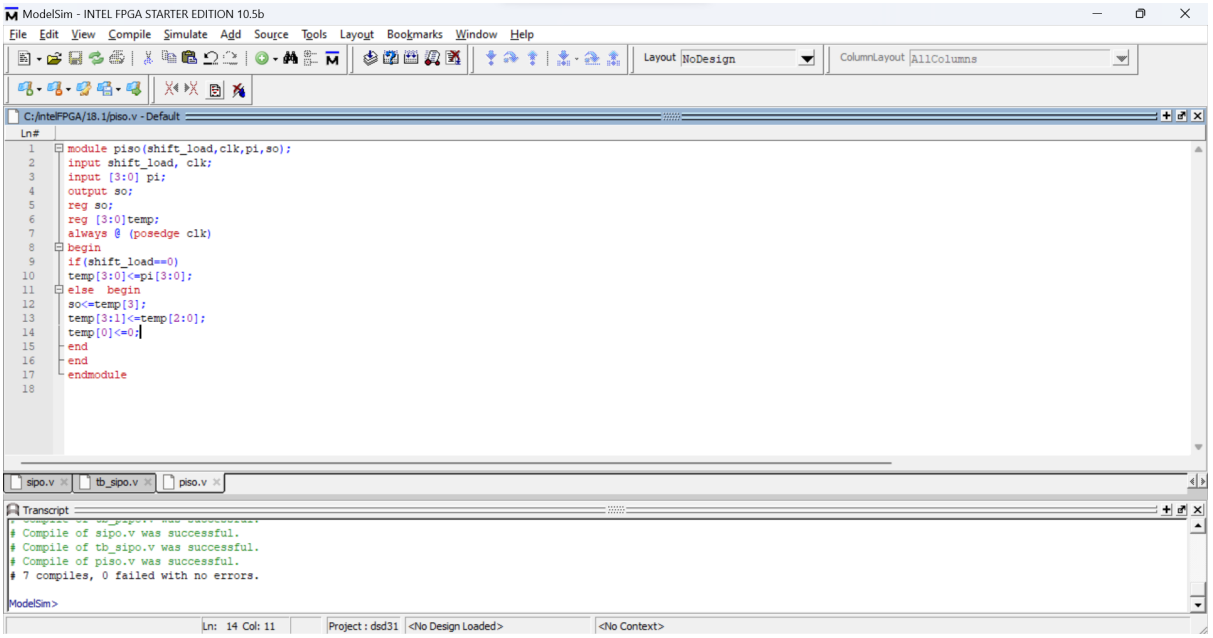
Test Bench for SIPO:

```
module tb_sipo;
reg si,clk;
wire [3:0]po;
sipo s1(si,clk,po);
initial clk=1'b1;
always #100 clk=~clk;
initial
begin
si=0;
#200
si=1;
#200
si=0;
#200
si=1;
end
endmodule
```



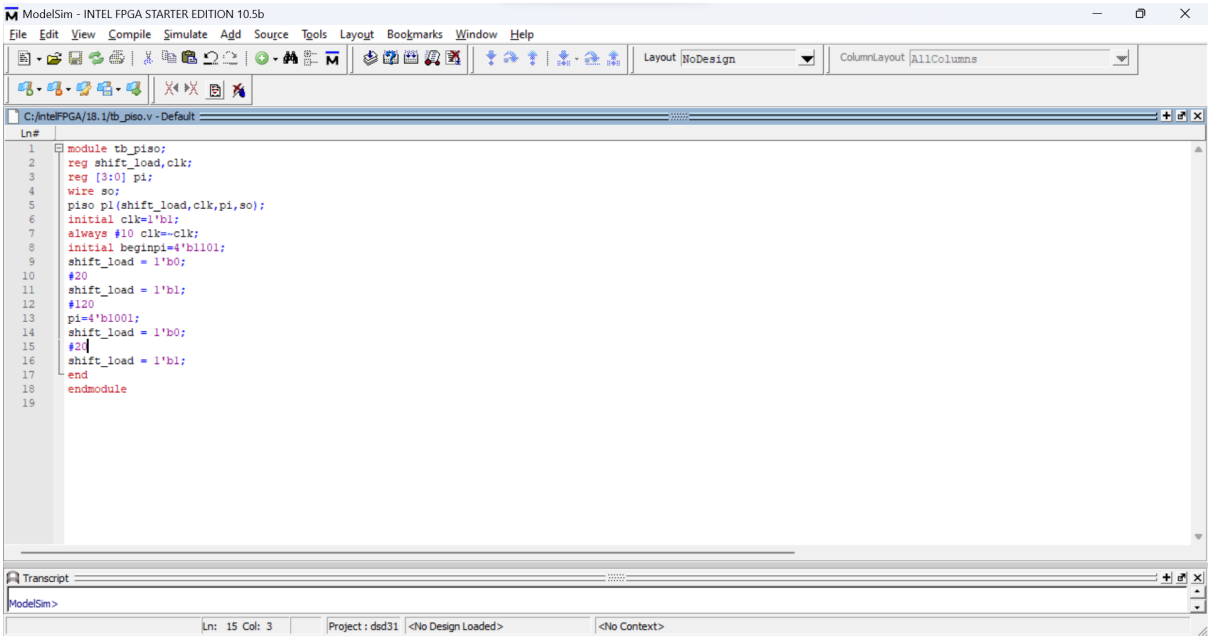
PISO:

```
module piso(shift_load,clk,pi,so);
input shift_load, clk;
input [3:0] pi;
output so;
reg so;
reg [3:0]temp;
always @ (posedge clk)
begin
if(shift_load==0)
temp[3:0]<=pi[3:0];
else  begin
so<=temp[3];
temp[3:1]<=temp[2:0];
temp[0]<=0;
end
end
endmodule
```



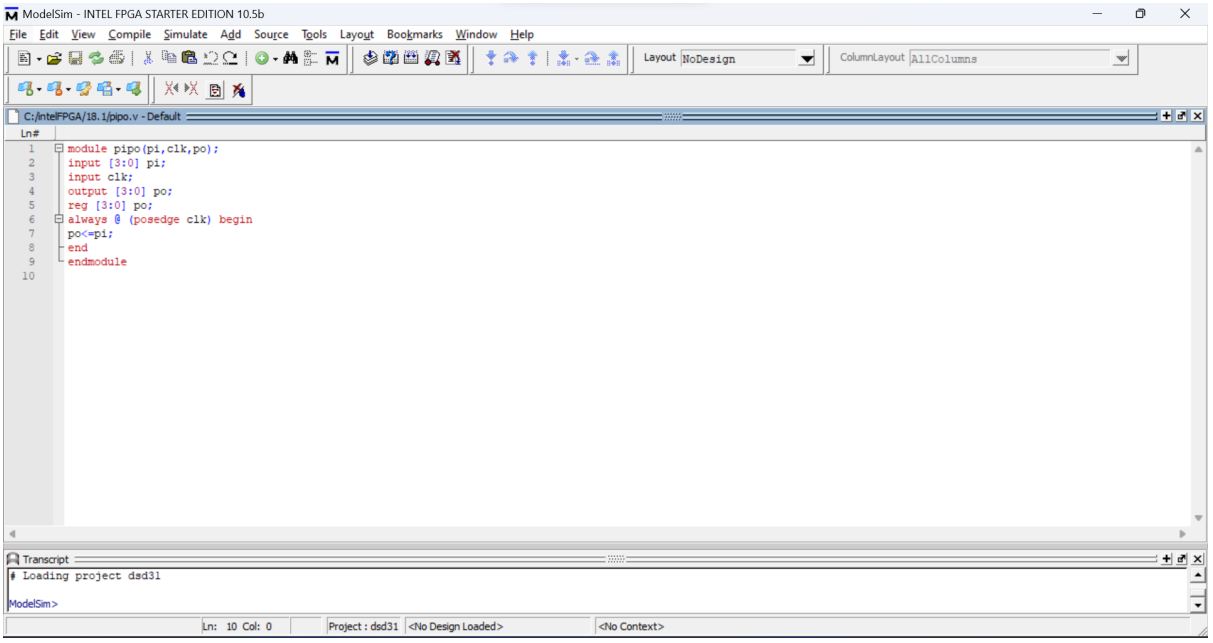
Test Bench for PISO:

```
module tb_piso;
reg shift_load,clk;
reg [3:0] pi;
wire so;
piso p1(shift_load,clk,pi,so);
initial clk=1'b1;
always #10 clk=~clk;
initial beginpi=4'b1101;
shift_load = 1'b0;
#20
shift_load = 1'b1;
#120
pi=4'b1001;
shift_load = 1'b0;
#20
shift_load = 1'b1;
end
endmodule
```



PIPO:

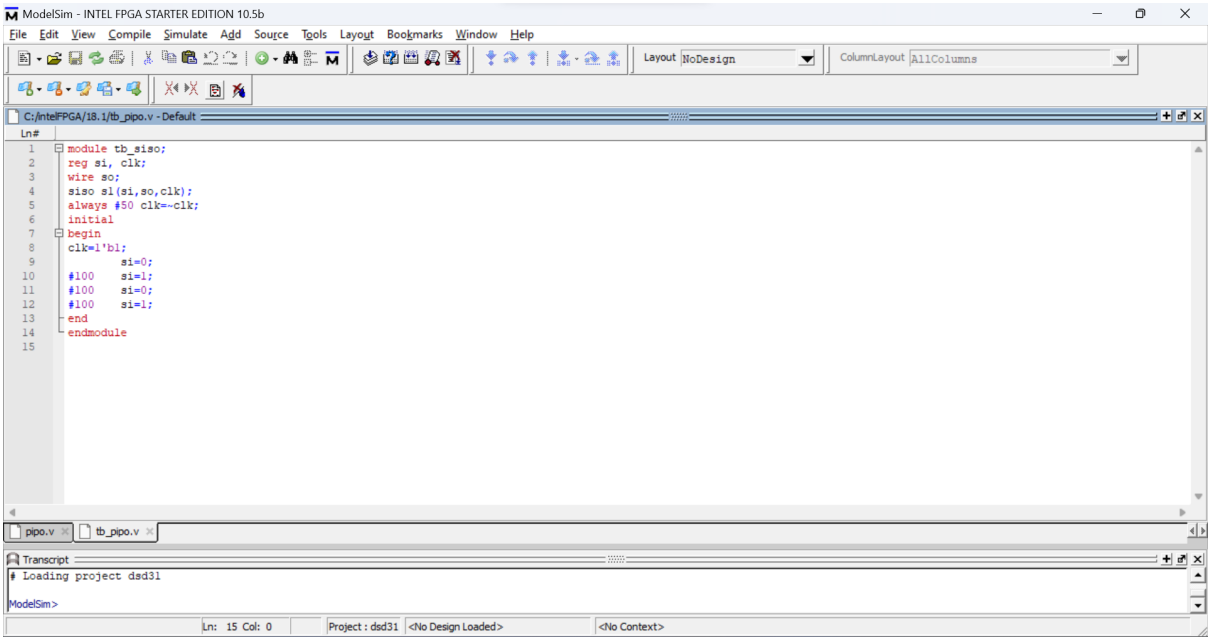
```
module pipo(pi,clk,po);
input [3:0] pi;
input clk;
output [3:0] po;
reg [3:0] po;
always @ (posedge clk) begin
po<=pi;
end
endmodule
```



Test Bench for PIPO:

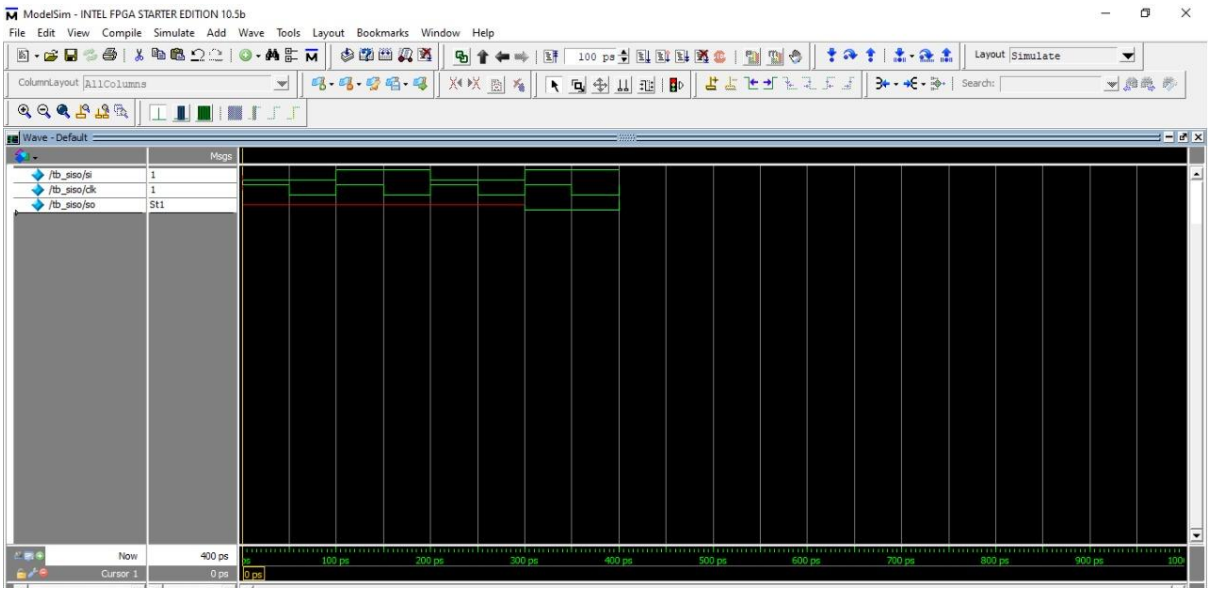
```
module tb_siso;
reg si, clk;
wire so;
```

```
siso s1(si,so,clk);
always #50 clk=~clk;
initial
begin
clk=1'b1;
        si=0;
#100    si=1;
#100    si=0;
#100    si=1;
end
endmodule
```

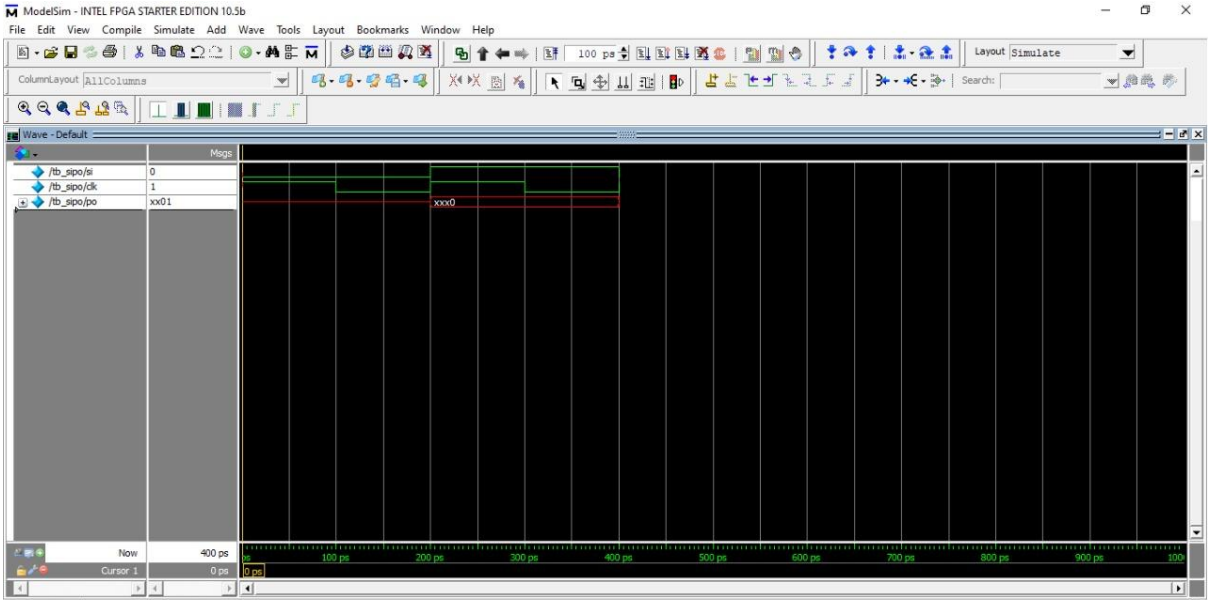


Output:

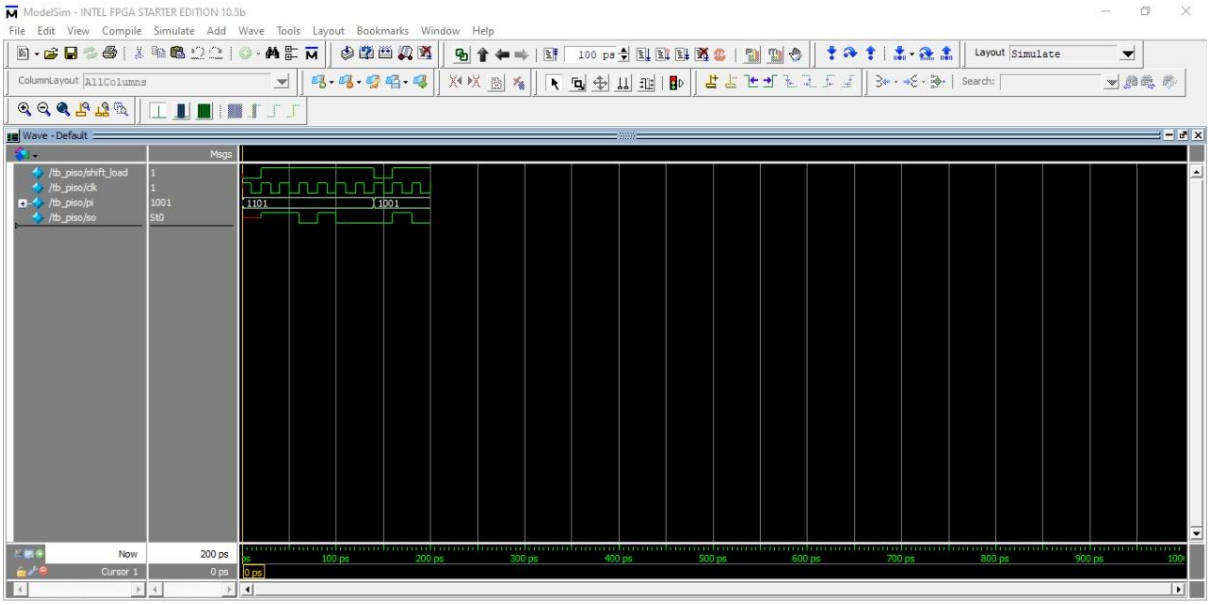
SISO:



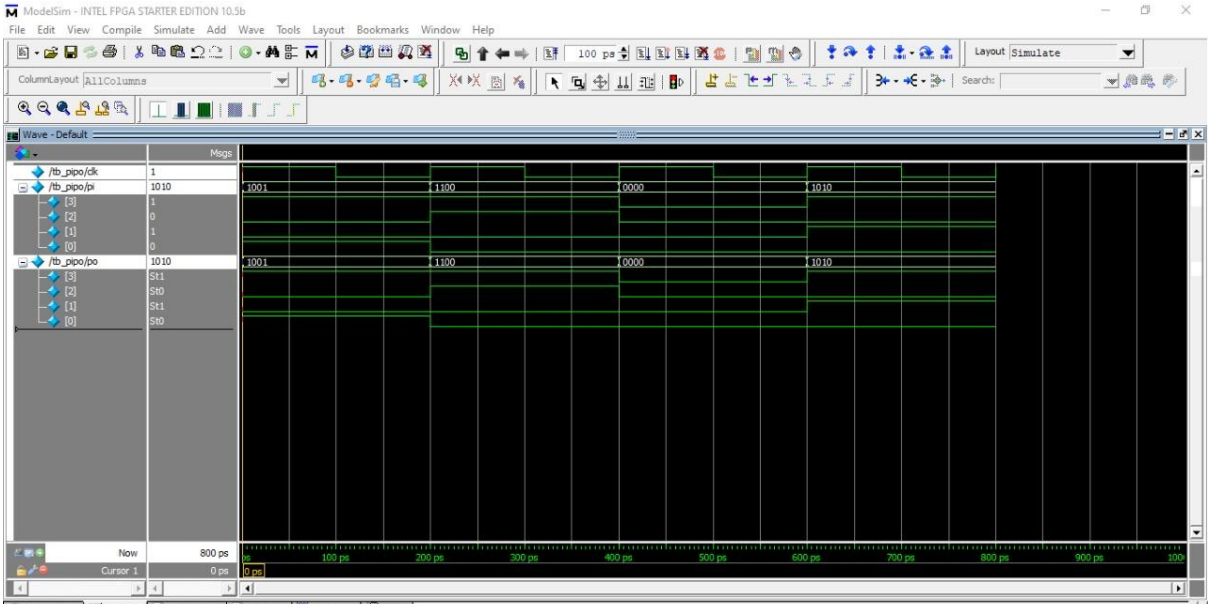
SIPO:


ModelSim - INTEL FPGA STARTER EDITION 10.5b
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
Msgs
/tb_sipo/si
/tb_sipo/clk
/tb_sipo/po
0
1
xx01
xxx0
Now
400 ps
Cursor 1
0 ps
100 ps
200 ps
300 ps
400 ps
500 ps
600 ps
700 ps
800 ps
900 ps

PISO:


ModelSim - INTEL FPGA STARTER EDITION 10.5b
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
Msgs
/tb_piso/shift_load
/tb_piso/clk
/tb_piso/pi
/tb_piso/so
1
1
1001
St0
1101
1001
Now
200 ps
Cursor 1
0 ps
100 ps
200 ps
300 ps
400 ps
500 ps
600 ps
700 ps
800 ps
900 ps

PIPO:



Result/ Inference:

From the given experiment, the shift register is verified using verilog programming.

Experiment No. 9 (Software) **Date: 09.10.2022**

Design of 4-bit Parallel Adder and 2-bit Multiplier

Aim: To implement Parallel Adder and 2-bit Multiplier using logic gates in Verilog Programming.

Software Required: ModelSim

Procedure:

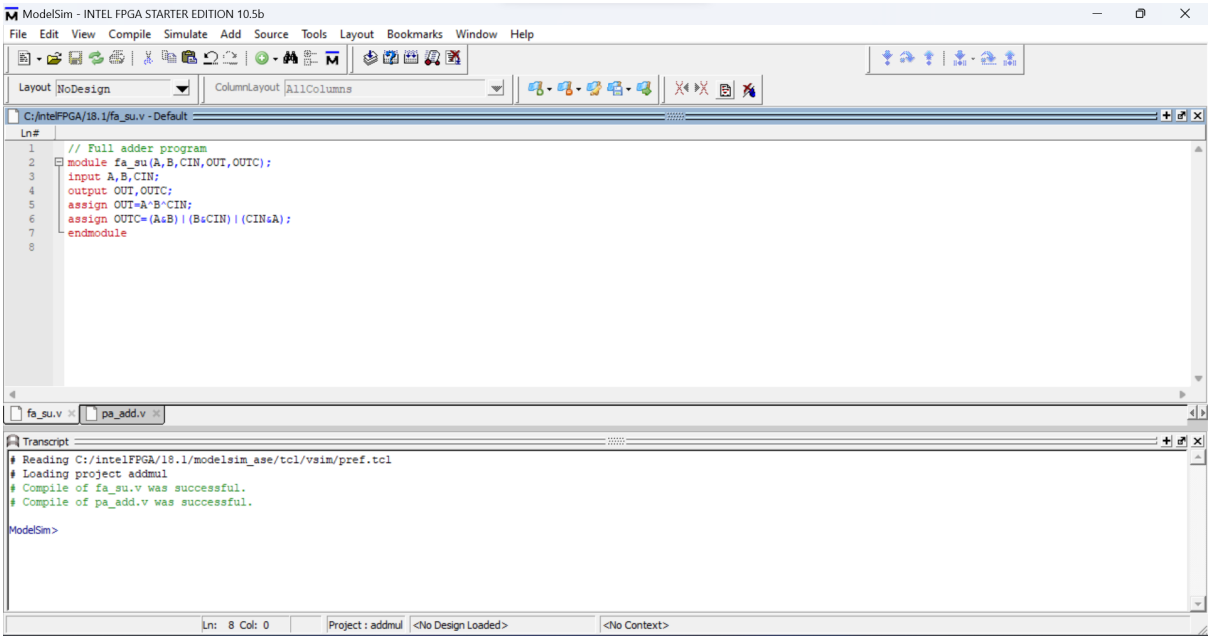
1. Open ModelSim.
2. Create New Project, then create a new file by selecting the verilog in the drop-down.
3. Type the code, then compile it.
4. Type the test bench, compile and simulate it.
5. Click add wave in the window and then run it.

Verilog Code:

Parallel Adder

Full Adder:

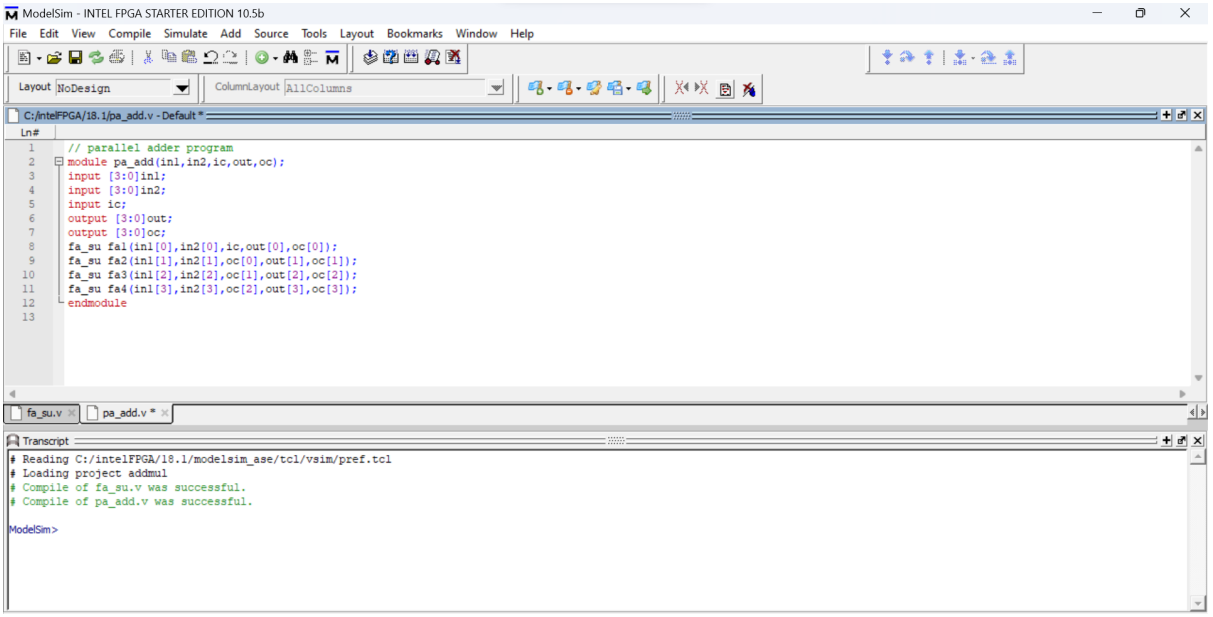
```
// Full Adder Program
module fa_su(A,B,CIN,OUT,OUTC);
input A,B,CIN;
output OUT,OUTC;
assign OUT=A^B^CIN;
assign OUTC=(A&B)|(B&CIN)|(CIN&A);
endmodule
```



Parallel Adder Program:

```
// parallel adder program
module pa_add(in1,in2,ic,out,oc);
input [3:0]in1;
input [3:0]in2;
```

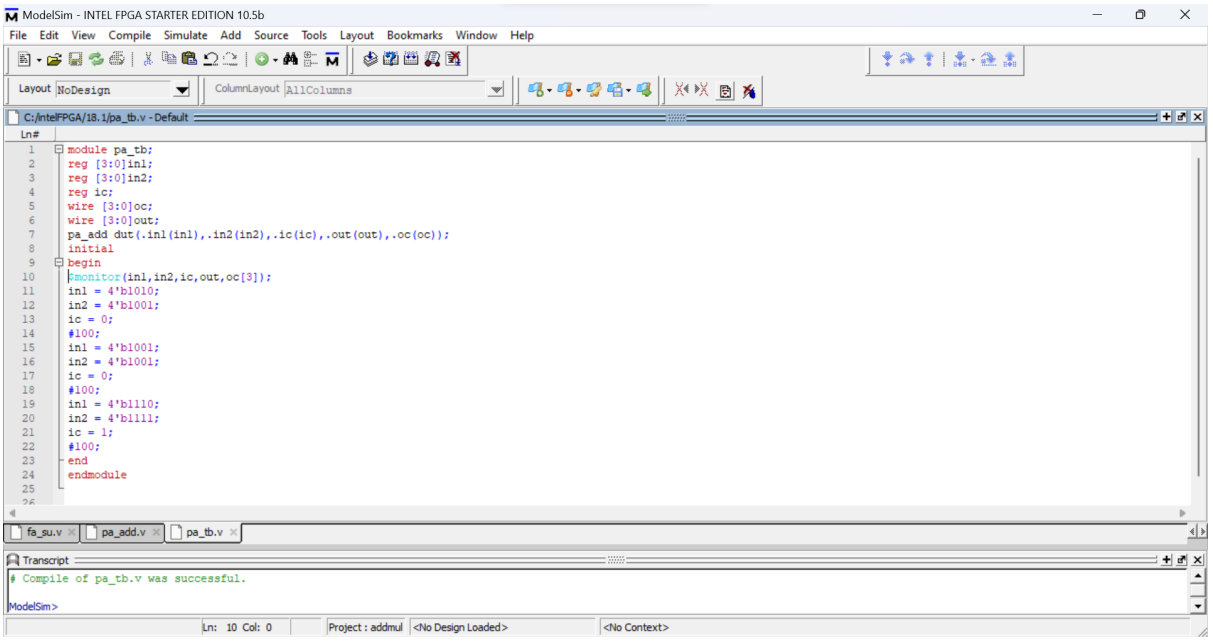
```
input ic;
output [3:0]out;
output [3:0]oc;
fa_su fa1(in1[0],in2[0],ic,out[0],oc[0]);
fa_su fa2(in1[1],in2[1],oc[0],out[1],oc[1]);
fa_su fa3(in1[2],in2[2],oc[1],out[2],oc[2]);
fa_su fa4(in1[3],in2[3],oc[2],out[3],oc[3]);
endmodule
```



Test Bench:

```
module pa_tb;
reg [3:0]in1;
reg [3:0]in2;
reg ic;
wire [3:0]oc;
wire [3:0]out;
pa_add dut(.in1(in1),.in2(in2),.ic(ic),.out(out),.oc(oc));
initial
begin
$monitor(in1,in2,ic,out,oc[3]);
in1 = 4'b1010;
in2 = 4'b1001;
ic = 0;
#100;
in1 = 4'b1001;
in2 = 4'b1001;
ic = 0;
#100;
in1 = 4'b1110;
in2 = 4'b1111;
ic = 1;
#100;
```

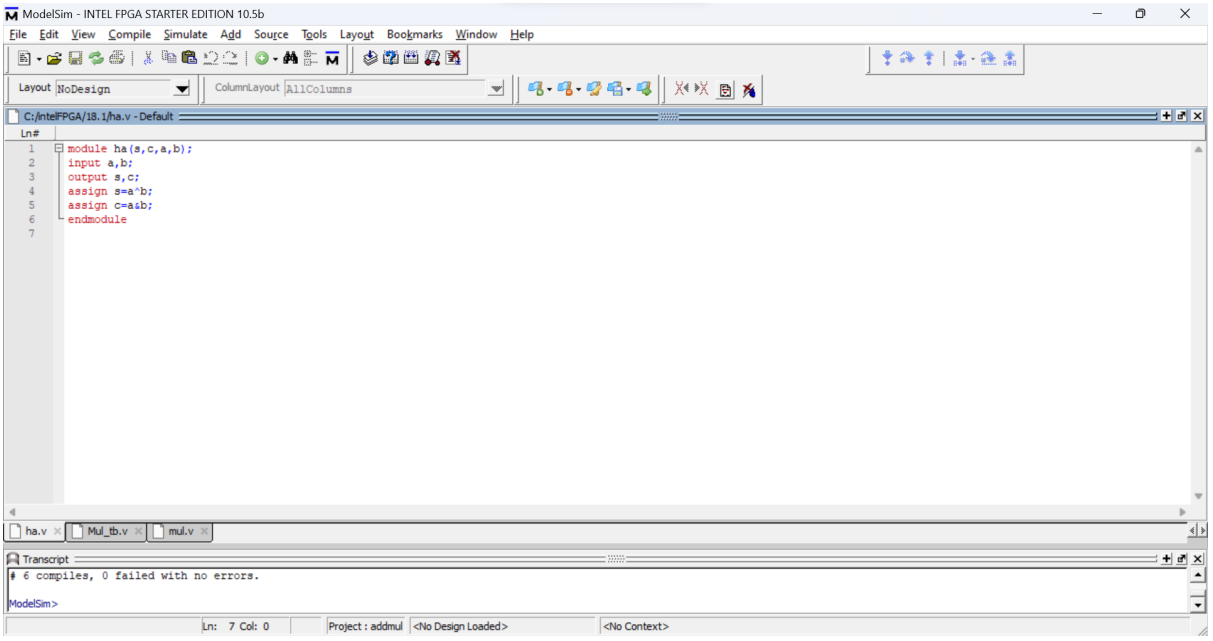
end
endmodule



2- bit Multiplier

Half Adder:

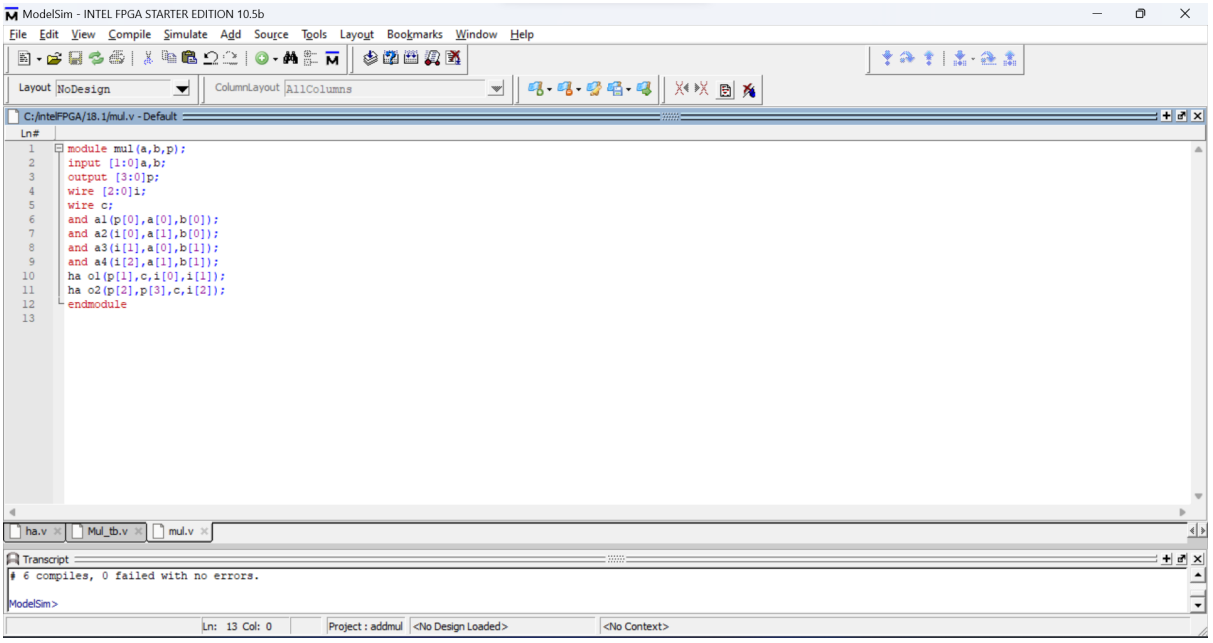
module ha(s,c,a,b);
input a,b;
output s,c;
assign s=a^b;
assign c=a&b;
endmodule



Multiplier:

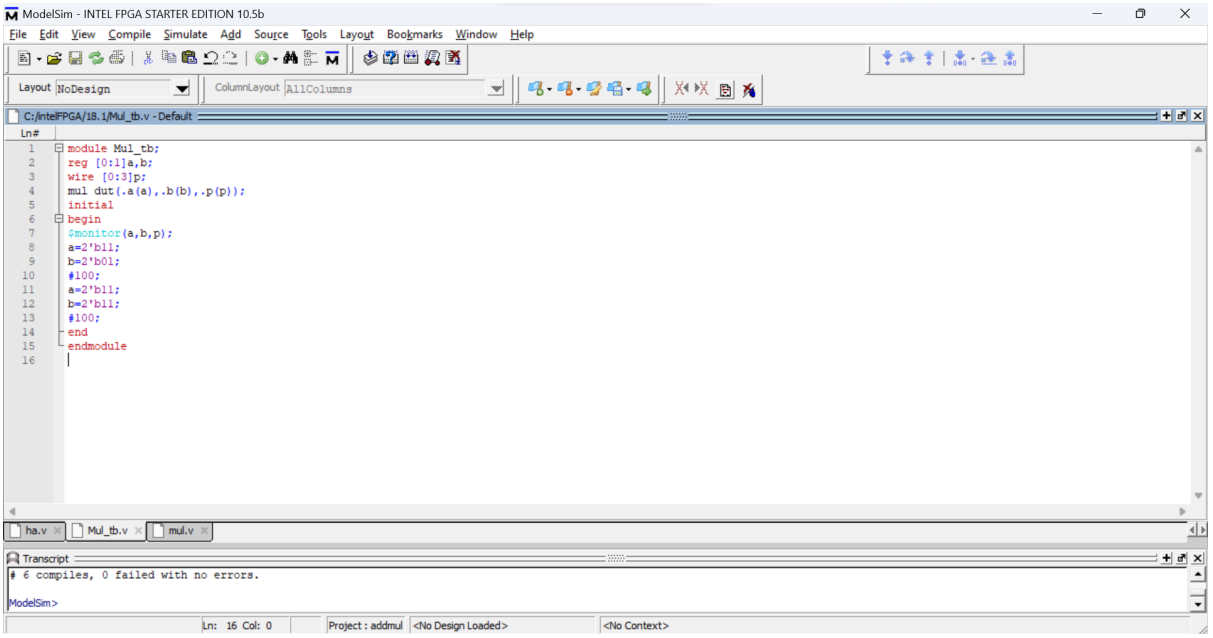
module mul(a,b,p);
input [1:0]a,b;
output [3:0]p;

```
wire [2:0]i;
wire c;
and a1(p[0],a[0],b[0]);
and a2(i[0],a[1],b[0]);
and a3(i[1],a[0],b[1]);
and a4(i[2],a[1],b[1]);
ha o1(p[1],c,i[0],i[1]);
ha o2(p[2],p[3],c,i[2]);
endmodule
```



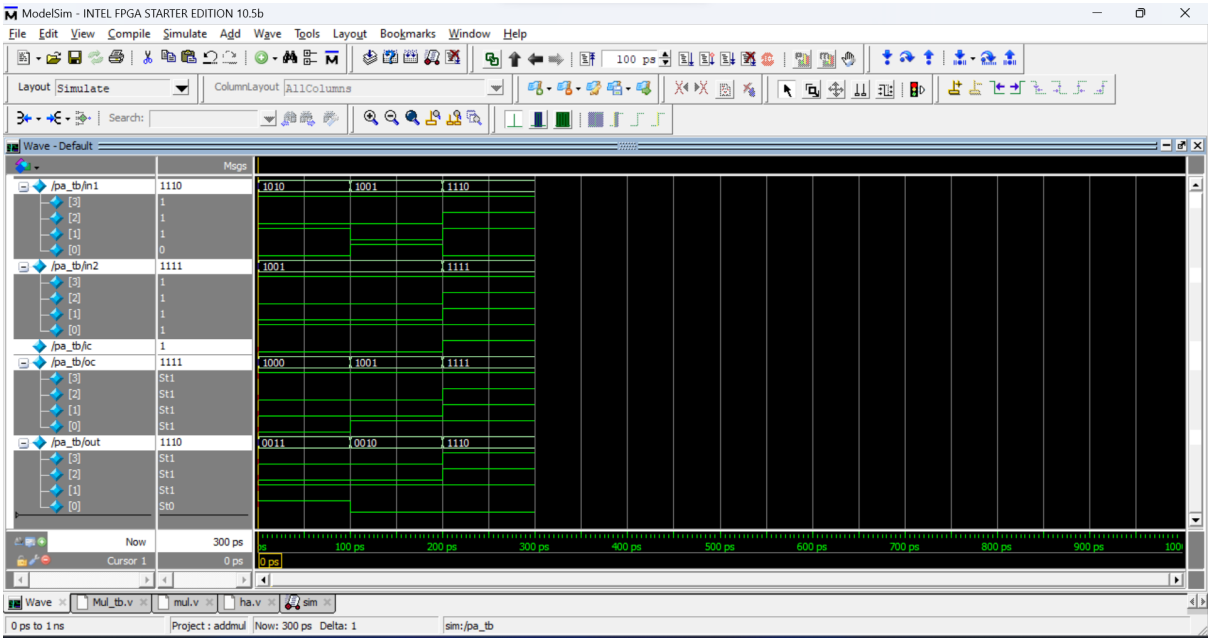
Test Bench:

```
module Mul_tb;
reg [0:1]a,b;
wire [0:3]p;
mul dut(.a(a),.b(b),.p(p));
initial
begin
$monitor(a,b,p);
a=2'b11;
b=2'b01;
#100;
a=2'b11;
b=2'b11;
#100;
end
endmodule
```

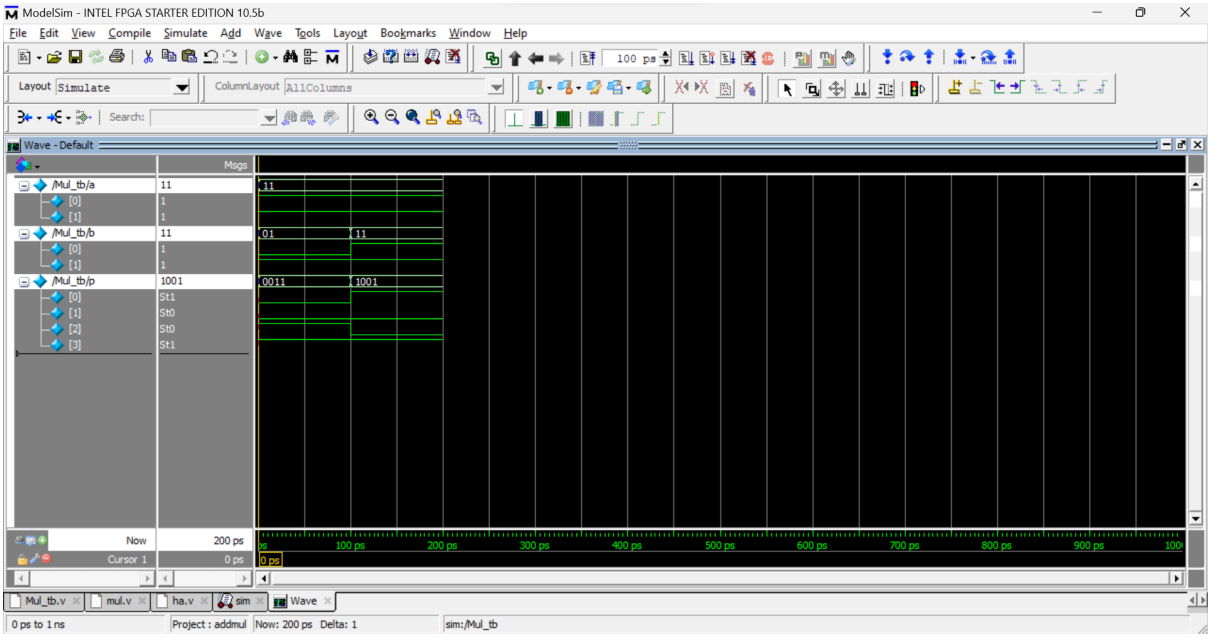


Output:

Parallel Adder:



Multiplier:



Result/ Inference:

From this experiment, the Parallel adder and multiplier circuits are verified using data-flow modelling and behavioural modelling using verilog programming.

Digital System Design Lab

Virtual Lab – 1

Aim: To design and implement a 2 bit Comparator using logic gates in Verilog.

Software Required: ModelSim

Procedure:

1. Open ModelSim, Create new project, new file and type the name.
2. Select Verilog in that drop-down menu.
3. Type the code, compile it.
4. Type the test bench for it and compile and simulate it.
5. Click Add Wave and Create the waveform.
6. Repeat this steps for different modelling.

Truth Table:

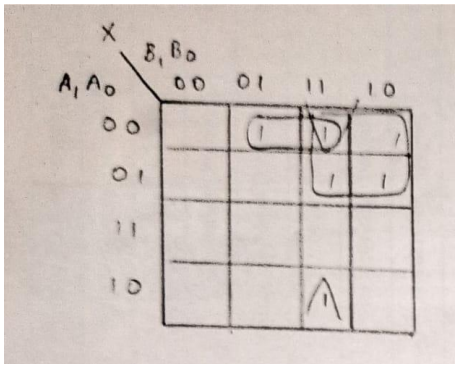
The truth table is given below.

Inputs				X	Y	Z
A1(MSB)	A0	B1(MSB)	B0	A<B	A>B	A=B
0	0	0	0	0	0	1
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	1	0
0	1	0	1	0	0	1
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	0	1	1	1	0	0
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	0
1	1	1	1	0	0	1

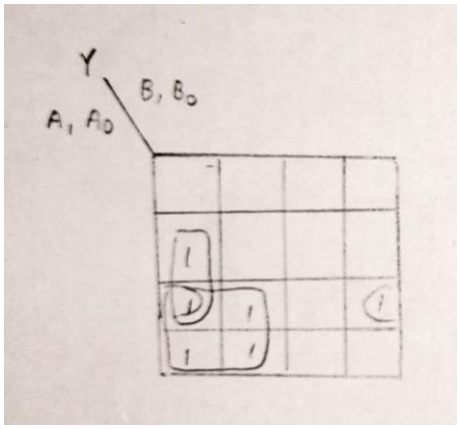
Two 2 bit numbers A1,A0 and B1,B0 are compared with each other. X, Y and Z are outputs.

X=1 when A<B and 0 otherwise; Y=1 when A>B and 0 otherwise; and Z=1 when A=B and 0 otherwise.

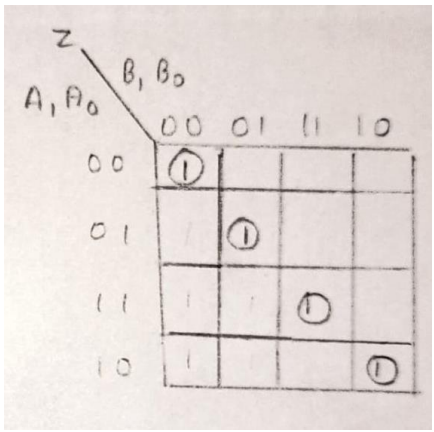
K- Map:



X = A1'A0'B0 + A0'B1B0 + A1'B1

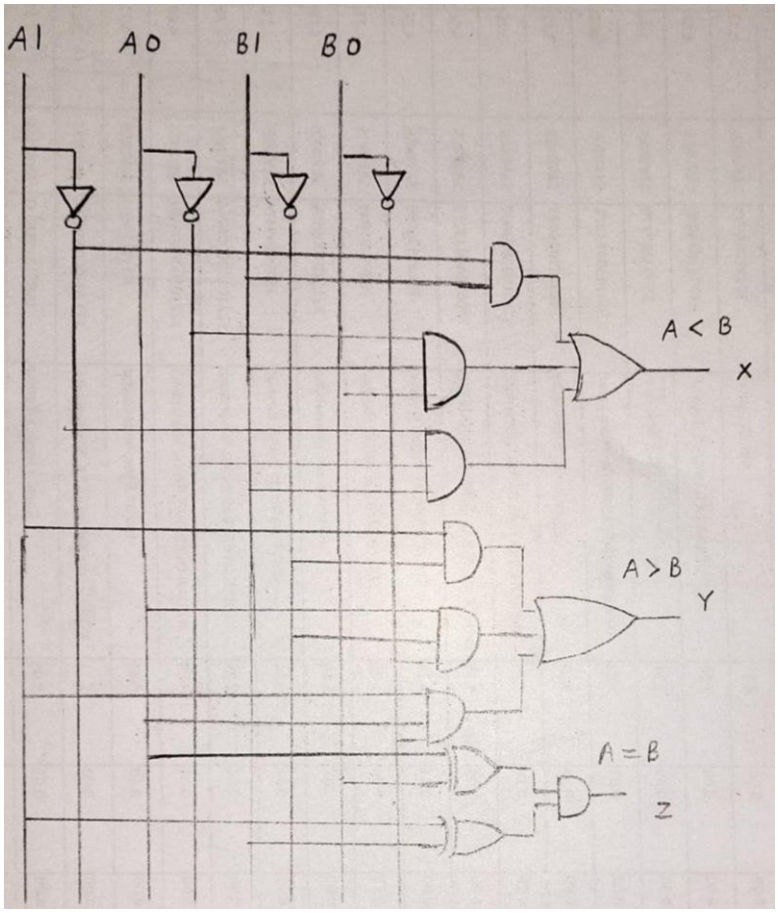


Y = A0B1'B0' + A1A0B0' + A1B1'



Z = (A0 ⊙ B0)(A1 ⊙ B1)

Logic Diagram:



Verilog Code:

Gate Modelling:

```
module vlgm (A1,A0,B1,B0,X,Y,Z);

input A1,A0,B1,B0;

output X,Y,Z;

wire i1,i2,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12;

not a1 (i1,A0);

not a2 (i2,A1);

not a3 (i3,B0);

not a4 (i4,B1);

and a5 (i5,i1,i2,B0);

and a6 (i6,i1,B0,B1);

and a7 (i7,i2,B1);

or a8 (X,i5,i6,i7);
```

and a9 (i8,A0,A1,i3);

and a10 (i9,A0,i3,i4);

and a11 (i10,A1,i4);

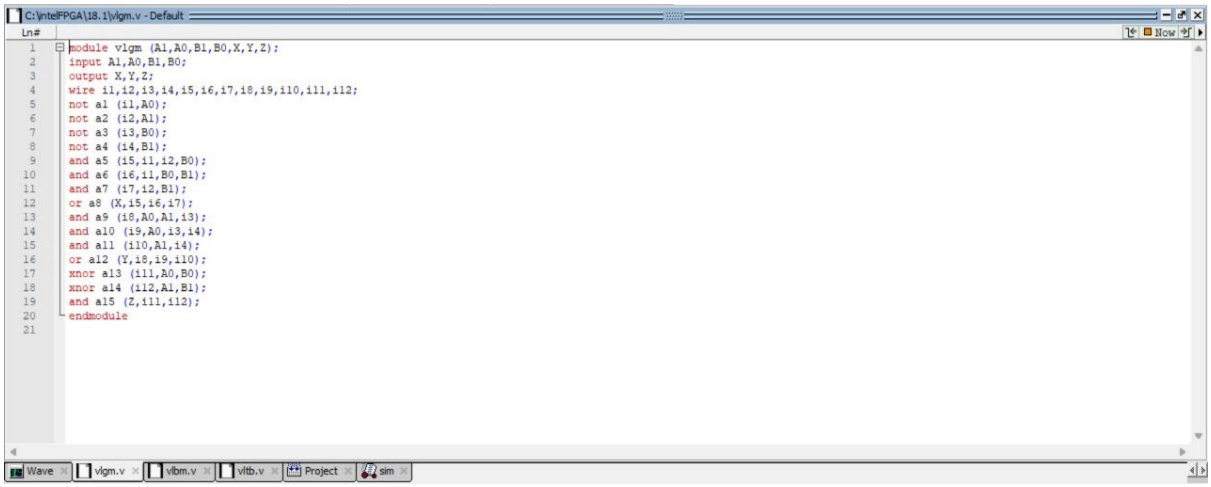
or a12 (Y,i8,i9,i10);

xnor a13 (i11,A0,B0);

xnor a14 (i12,A1,B1);

and a15 (Z,i11,i12);

endmodule



Data Flow Modelling:

module vldfm (A1,A0,B1,B0,X,Y,Z);

input A1,A0,B1,B0;

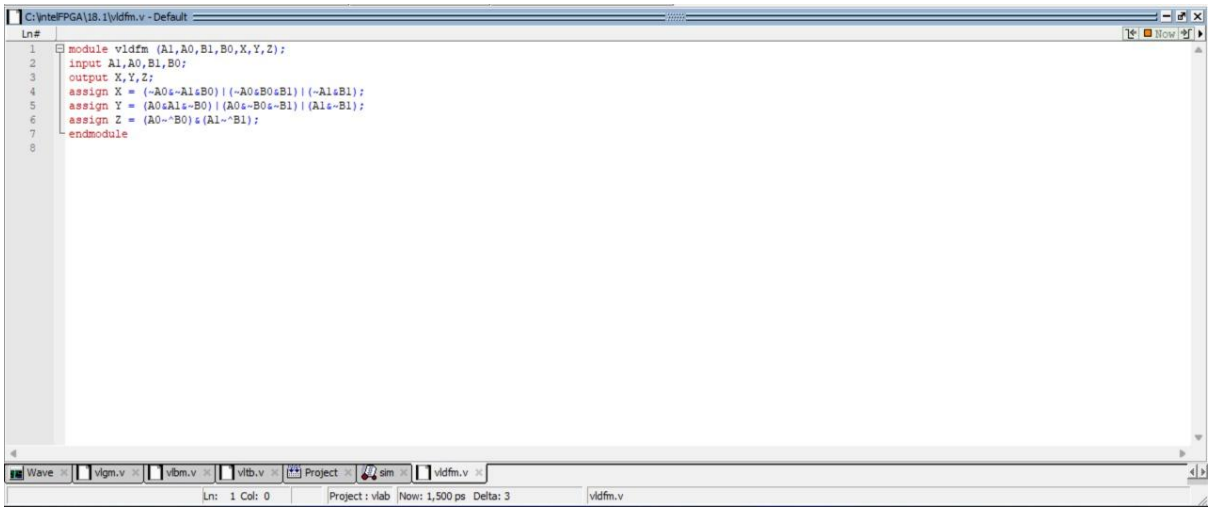
output X,Y,Z;

assign X = (~A0&~A1&B0)|(~A0&B0&B1)|(~A1&B1);

assign Y = (A0&A1&~B0)|(A0&~B0&~B1)|(A1&~B1);

assign Z = (A0~^B0)&(A1~^B1);

endmodule



Behavioural Modelling:

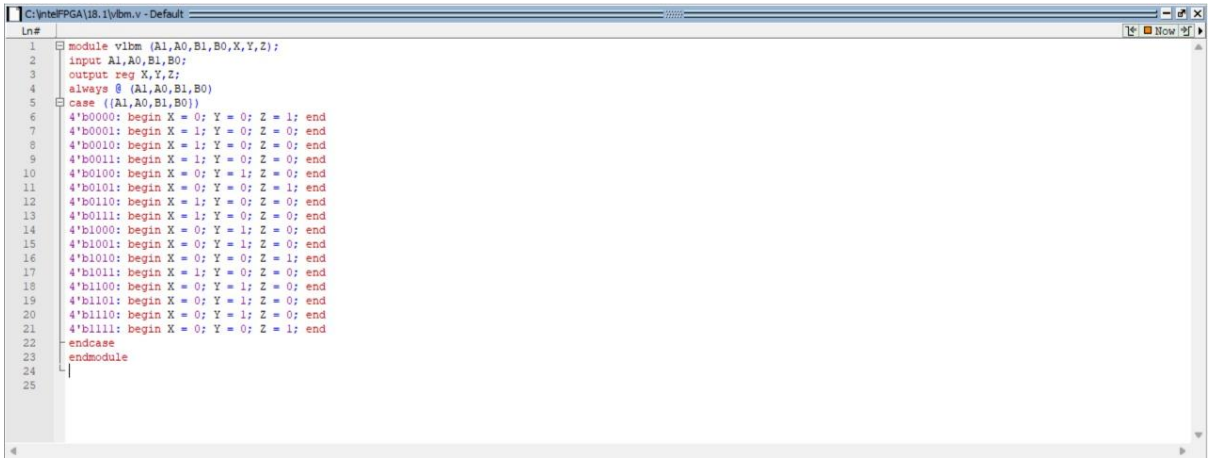
```
module vlbm (A1,A0,B1,B0,X,Y,Z);

input A1,A0,B1,B0;

output reg X,Y,Z;

always @ (A1,A0,B1,B0)

case ({A1,A0,B1,B0})

4'b0000: begin X = 0; Y = 0; Z = 1; end

4'b0001: begin X = 1; Y = 0; Z = 0; end

4'b0010: begin X = 1; Y = 0; Z = 0; end

4'b0011: begin X = 1; Y = 0; Z = 0; end

4'b0100: begin X = 0; Y = 1; Z = 0; end

4'b0101: begin X = 0; Y = 0; Z = 1; end

4'b0110: begin X = 1; Y = 0; Z = 0; end

4'b0111: begin X = 1; Y = 0; Z = 0; end

4'b1000: begin X = 0; Y = 1; Z = 0; end

4'b1001: begin X = 0; Y = 1; Z = 0; end

4'b1010: begin X = 0; Y = 0; Z = 1; end

4'b1011: begin X = 1; Y = 0; Z = 0; end

4'b1100: begin X = 0; Y = 1; Z = 0; end

4'b1101: begin X = 0; Y = 1; Z = 0; end
```

4'b1110: begin X = 0; Y = 1; Z = 0; end

4'b1111: begin X = 0; Y = 0; Z = 1; end

endcase

endmodule



Test Bench:

module vltb;

reg A1,A0,B1,B0;

wire X,Y,Z;

vlgm dut (.A1(A1),.A0(A0),.B1(B1),.B0(B0),.X(X),.Y(Y),.Z(Z));

initial begin

$monitor (A1,A0,B1,B0,X,Y,Z);

A1 = 1'b0;

A0 = 1'b0;

B1 = 1'b0;

B0 = 1'b0;

#100

A1 = 1'b0;

A0 = 1'b0;

B1 = 1'b0;

B0 = 1'b1;

```
#100
A1 = 1'b0;
A0 = 1'b0;
B1 = 1'b1;
B0 = 1'b0;
#100
A1 = 1'b0;
A0 = 1'b0;
B1 = 1'b1;
B0 = 1'b1;
#100
A1 = 1'b0;
A0 = 1'b1;
B1 = 1'b0;
B0 = 1'b0;
#100
A1 = 1'b0;
A0 = 1'b1;
B1 = 1'b0;
B0 = 1'b1;
#100
A1 = 1'b0;
A0 = 1'b1;
B1 = 1'b1;
B0 = 1'b0;
#100
A1 = 1'b0;
```

```
A0 = 1'b1;

B1 = 1'b1;

B0 = 1'b1;

#100

A1 = 1'b1;

A0 = 1'b0;

B1 = 1'b0;

B0 = 1'b0;

#100

A1 = 1'b1;

A0 = 1'b0;

B1 = 1'b0;

B0 = 1'b1;

#100

A1 = 1'b1;

A0 = 1'b0;

B1 = 1'b1;

B0 = 1'b0;

#100

A1 = 1'b1;

A0 = 1'b0;

B1 = 1'b1;

B0 = 1'b1;

#100

A1 = 1'b1;

A0 = 1'b1;

B1 = 1'b0;
```

B0 = 1'b0;

#100

A1 = 1'b1;

A0 = 1'b1;

B1 = 1'b0;

B0 = 1'b1;

#100A

A1 = 1'b1;

A0 = 1'b1;

B1 = 1'b0;

B0 = 1'b1;

#100

A1 = 1'b1;

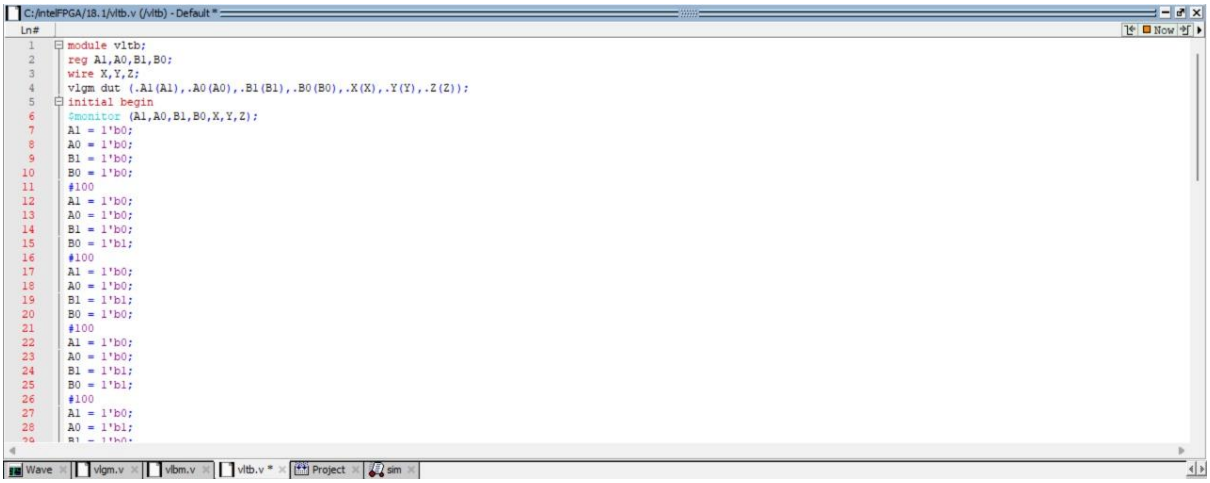
A0 = 1'b1;

B1 = 1'b1;

B0 = 1'b1;

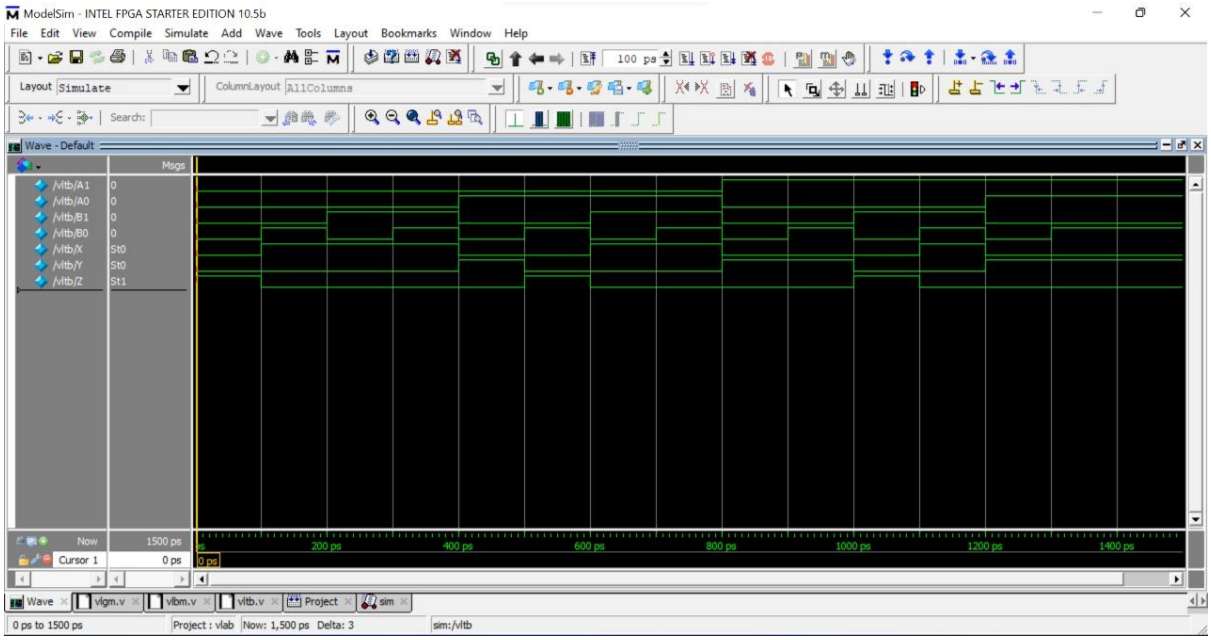
end

endmodule

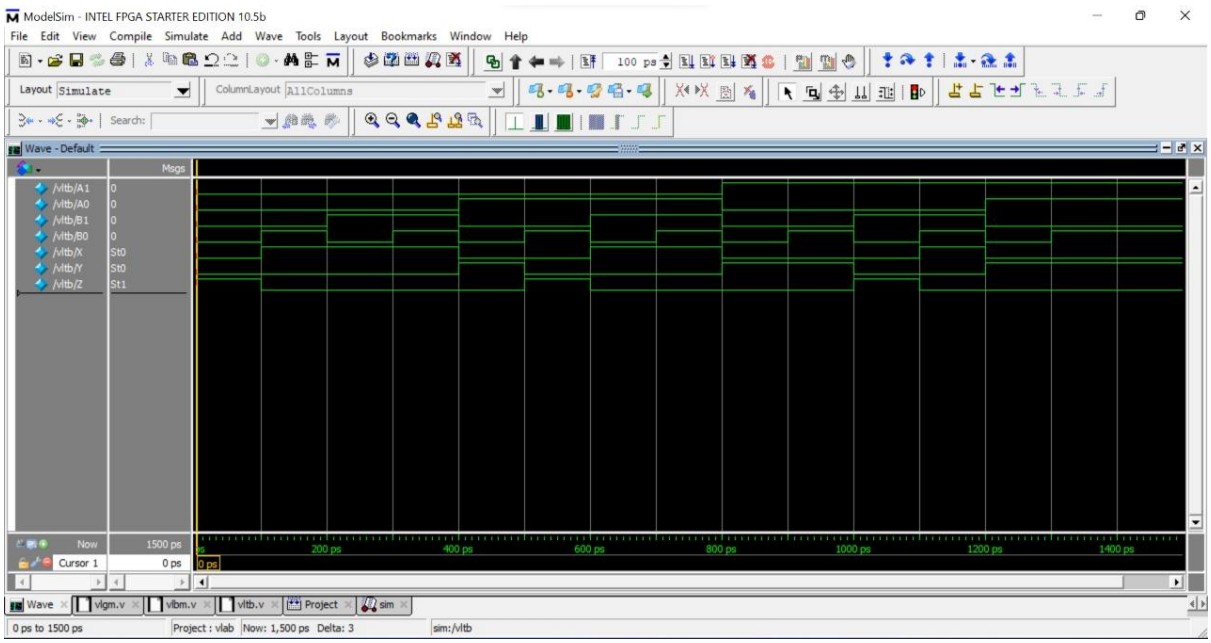


Output:

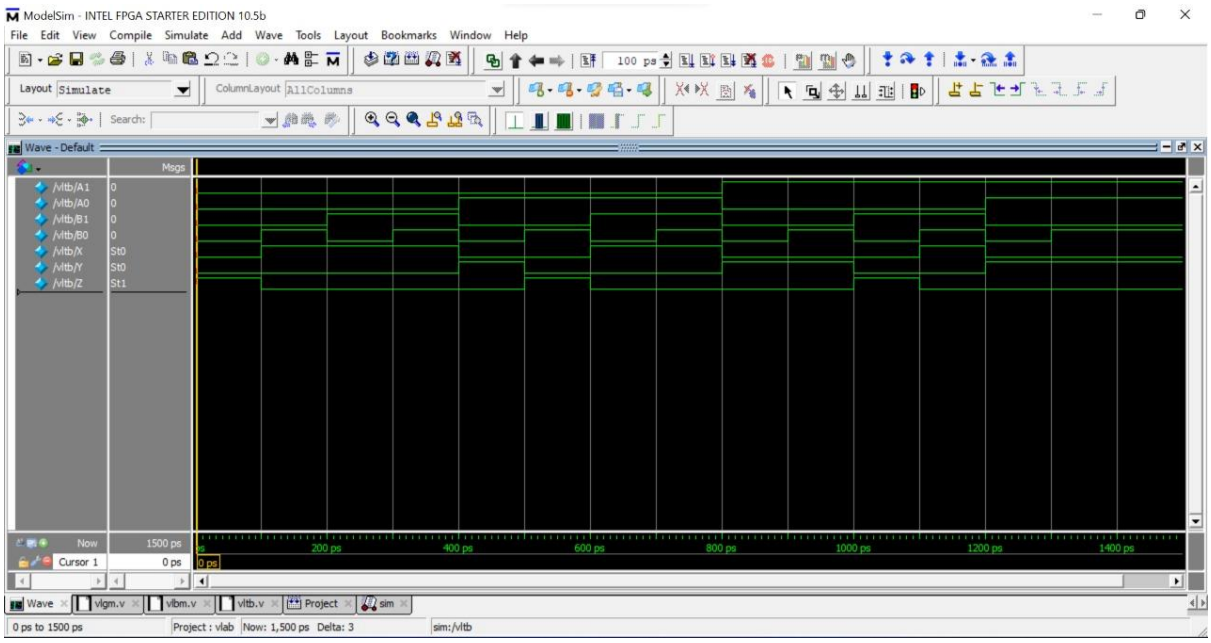
Gate Modelling:



Data Flow Modelling:



Behavioural Modelling:



Result:

A 2 bit Comparator is designed and implemented using logic gates in Verilog.