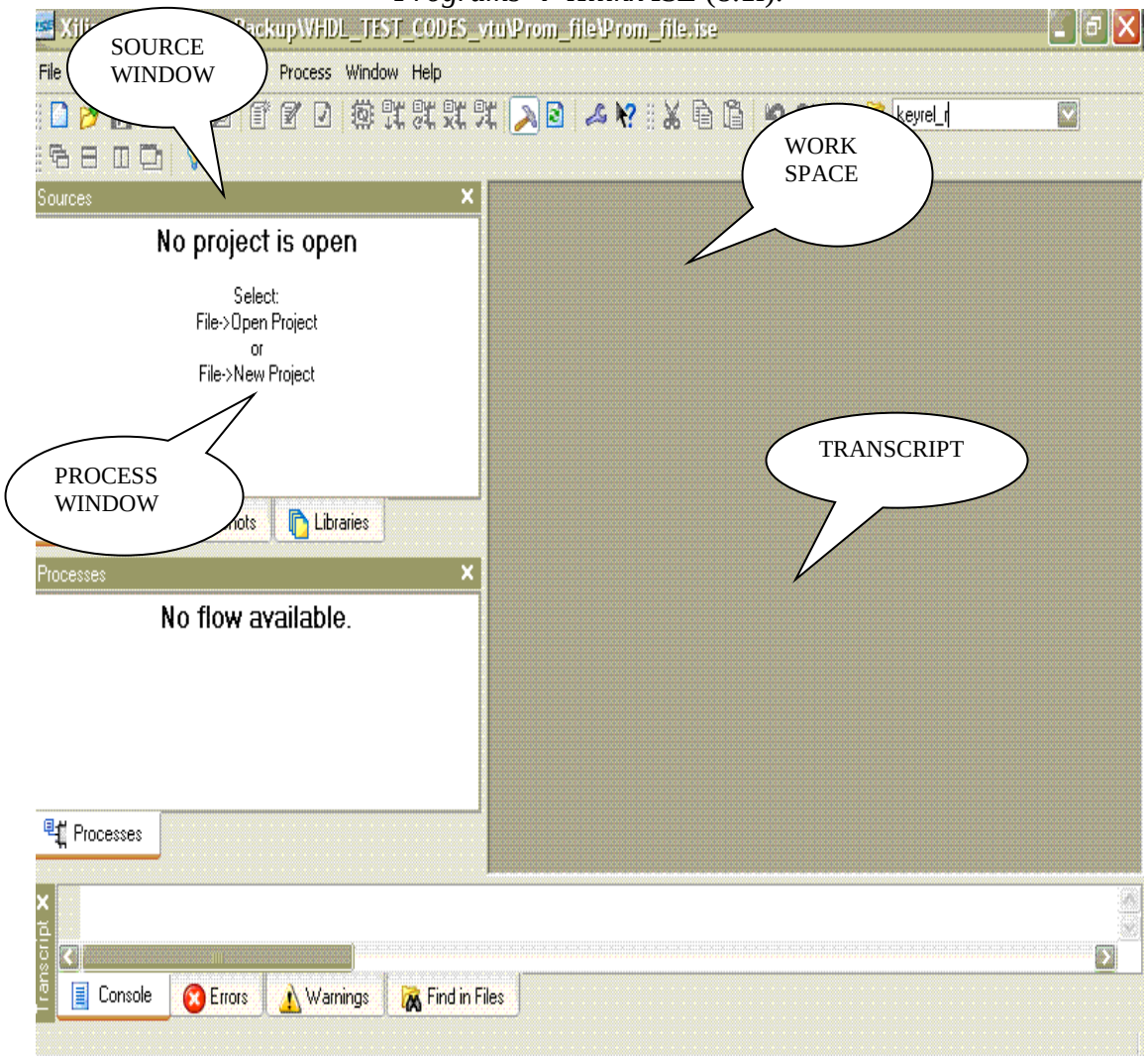
CHENDU COLLEGE OF ENGINEERING & TECHNOLOGY

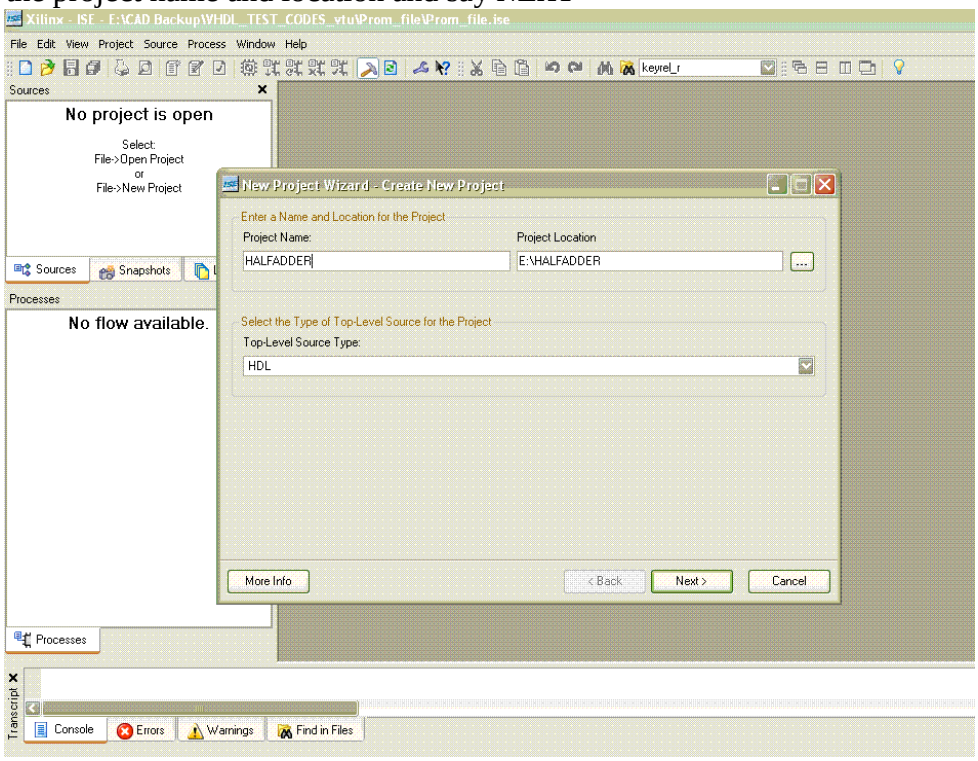
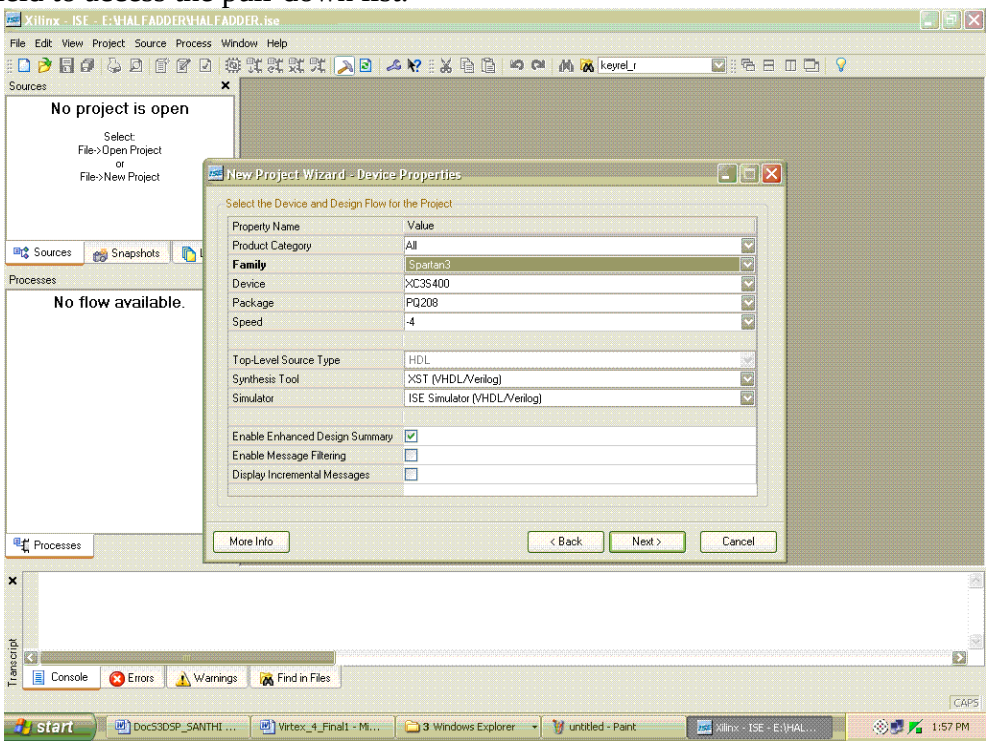
(Approved by AICTE, New Delhi and Affiliated to Anna University)
Zamin Endathur, Madurantakam, Kancheepuram District – 603311.

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

VLSI LAB MANUAL

VI SEMESTER ECE

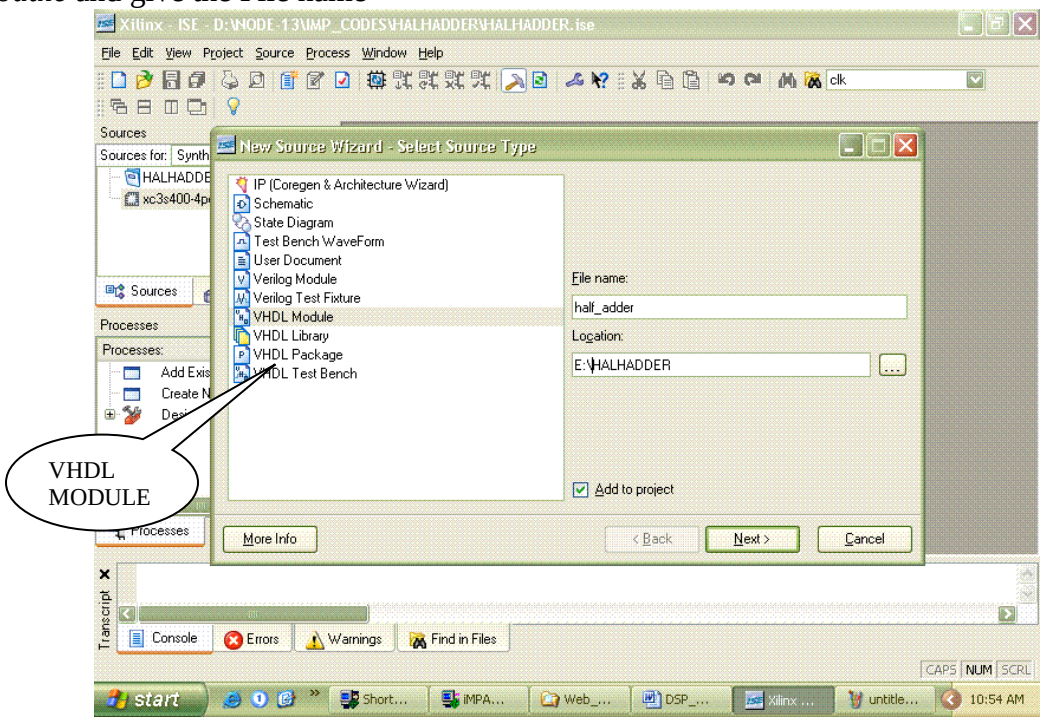
Step 1 :	Start the Xilinx Project Navigator by using the desktop shortcut or by using the *Start* → *Programs* → *Xilinx ISE* (8.1i). 

Step 2	*Create a new project* In the window go to *FILE →New project*. Specify the project name and location and say *NEXT*  *Select Device*. Use the pull-down arrow to select the Value for each Property Name. Click in the field to access the pull-down list.  Say *FINISH*. Project summary is seen.

Step 3:

Creating a new VHD file

Click on the symbol of FPGA device and then right click→ Click on *new source* →*VHDL module* and give the File name

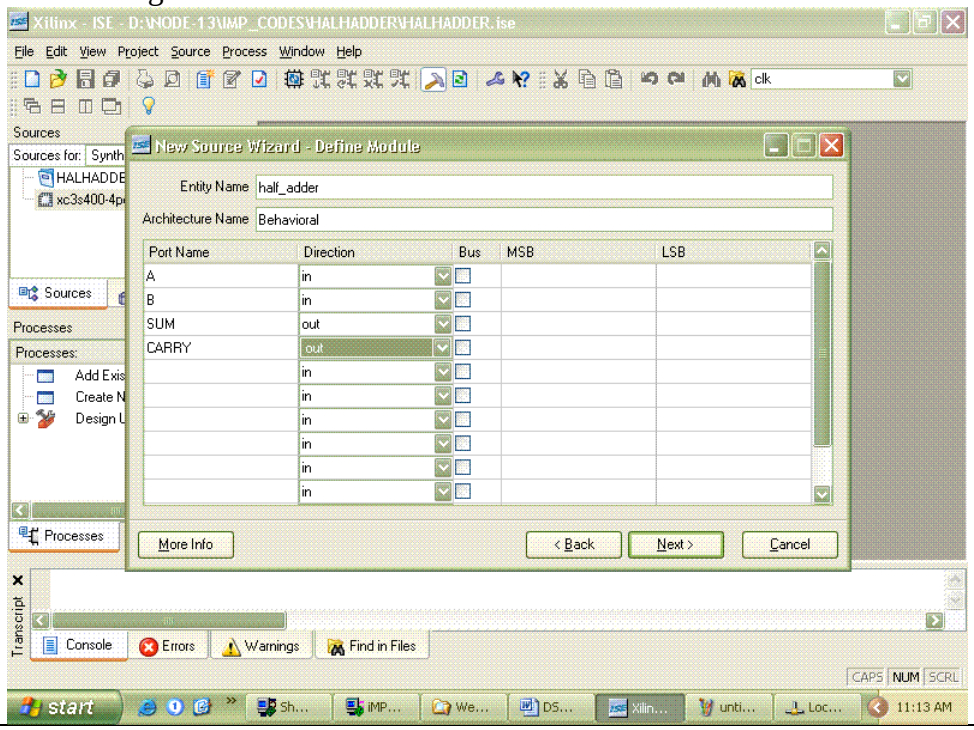


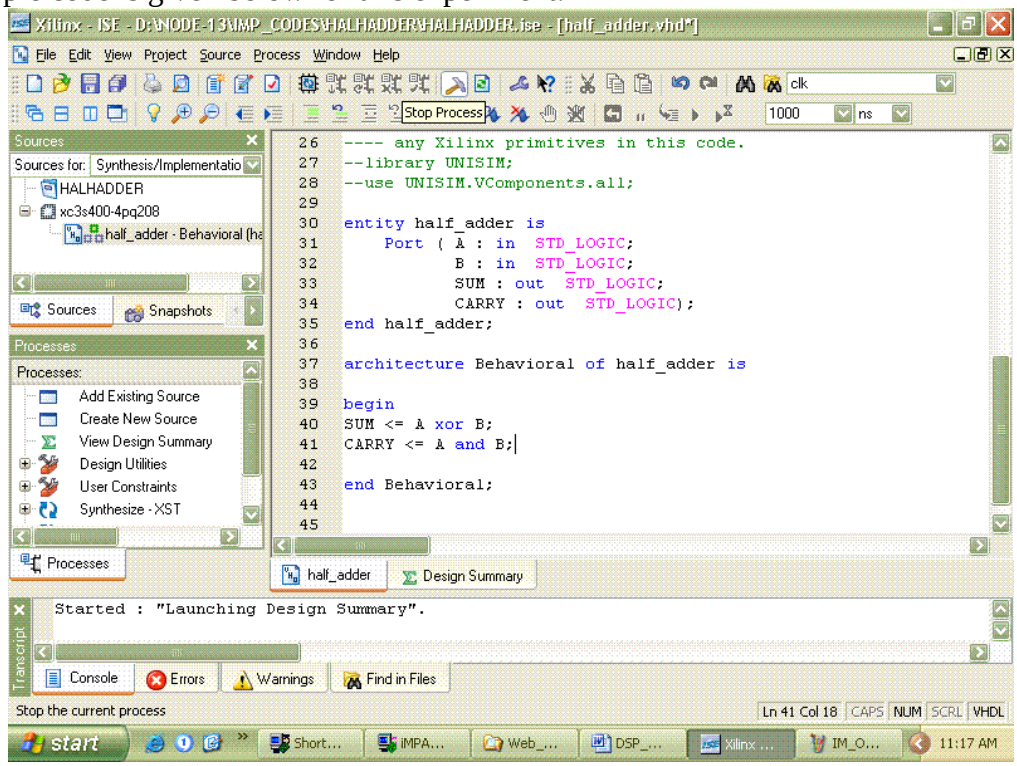
Then say *Next*→*Define ports*.In this case

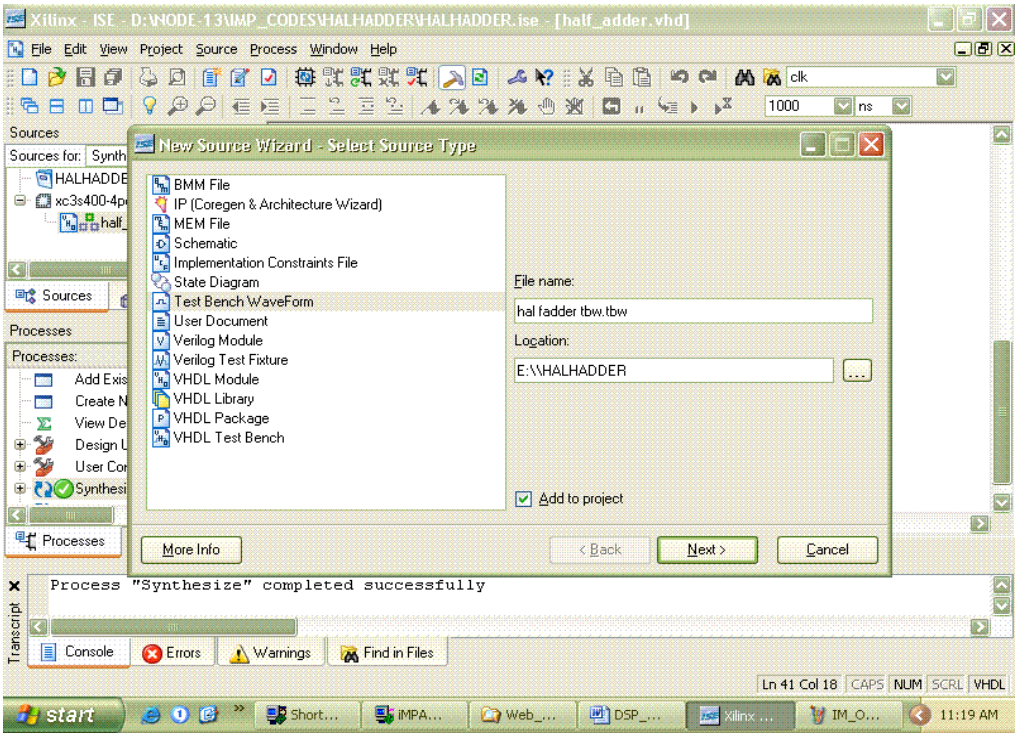
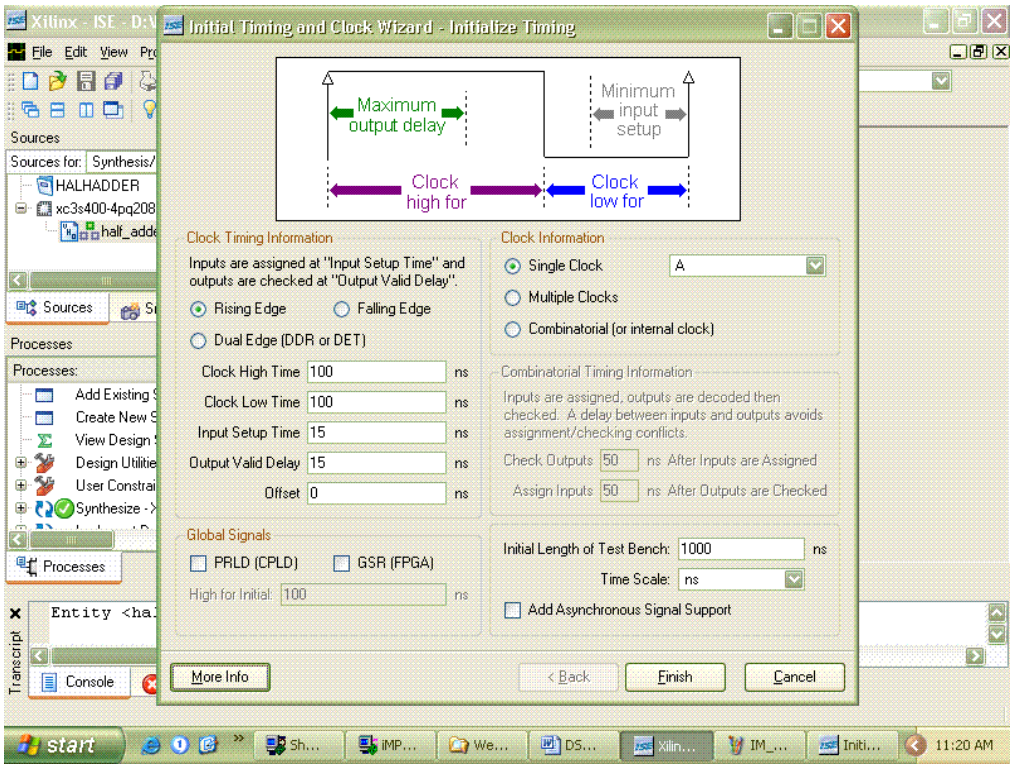
- *a* and *b* are the input ports defined as *in*
- *sum* and *carry* are output ports defined as *out*

after this say *Next* twice and then *Finish*

Skeleton of the design is shown in the VHDL editor.



Step 4:	*Writing the Behavioural VHDL Code in VHDL Editor* Sample code is given below for this experiment. 
Step 5	*Check Syntax* Run the *Check syntax* → *Process window* → *synthesize* →*check syntax* >, and remove errors if present.

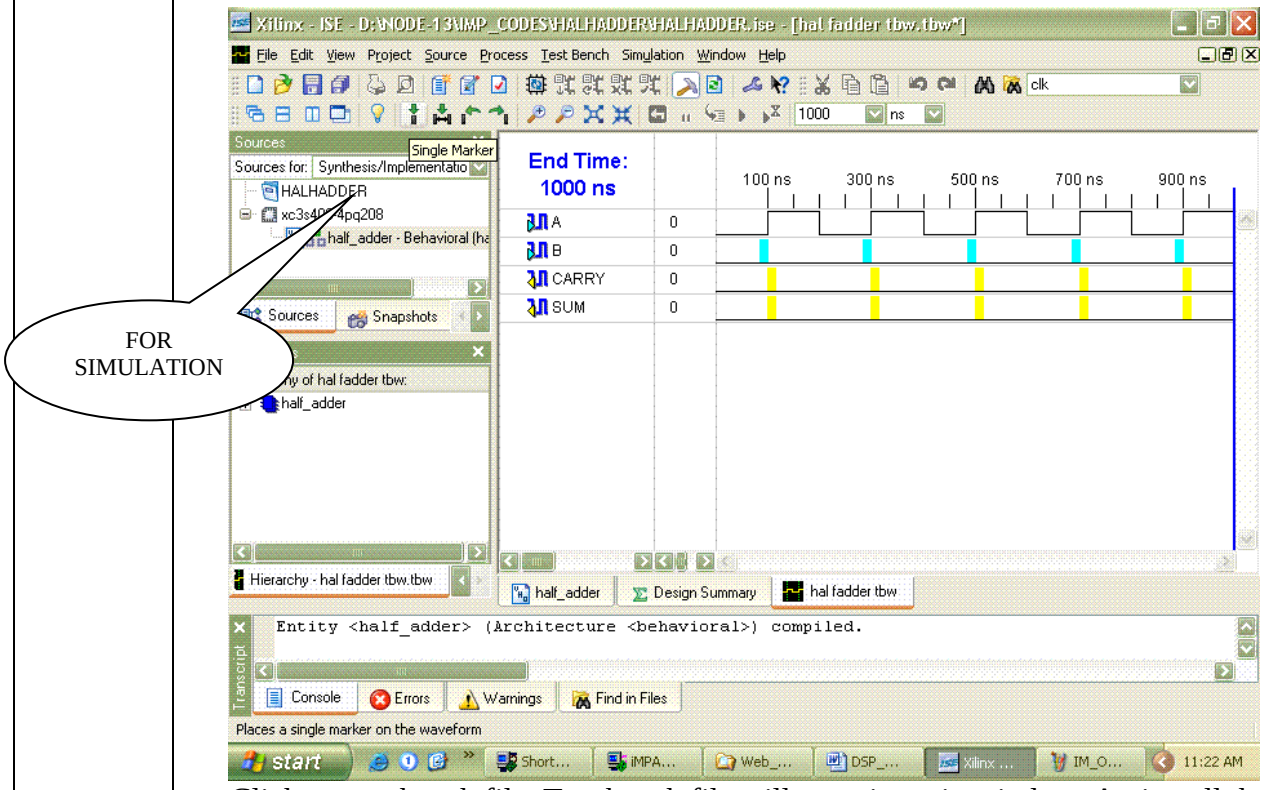
Step 6	*Creating a test bench file* Verify the operation of your design before you implement it as hardware. Simulation can be done using ISE simulator. For this click on the symbol of FPGA device and then right click → *Click on new source* →*Test Bench Waveform and give the name* → *Select entity* →*Finish.*  Select the desired parameters for simulating your design. In this case *combinational circuit and Simulation time.* 

Step 7:

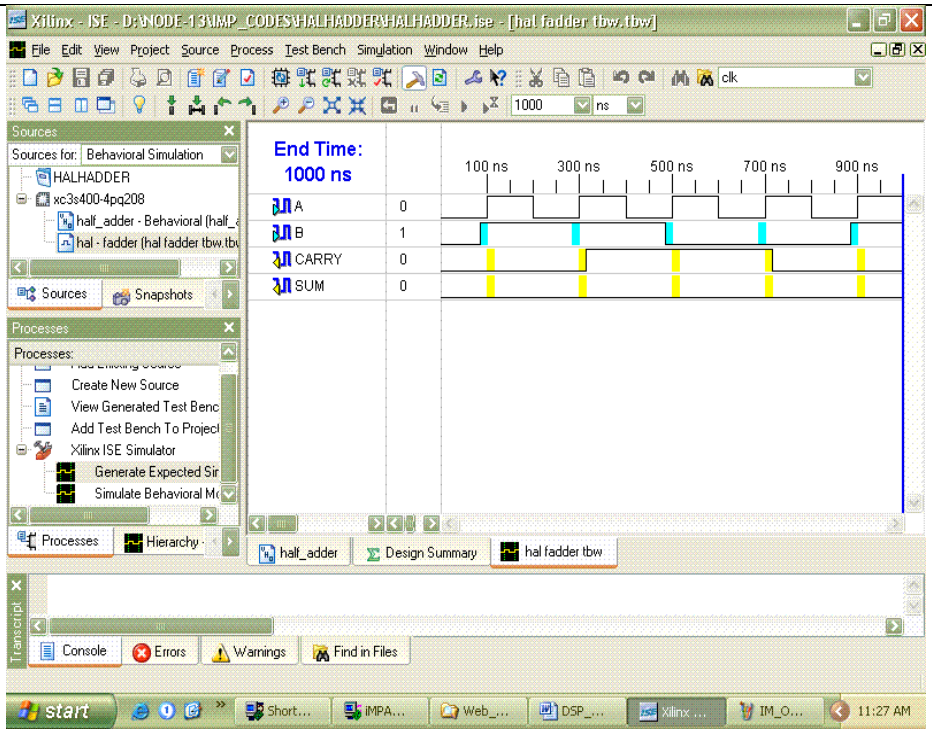
Simulate the code
Simulation Tools
ISE tool supports the following simulation tools:

- *HDL Bencher is an automated test bench creation tool. It is fully integrated with Project Navigator.*
- *ModelSim from Model Technology, Inc., is integrated in Project Navigator to simulate the design at all steps (Functional and Timing). ModelSim XE, the Xilinx Edition of Model Technology, Inc.'s ModelSim application, can be installed from the MTI CD included in your ISE Tool*

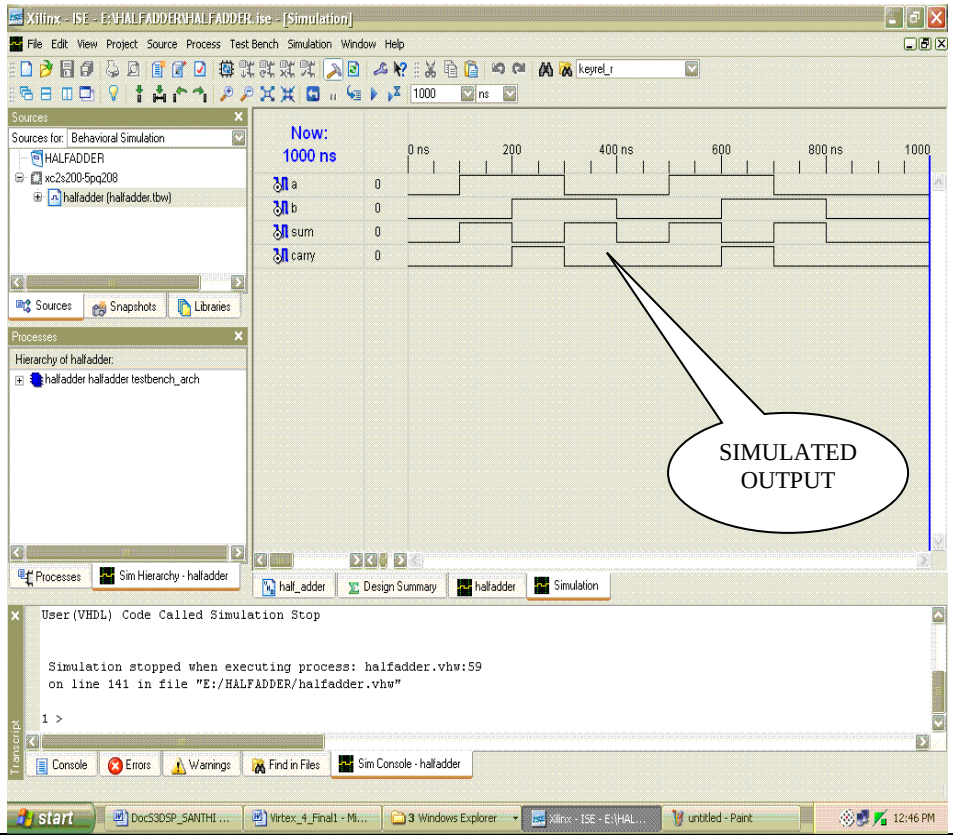
In source Window from the Drop-down menu select *Behavioural Simulation* to view the created test Bench file.



Click on test bench file. Test bench file will open in main window. Assign all the signals and save File. From the source of process window. Click on *Simulate Behavioral Model* in Process window.



Verify your design in wave window by seeing behaviour of output signal with respect to input signal. Close the ISE simulator window

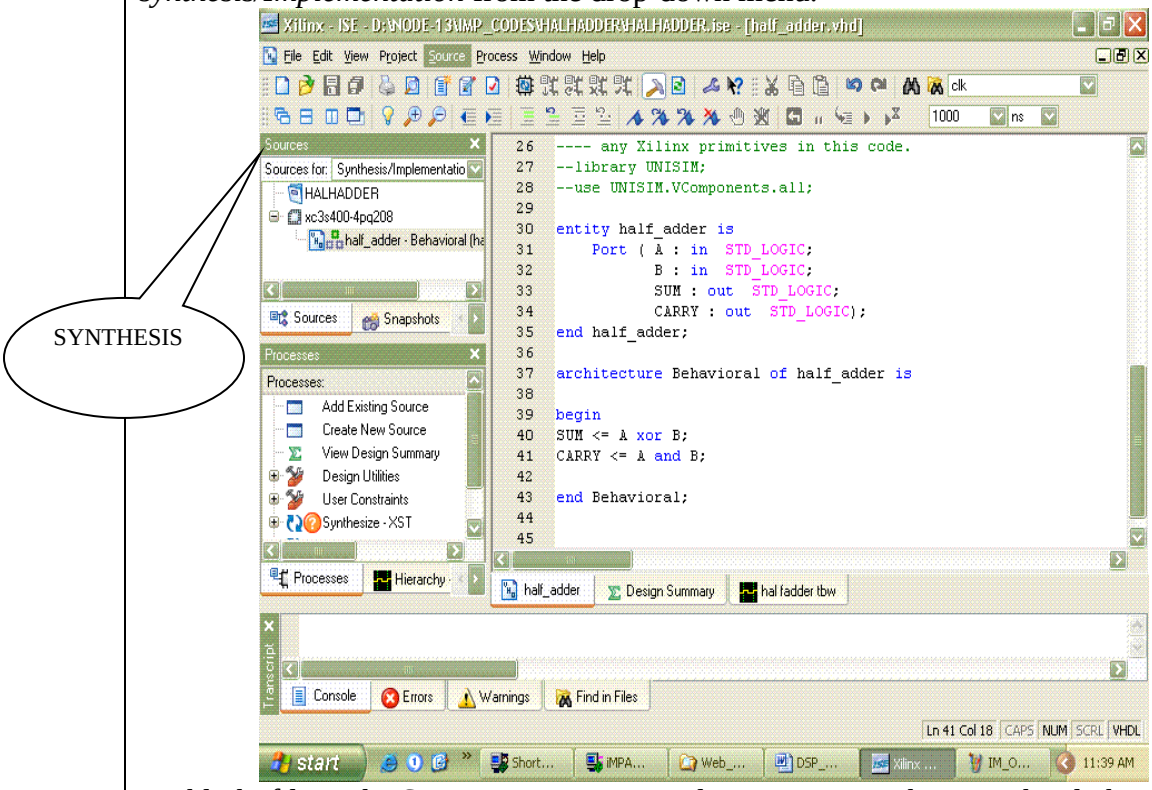


Step 8:

Synthesize the design using XST.

Translate your design into gates and optimize it for the target architecture. This is the synthesis phase.

Again for synthesizing your design, from the source window select, *synthesis/Implementation* from the drop-down menu.

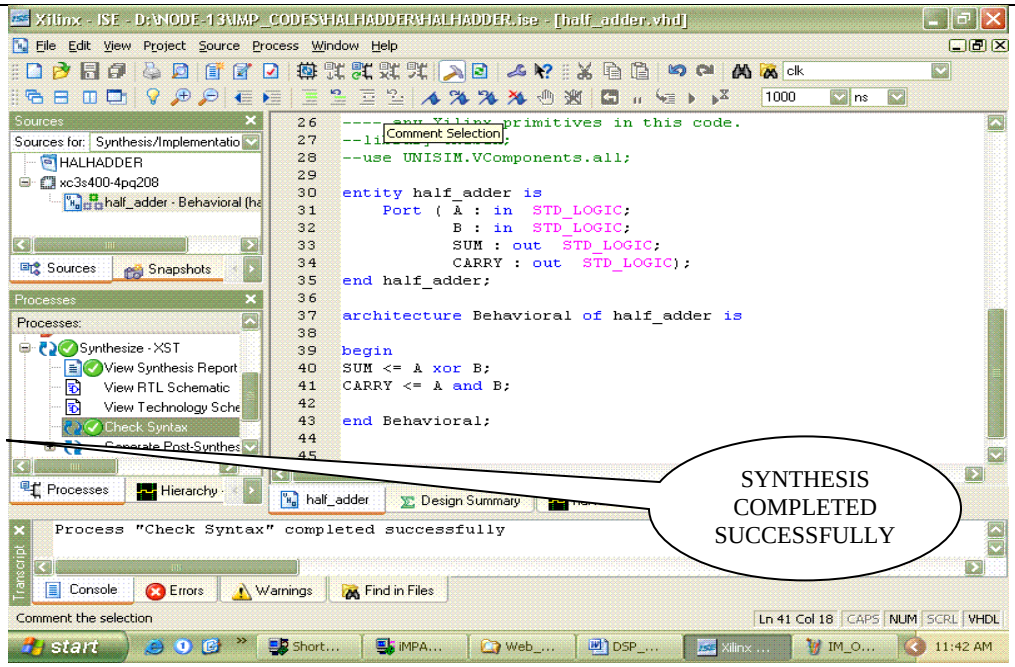


Highlight file in the Sources in Project window. To run synthesis, right-click on Synthesize, and the Run option, *or* double-click on Synthesize in the Processes for Current Source window. Synthesis will run, and

- a green check ✓will appear next to Synthesize when it is successfully completed.
- a red cross ✘indicates an error was generated and
- a yellow exclamation ! mark indicates that a warning was generated, (warnings are OK).

Check the synthesis report.

If there are any errors correct it and rerun synthesis.

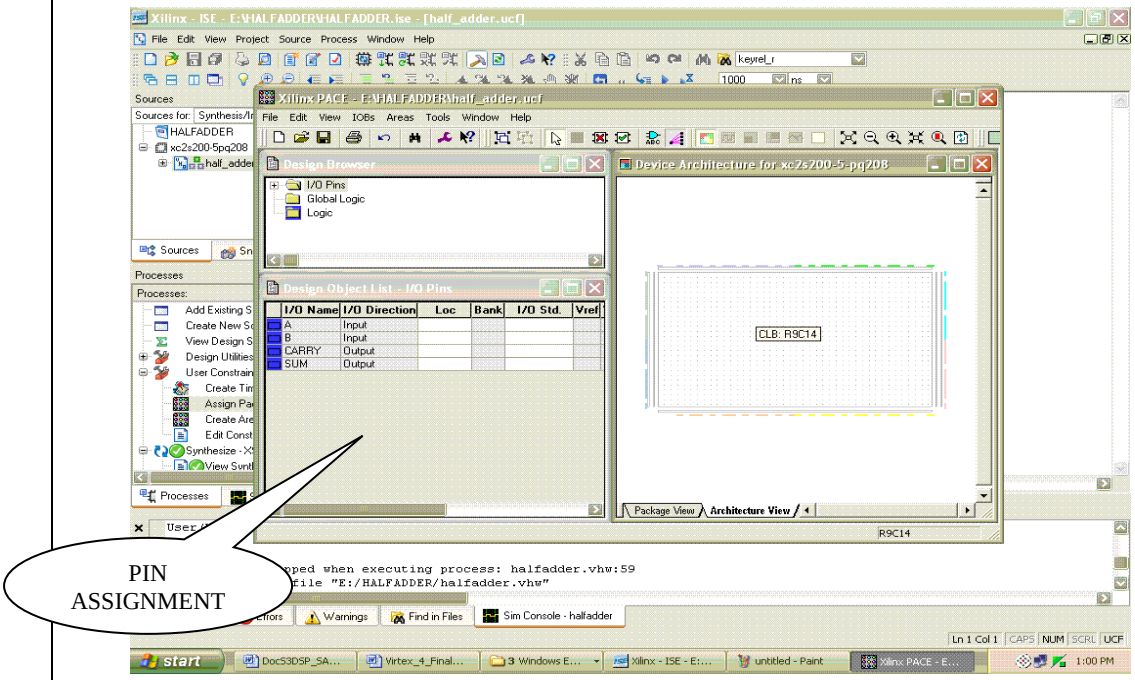


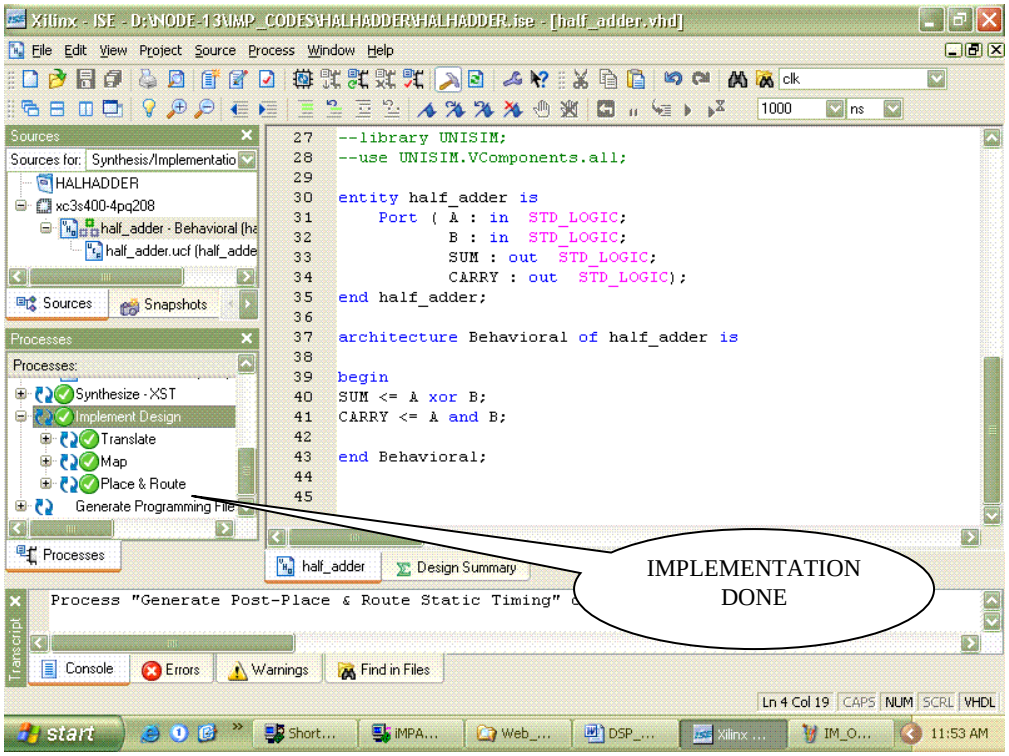
Step 9:

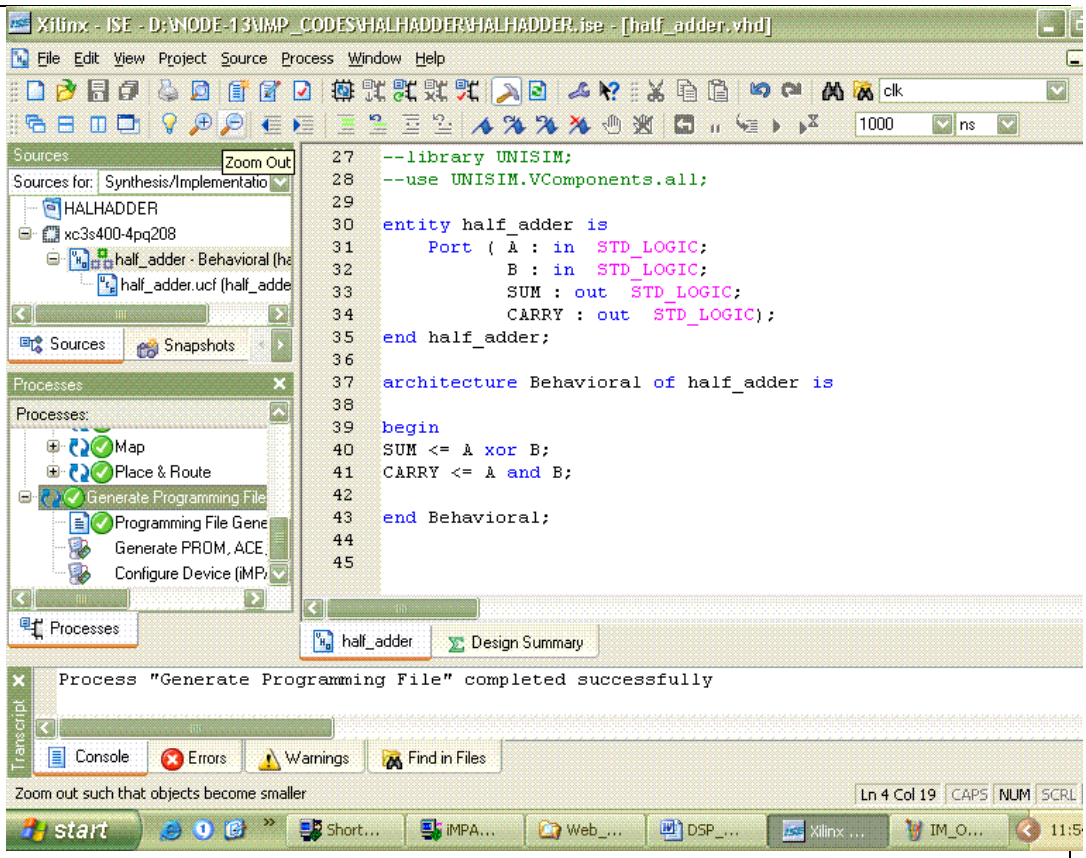
Create Constraints File(UCF)

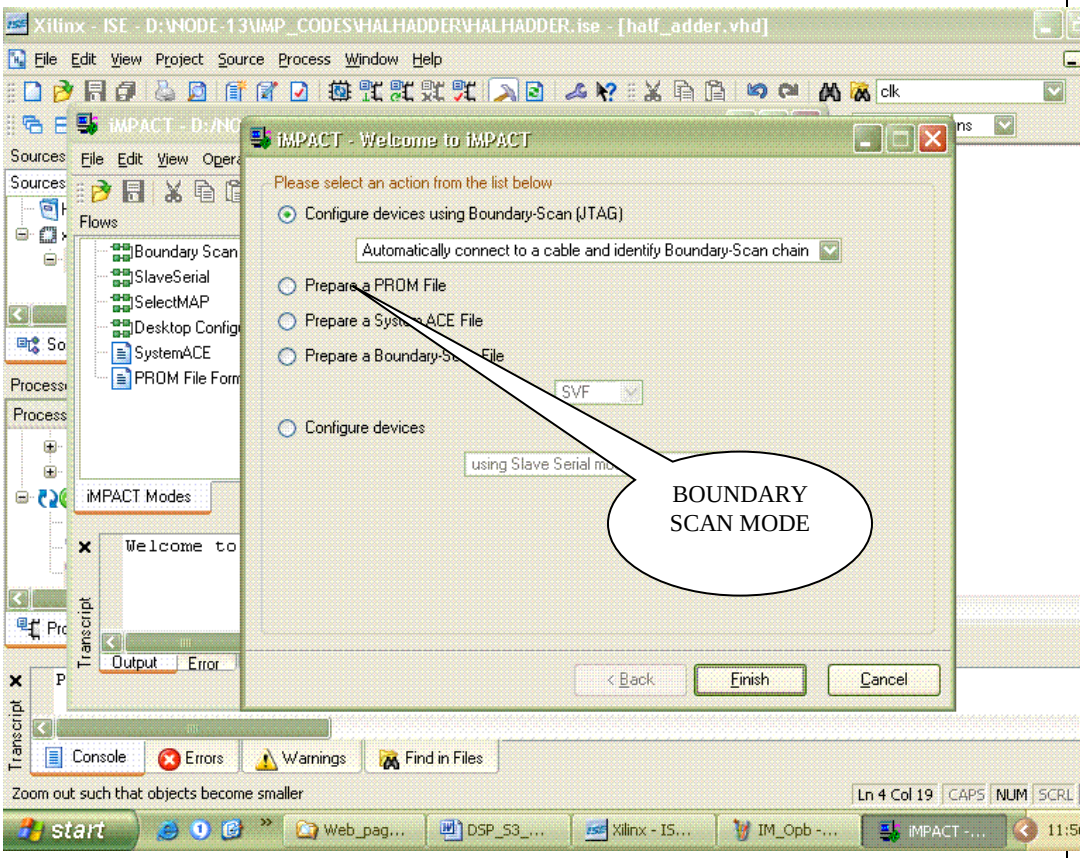
Click on the symbol of FPGA device and then right click→ Click on new source →Implementation Constraints File and give the name → Select entity→Finish.

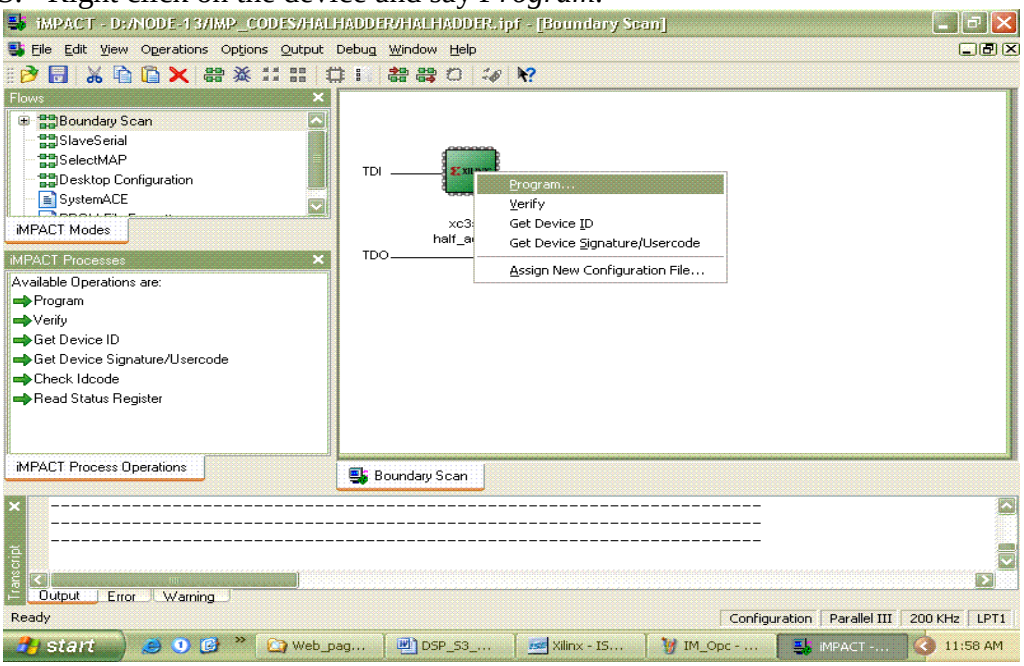
Click on *User Constraint and* in that Double Click on *Assign Package Pins* option in Process window. Xilinx PACE window opens. Enter all the pin assignments in PACE., depending upon target device and number of input and outputs used in your design. *(sample code is given below for given design.)*



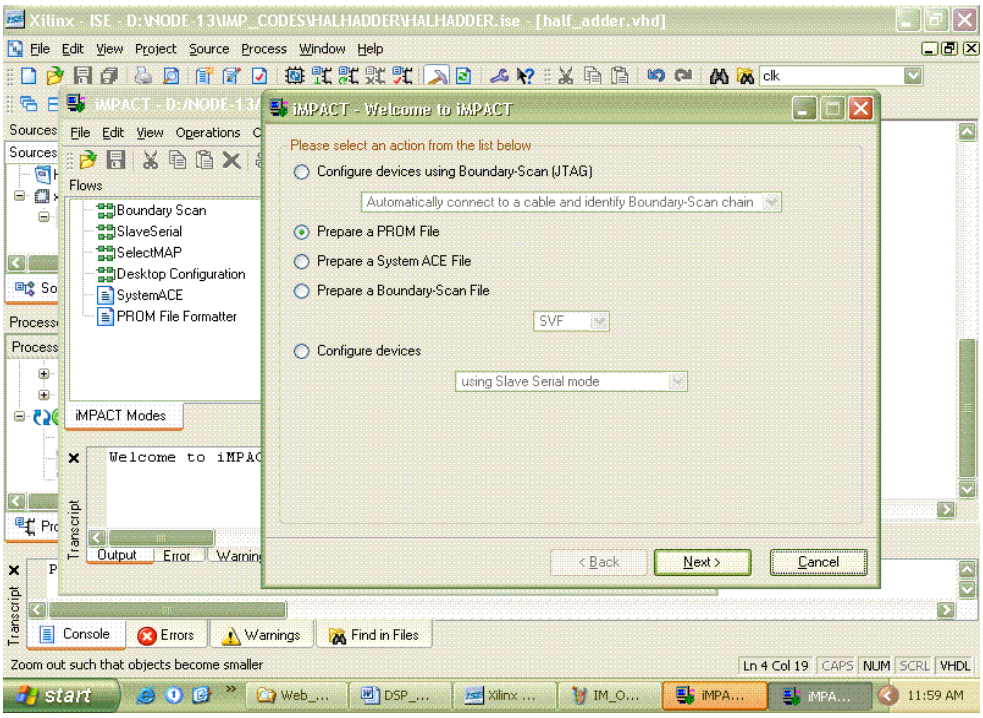
Step 10:	*Implementing a Design* Once synthesis is complete, you can place and route your design to fit into a Xilinx device, and you can also get some post place-and-route timing information about the design. The implementation stage consists of taking the synthesized netlist through translation, mapping, and place and route. To check your design as it is implemented, reports are available for each stage in the implementation process. Use the Xilinx Constraints Editor to add timing and location constraints for the implementation of your design. This procedure runs you through the basic flow for implementation. Right-click on *Implement Design*, and choose the Run option, *or* double left-click on *Implement Design.* 
Step 11:	*Generating Programming File* Right-click on *Generate Programming File*, choose the Run option, *or* double left-click on *Generate Programming File***.** This will generate the Bit stream
Step 12	*Downloading in* Boundary Scan Mode. *Note : Xilinx provides 2-tools for downloading purpose, viz.* • *iMPACT - is a command line and GUI based tool* • *PROM File Formatter*


Xilinx - ISE - D:\NODE-13\IMP_CODES\HALHADDER\HALHADDER.ise - [half_adder.vhd]
File Edit View Project Source Process Window Help
Sources
Sources for: Synthesis/Implementation
HALHADDER
xc3s400-4pq208
half_adder - Behavioral (ha
half_adder.ucf (half_adde
Sources Snapshots
Processes
Processes:
Map
Place & Route
Generate Programming File
Programming File Gene
Generate PROM, ACE
Configure Device (iMP
Processes
Zoom Out
27 --library UNISIM;
28 --use UNISIM.VComponents.all;
29
30 entity half_adder is
31 Port (A : in STD_LOGIC;
32 B : in STD_LOGIC;
33 SUM : out STD_LOGIC;
34 CARRY : out STD_LOGIC);
35 end half_adder;
36
37 architecture Behavioral of half_adder is
38
39 begin
40 SUM <= A xor B;
41 CARRY <= A and B;
42
43 end Behavioral;
44
45
half_adder
Design Summary
Process "Generate Programming File" completed successfully
Transcript
Console
Errors
Warnings
Find in Files
Zoom out such that objects become smaller
Ln 4 Col 19
CAPS
NUM
SCRL
start

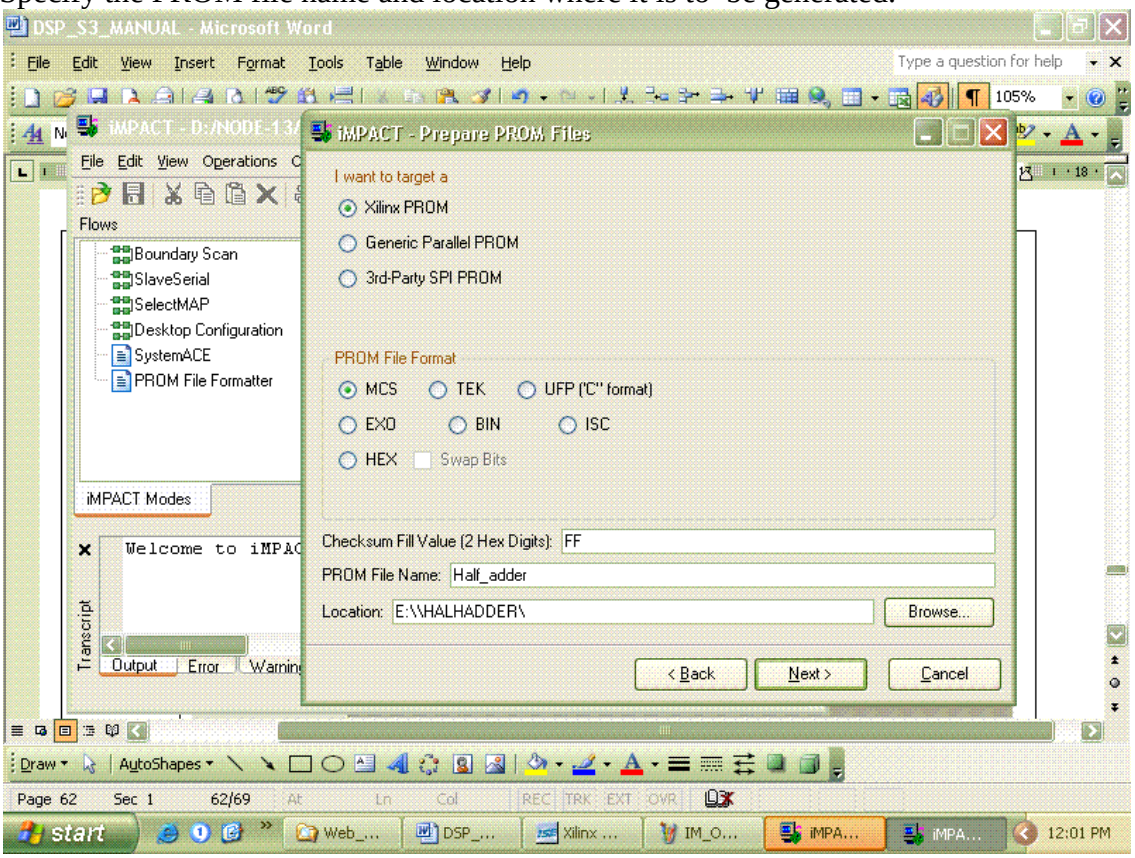

Xilinx - ISE - D:\NODE-13\IMP_CODES\HALHADDER\HALHADDER.ise - [half_adder.vhd]
File Edit View Project Source Process Window Help
iMPACT - Welcome to iMPACT
Please select an action from the list below
Configure devices using Boundary-Scan (JTAG)
Automatically connect to a cable and identify Boundary-Scan chain
Prepare a PROM File
Prepare a System ACE File
Prepare a Boundary-Scan File
SVF
Configure devices
using Slave Serial mo
Flows
Boundary Scan
SlaveSerial
SelectMAP
Desktop Config
SystemACE
PROM File Form
iMPACT Modes
Welcome to
Output
Error
Transcript
< Back
Finish
Cancel
BOUNDARY SCAN MODE
Console
Errors
Warnings
Find in Files
Zoom out such that objects become smaller
Ln 4 Col 19
CAPS
NUM
SCRL
start

	Procedure for downloading using iMPACT 1. Boundary Scan Mode *1.* Right click on "*Configure Device (iMPACT)" -> and Say RUN* or Double click on "*Configure Device (iMPACT)".* 2. Right click in workspace and say *Initialize chain* .The device is seen. *3.* Right click on the device and say *Program.*  If the device is programmed properly, it says *Programming Succeeded* or else. *Programming Failed*. The *DONE* Led glows green if programming succeeds.
	Note: Before downloading make sure that Protoboard is connected to PC's parallel port with the cable provided and power to the Protoboard is ON.
Step 13:	Apply input through DIP Switches, output is displayed on LEDs
Step 14:	Configuration through PROM: *Generating PROM file*: FPGA can also be configured in Master Serial Mode through PROM. For this you need to program the PROM through a .mcs file.

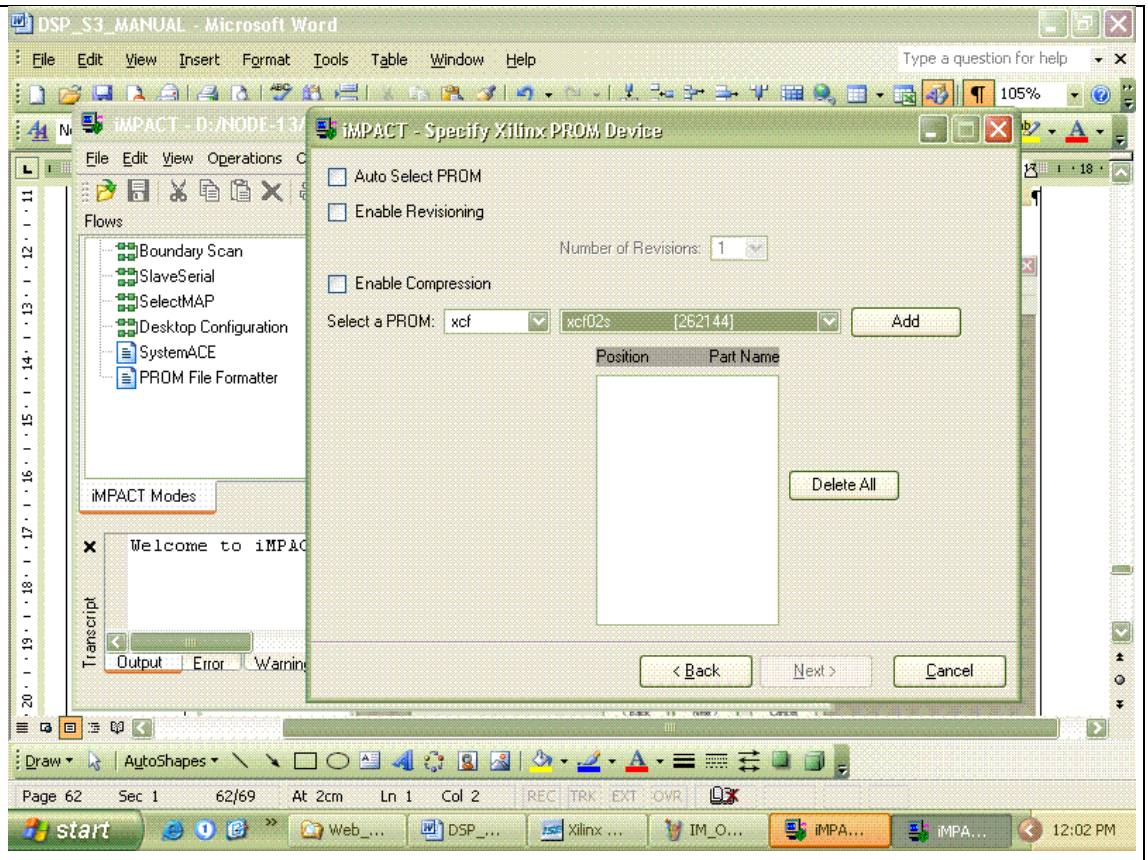
Right click on "*Generate PROM,ACE or JTAG file" -> and Say RUN* or Double click on "*Generate PROM,ACE or JTAG file"*



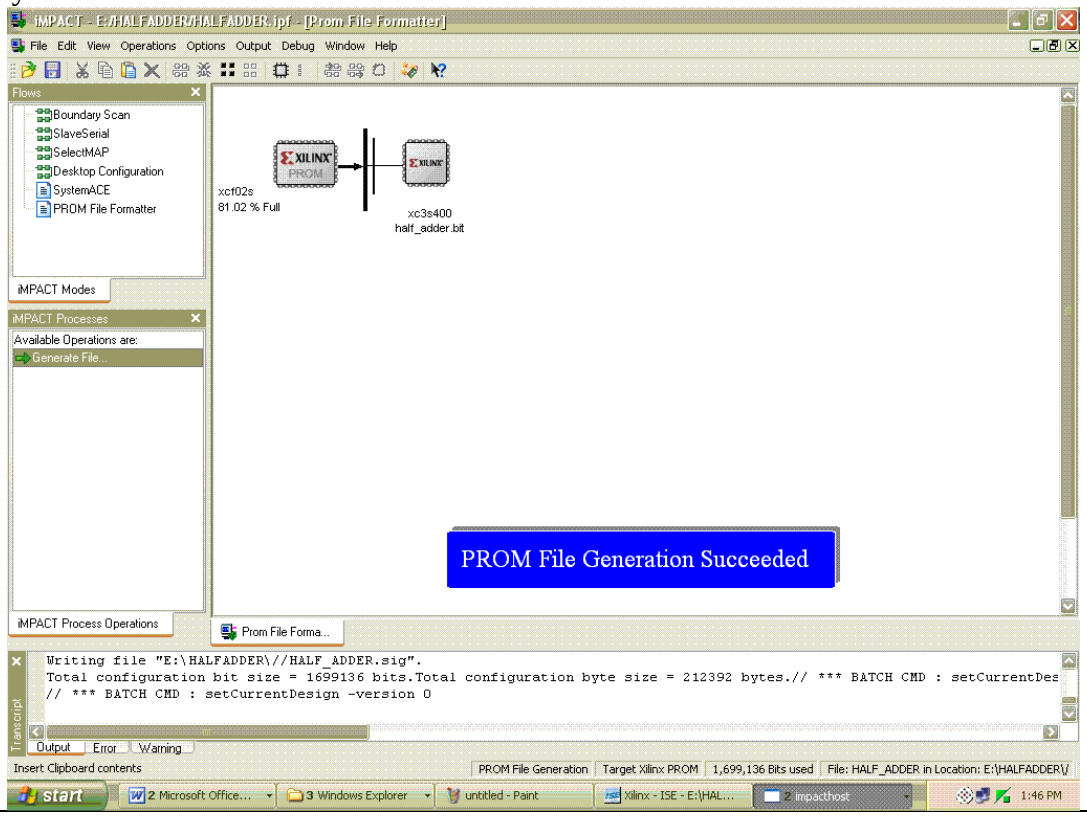
Specify the PROM file name and location where it is to be generated.



Specify the desired parameters of the PROM on board and say *ADD* then *FINISH*



Say *Generate File* from the Process Window.

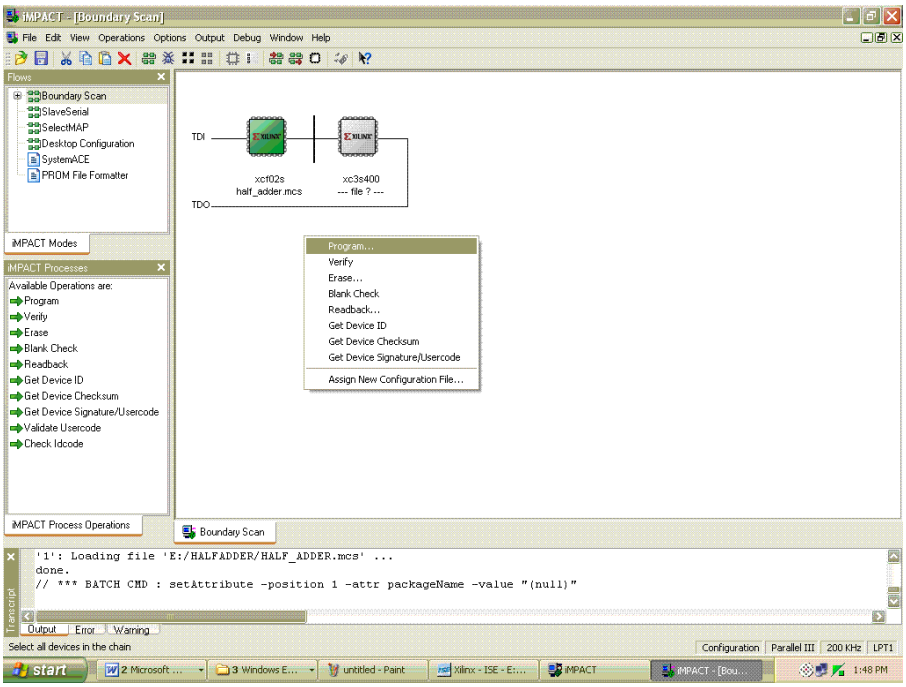


PROGRAMMING THE PROM

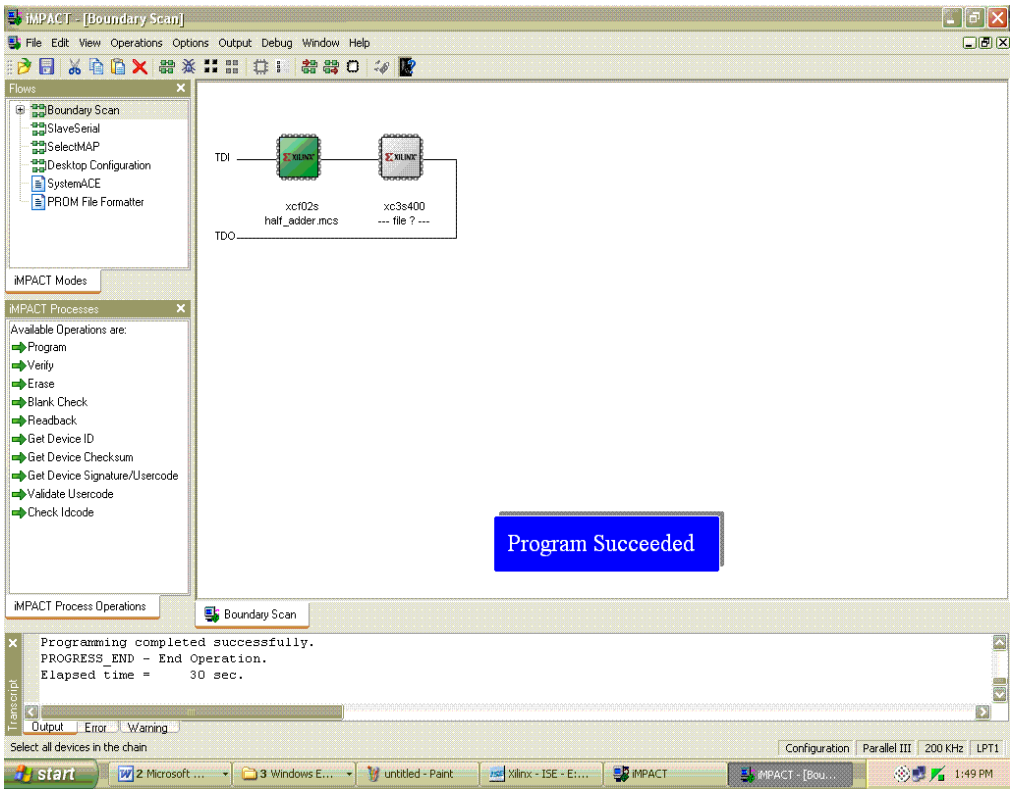
Note: Check the Jumper setting on the board. Refer the Chapter jumper Setting

Similar to Step 12.Initialize chain through iMPACT. PROM and FPGA devices on board are seen .Assign the generated mcs file and bit file as desired.

Right click the PROM symbol and say *PROGRAM.*



Now, whenever the board is powered on in master serial mode, FPGA is configured through PROM automatically.



Expt . No: 1 Date :	# Full Adder

Aim:

Realize the full adder using Verilog.

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

A combinational circuit that performs the addition of three bits is called a half-adder. This circuit needs three binary inputs and produces two binary outputs. One of the input variables designates the augend and other designates the addend. Mostly, the third input represents the carry from the previous lower significant position. The output variables produce the sum and the carry.

The simplified Boolean functions of the two outputs can be obtained as below:

Sum S = x $\oplus$ y $\oplus$ z

Carry C = xy + xz + yz

Where x, y & z are the two input variables.

Procedure:

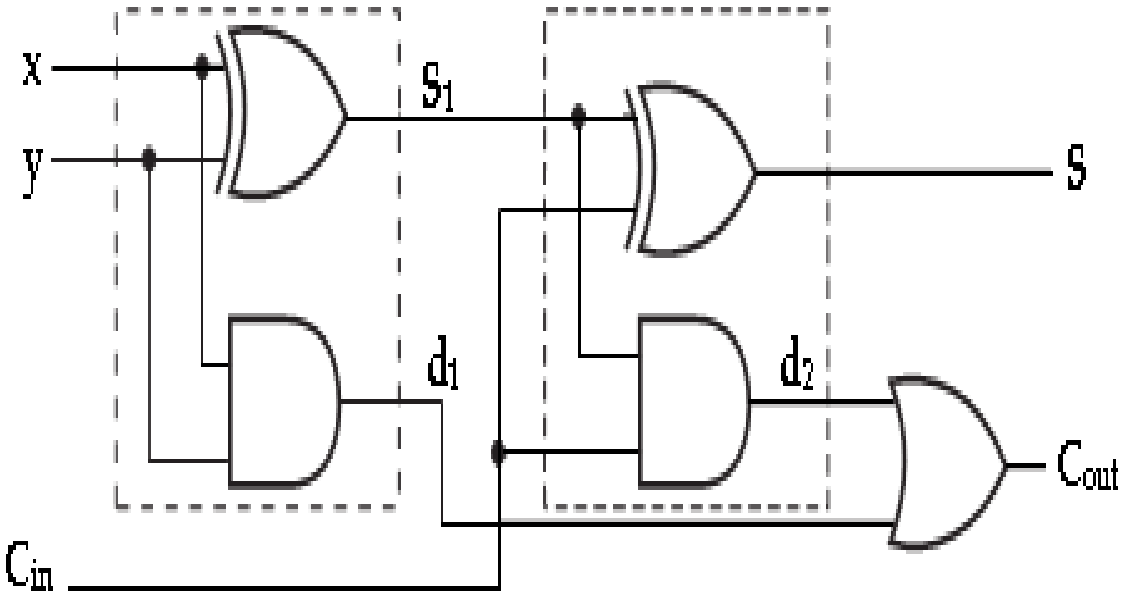
1. The full-adder circuit is designed and the Boolean function is found out.
2. The Verilog Module Source for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

Program:

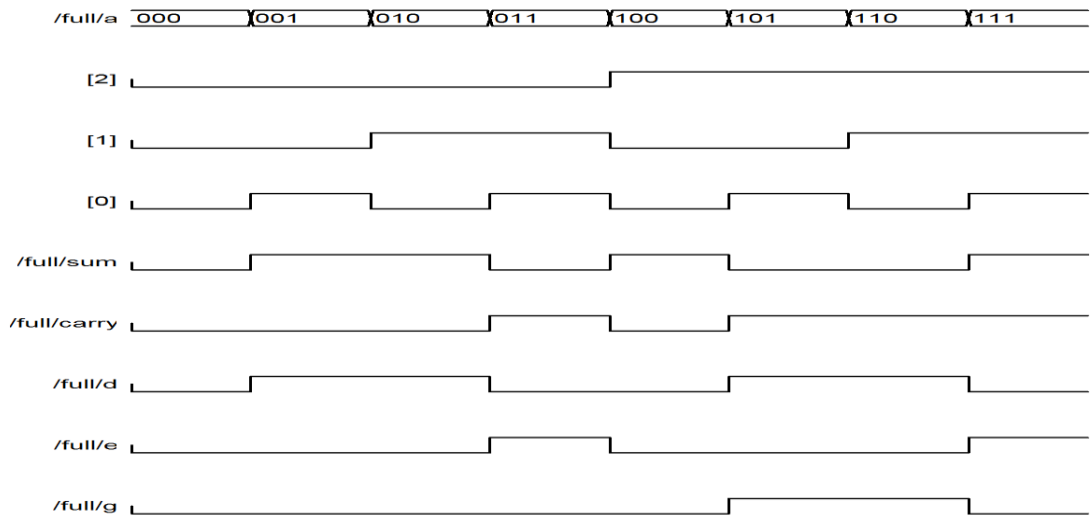
```
//Gate-level description of Full Adder using two Half Adder
//Description of Half Adder
module halfadder(s,co,x,y);
input x,y;
output s,co;
//Instatiate primitive gates
xor (s,x,y);
and (co,x,y);
endmodule
//Description of Full Adder
module fulladder(s,co,x,y,ci);
input x,y,ci;
```

```
output s,co;
wire s1,d1,d2;    //Outputs of first XOR and AND gates
//Instantiate Half Adder
halfadder ha_1(s1,d1,x,y);
halfadder ha_2(s,d2,s1,ci);
or or_gate(co,d2,d1);
endmodule
//Stimulus for testing Full Adder
module simulation;
reg x,y,ci;
wire s,co;
//Instantiate Full Adder
fulladder fa_test(s,co,x,y,ci);
initial
begin
x=1'b0; y=1'b0; ci=1'b0;
#100    x=1'b0; y=1'b0; ci=1'b1;
#100    x=1'b0; y=1'b1; ci=1'b0;
#100    x=1'b0; y=1'b1; ci=1'b1;
#100    x=1'b1; y=1'b0; ci=1'b0;
#100    x=1'b1; y=1'b0; ci=1'b1;
#100    x=1'b1; y=1'b1; ci=1'b0;
#100    x=1'b1; y=1'b1; ci=1'b1;
end
endmodule
```

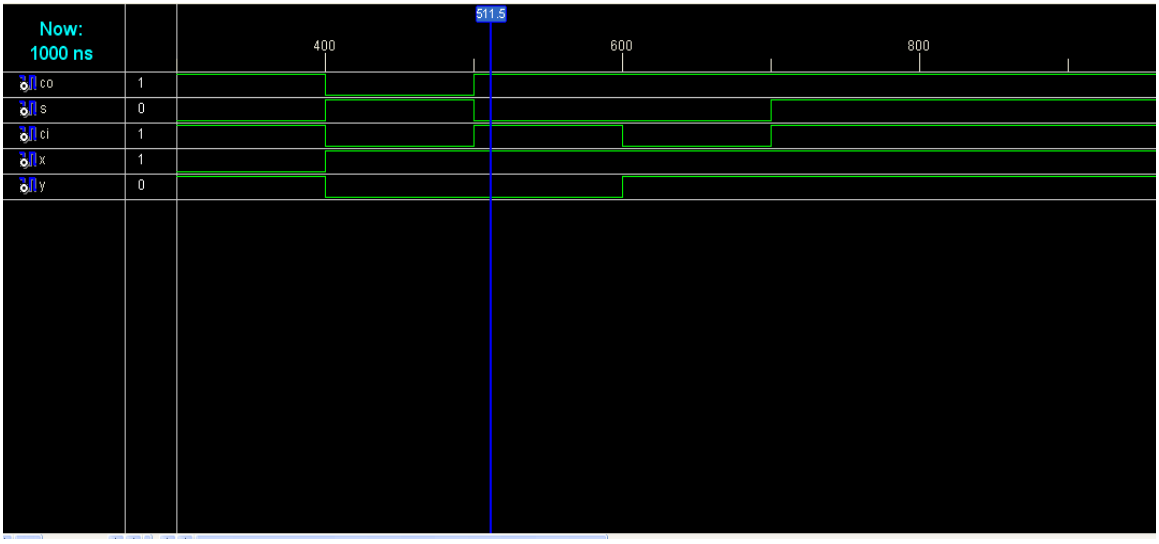
Diagram:



Waveform:



Waveform:



Result:

Thus the logic circuit for the Full adder is designed in Verilog HDL and the output is verified.

Expt . No: 2 Date :	# Design of 8 Bit Adders **1. DESIGN of RIPPLE CARRY ADDER using VERILOG HDL**

Aim:

To Design Ripple Carry Adder using Verilog HDL

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

The *n*-bit adder built from n one –bit full adders is known as ripple carry adder because of the carry is computed. The addition is not complete until *n-1*th adder has computed its S_{n-1} output; that results depends upon ci input, n and so on down the line, so the critical delay path goes from the 0-bit inputs up through c_i's to the *n-1* bit.(We can find the critical path through the n-bit adder without knowing the exact logic in the full adder because the delay through the n-bit adder without knowing the exact logic in the full adder because the delay through the n-bit carry chain is so much longer than the delay from a and b to s). The ripple-carry adder is area efficient and easy to design but it is when n is large.It can also be called as cascaded full adder.

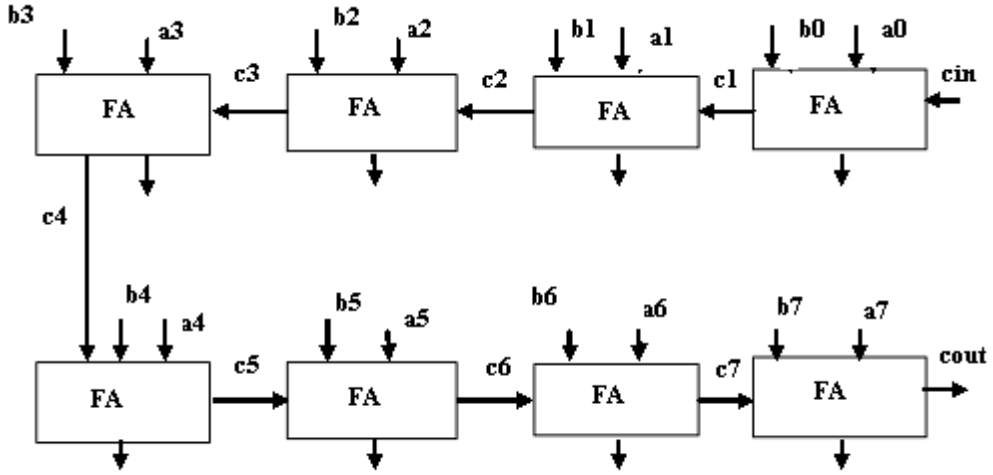
The simplified Boolean functions of the two outputs can be obtained as below:

Sum $s_i = a_i$ xor b_i xor c_i
Carry $c_{i+1} = a_i b_i + b_i c_i + a_i c_i$
Where x, y & z are the two input variables.

Procedure:

1The full-adder circuit is designed and the Boolean function is found out.
2.The Verilog Module Source for the circuit is written.
3.It is implemented in Model Sim and Simulated.
4.Signals are provided and Output Waveforms are viewed.

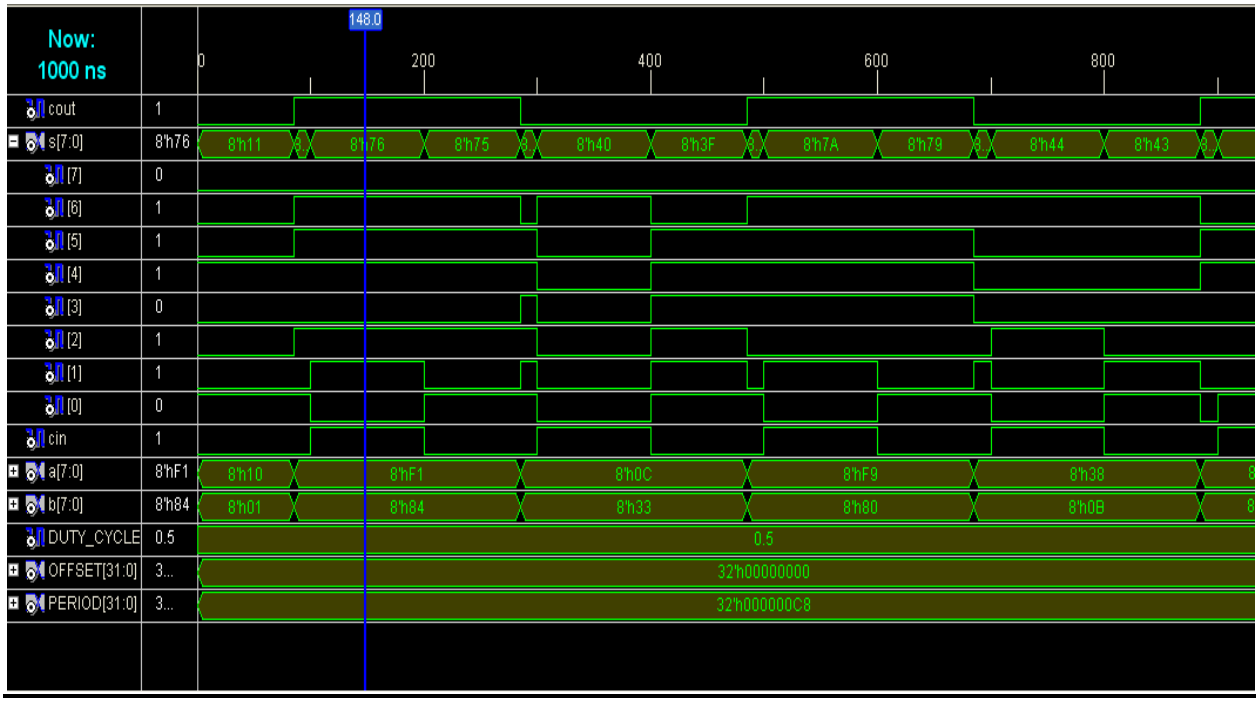
Circuit diagram:



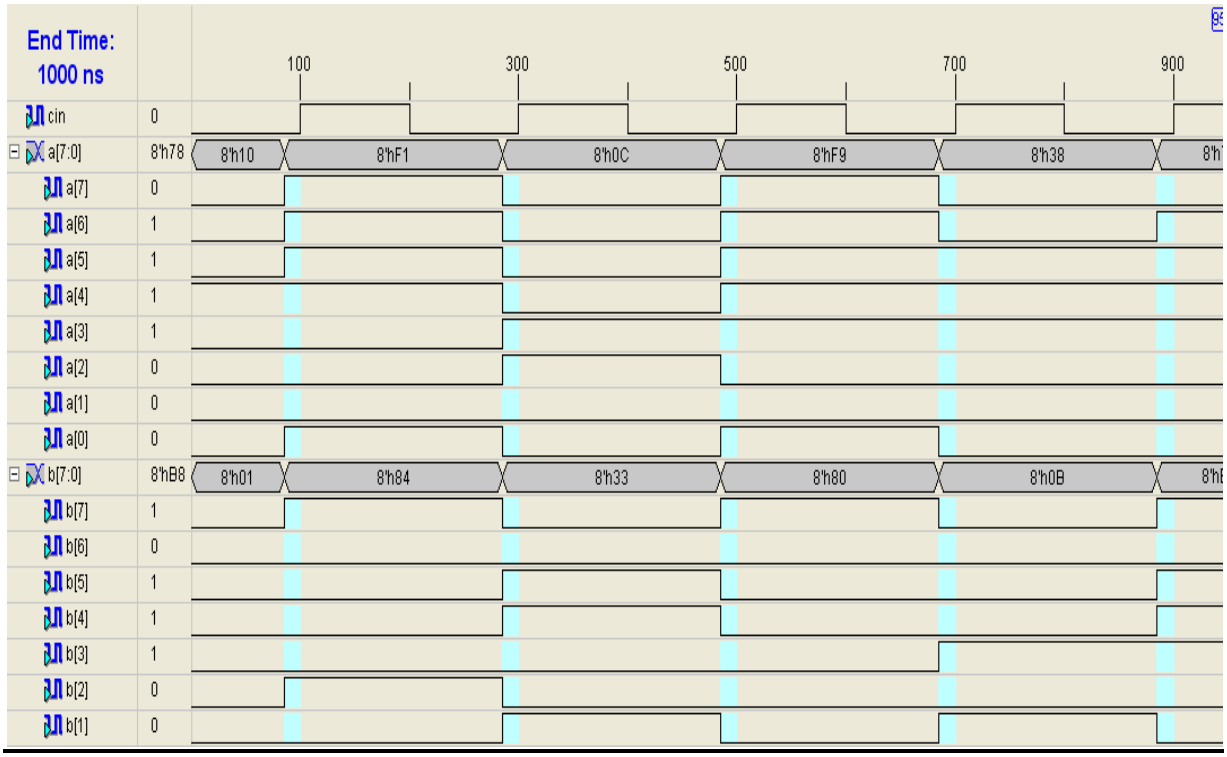
Ripple carry adder using verilog code:

```
module ripplecarryadder(s,cout,a,b,cin);
output[7:0]s;
output cout;
input[7:0]a,b;
input cin;
wire c1,c2,c3,c4,c5,c6,c7;
fulladd fa0(s[0],c1,a[0],b[0],cin);
fulladd fa1(s[1],c2,a[1],b[1],c1);
fulladd fa2(s[2],c3,a[2],b[2],c2);
fulladd fa3(s[3],c4,a[3],b[3],c3);
fulladd fa4(s[4],c5,a[4],b[4],c4);
fulladd fa5(s[5],c6,a[5],b[5],c5);
fulladd fa6(s[6],c7,a[6],b[6],c6);
fulladd fa7(s[7],cout,a[7],b[7],c7);

endmodule
module fulladd(s,cout,a,b,cin);
output s,cout;
input a,b,cin;
wire s1,c1,c2;
xor(s1,a,b);
xor(s,s1,cin);
and(c1,a,b);
and(c2,s1,cin);
xor(cout,c2,c1);
endmodule
```

Waveform of ripple carry adder:



Test bench wave form of ripple carry adder:



RESULT:

Thus the logic circuit for the Ripple carry adder is designed in Verilog HDL and the output is verified.

Expt . No: 3 Date :	# Design of 8 Bit Adders **2.DESIGN CARRY SAVE ADDER USING VERILOG HDL**

Aim:

To design Carry Save Adder using Verilog HDL

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

Carry save adders are suitable when three or more operands are to be added, as in some multiplication schemes. In this adder a separate sum and carry bit is generated for partial results , except when the last operand is added. For example, when three numbers are added, the first two are added using a carry save adder. The partial result is two numbers corresponding to the sum and the carry .The last operand is added using a second carry save adder stage. The results become the sum and carry numbers. Thus a carry save adder reduces the number of operands by one for each adder stage. Finally the sum and carry are added using an adder with carry propagation- for example carry look ahead adder.

Procedure;

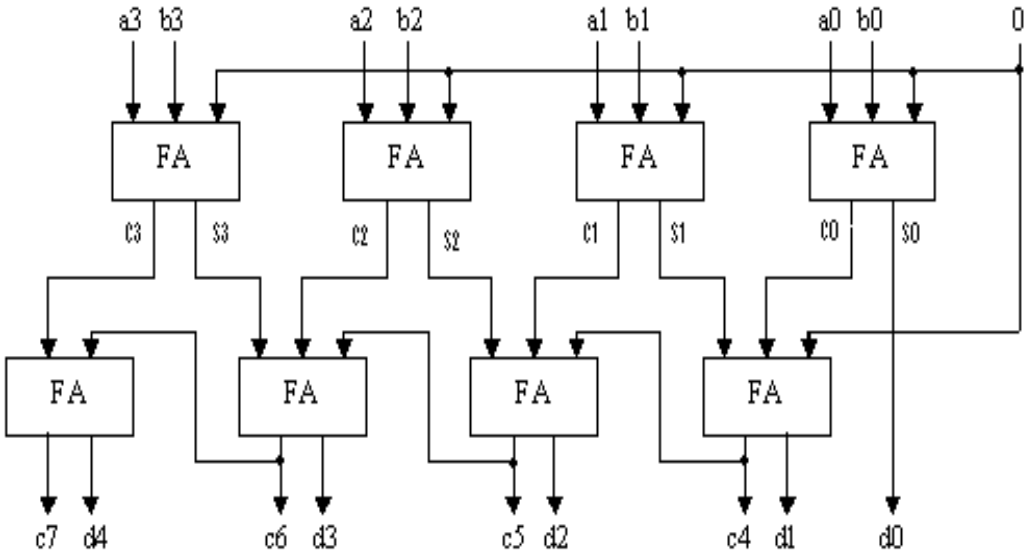
1. The carry save adder is designed.
2. The Verilog program source code for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

Carry save adder using Verilog:

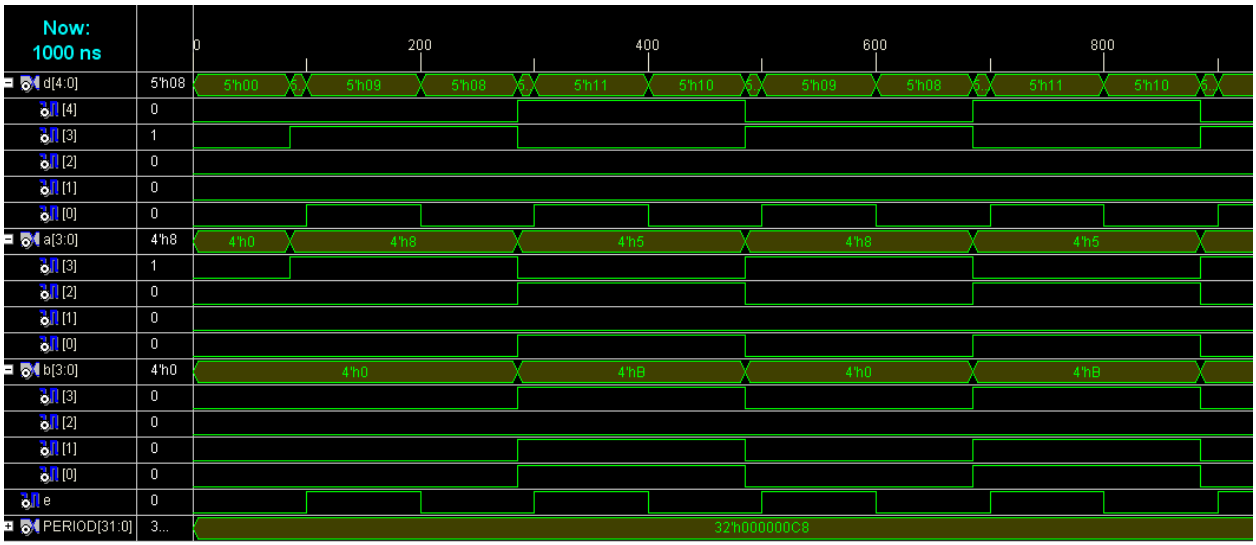
```
module carrysaveadder(d,a,b,e);
   output [4:0]d;
   input e;
   input [3:0]a,b;
wire s1,s2,s3,c0,c1,c2,c3,c4,c5,c6,c7;

fulladder a1(d[0],c7,a[0],b[0],e);
fulladder a2(s3,c6,a[1],b[1],e);
```

```
fulladder a3(s2,c5,a[2],b[2],e);
fulladder a4(s1,c4,a[3],b[3],e);
fulladder a5(d[1],c3,c7,s3,e);
fulladder a6(d[2],c2,c6,c3,s2);
fulladder a7(d[3],c1,c5,s1,c2);
fulladder a8(d[4],c0,c4,c1,e);
endmodule

module fulladder(s,c, x,y,z);
   output s,c;
   input x,y,z;
xor (s,x,y,z);
assign c = ((x & y )|(y & z)|( z & x)) ;
endmodule
```

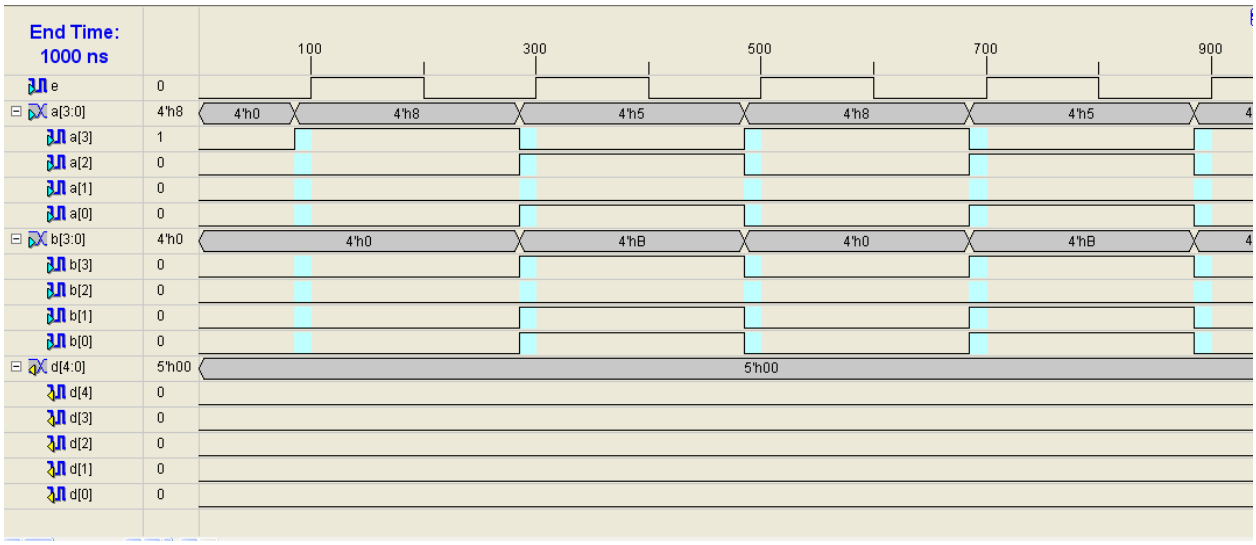
Logic Diagram:



Waveform carry save adder:



Test bench waveform carry-save adder:



RESULT:

Thus the logic circuit for the carry save adder is designed in Verilog HDL and the output is verified.

Expt . No: 4 Date :	# Design of 8 Bit Adders **3.DESIGN CARRY SELECT ADDER USING VERILOG HDL**

Aim:

To design a Carry Select Adder using Verilog HDL

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

Carry-select adders use multiple narrow adders to create fast wide adders. A carry-select adder provides two separate adders for the upper words, one for each possibility. A MUX is then used to select the valid result. Consider an 8-bit adder that is split into two 4-bit groups. The lower-order bits and are fed into the 4_bit adder l to produce the sum bits and a carry-out bit .the higher order bits and are used as input to one 4_bit adder and and $y_{11}y_{10}y_9y_8$ are used as input of the another 4_bit adder. Adder U0 calculates the sum with a carry-in of C3=0.while U1 does the same only it has a carry-in value of C3=1.both sets of results are used as inputs to an array of 2:1 MUXes .the carry bit from the adder L is used as the MUX select signal. If =0 then the results U0 are sent to the output, while a value of =1 selects the results of U1 for $s_{11}s_{10}s_9s_8$. The carry-out bit is also selected by the MUX array.

Procedure:

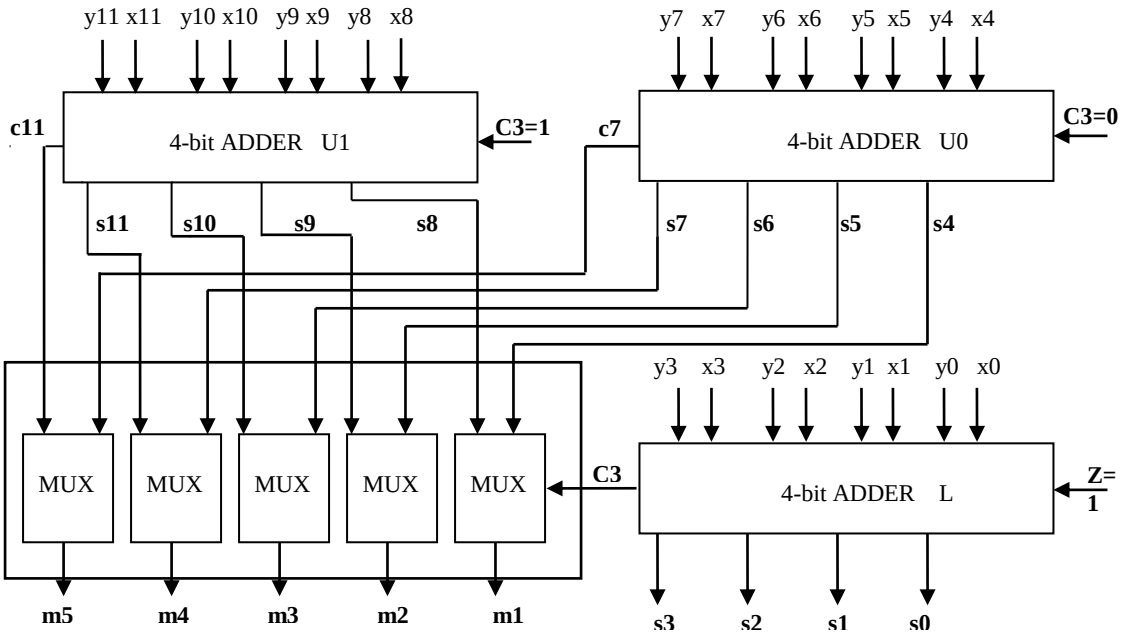
1. The carry-select adder circuit is designed and the Boolean function is found out.
2. The Verilog Module Source for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

carry-select adder using verilog:

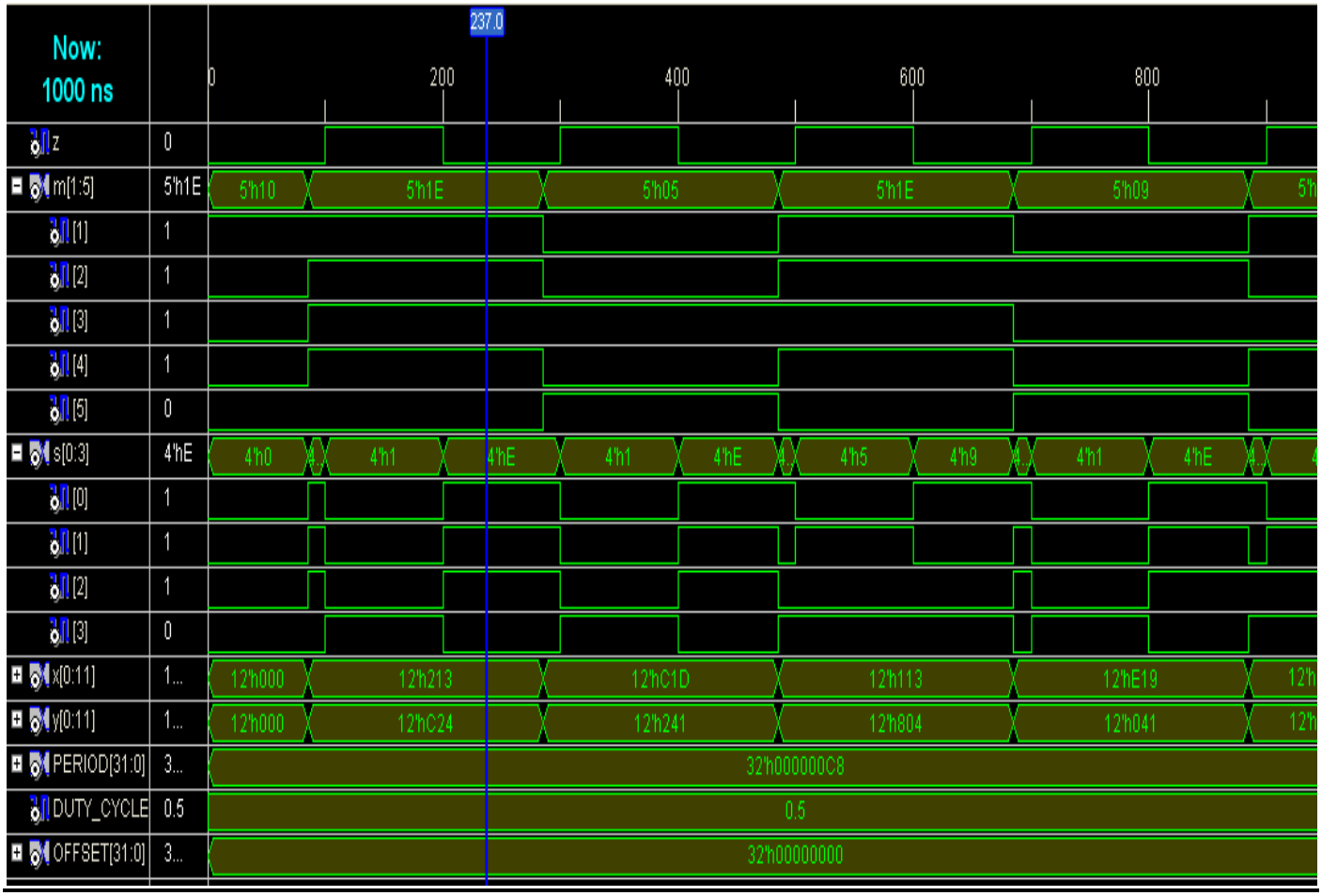
```
module project2(s, m, x, y, z);
output [0:3]s;
output [1:5]m;
input [0:11]x;
input [0:11]y;
input z;
wire c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,s4,s5,s6,s7,s8,s9,s10,s11;
```

```
fulladder f1(s[0],c0,x[0],y[0],z);
fulladder f2(s[1],c1,x[1],y[1],c0);
fulladder f3(s[2],c2,x[2],y[2],c1);
fulladder f4(s[3],c3,x[3],y[3],c2);
fulladder f5(s4,c4,x[4],y[4],c3);
fulladder f6(s5,c5,x[5],y[5],c4);
fulladder f7(s6,c6,x[6],y[6],c5);
fulladder f8(s7,c7,x[7],y[7],c6);
fulladder f9(s8,c8,x[8],y[8],~c3);
fulladder f10(s9,c9,x[9],y[9],c8);
fulladder f11(s10,c10,x[10],y[10],c9);
fulladder f12(s11,c11,x[11],y[11],c10);
muxer mu1(m[1],s4,s8,c3);
muxer mu2(m[2],s5,s9,c3);
muxer mu3(m[3],s6,s10,c3);
muxer mu4(m[4],s7,s11,c3);
muxer mu5(m[5],c7,c11,c3);
endmodule
module fulladder (s,c,x,y,z);
output s,c;
input x,y,z;
xor (s,x,y,z);
assign c = ((x & y) | (y & z) | (z & x));
endmodule

module muxer (m,s1,s2,c);
output m;
input s1,s2,c;
wire f,g,h;
not (f,c);
and (g,s1,c);
and (h,s2,f);
or (m,g,h);
endmodule
```

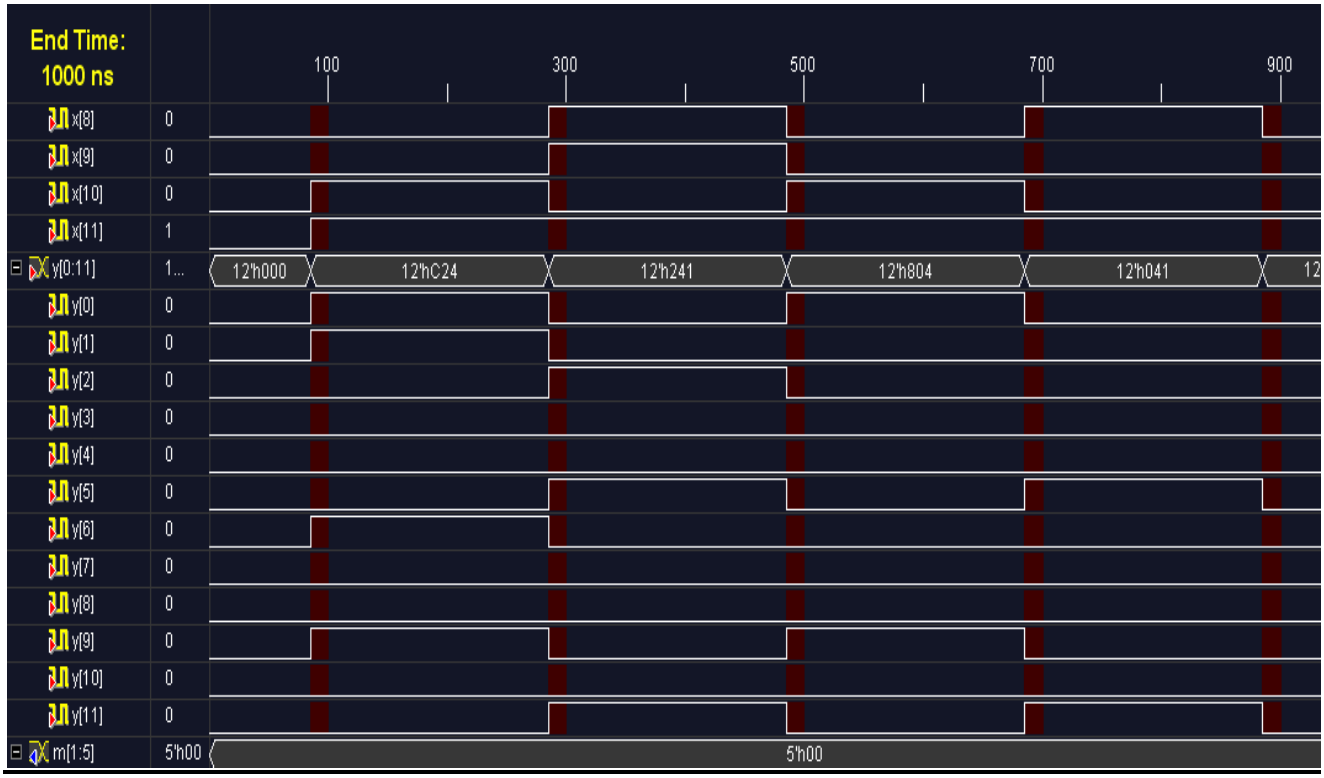
Logic Diagram:



Waveform of carry-select adder:



Test bench waveform of carry-select adder:



RESULT:

Thus the logic circuit for the carry select adder is designed in Verilog HDL and the output is verified.

Expt . No: 5 Date :	# Design of 8 Bit Adders **4.BCD ADDER REALIZATION IN VERILOG HDL**

Aim:

To design a BCD adder circuit using Verilog HDL

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

A BCD adder is the circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. The input digit does not exceed 9,the output sum cannot greater than 9+9+1=19, the 1 in the sum being an input carry. Suppose we apply two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to zero, nothing is added in the binary sum . When it is equal to one, binary 0110 is added to binary sum through the bottom 4-bit binary adder. Output generated from bottom binary adder can be ignored.

The output carry can be expressed in Boolean function

$$k = c4 + s3s2 + s3s1$$

Procedure:

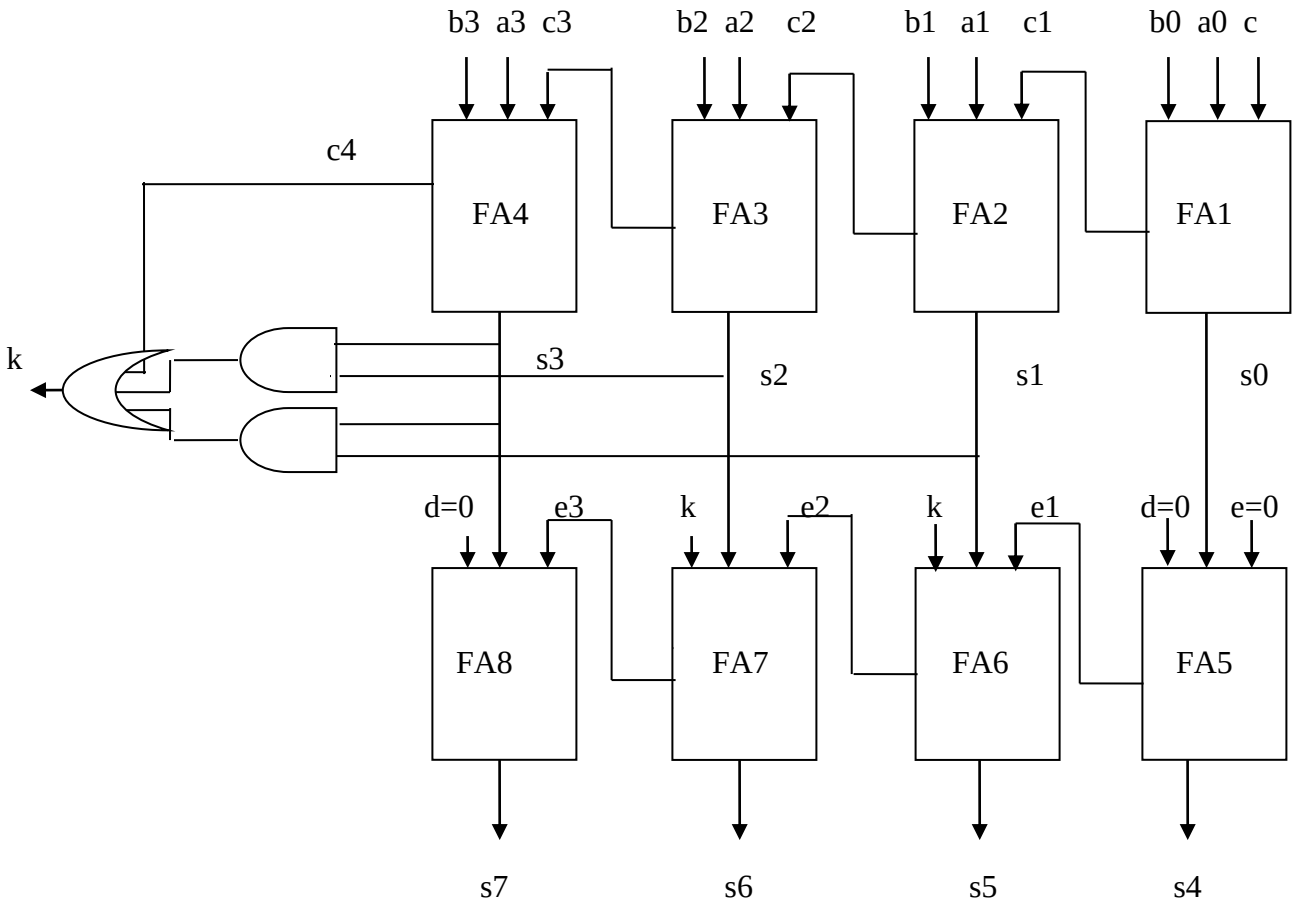
1. The BCD adder circuit is designed and the Boolean function is found out.
2. The VHDL program source code for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

Bcd adder using Verilog :

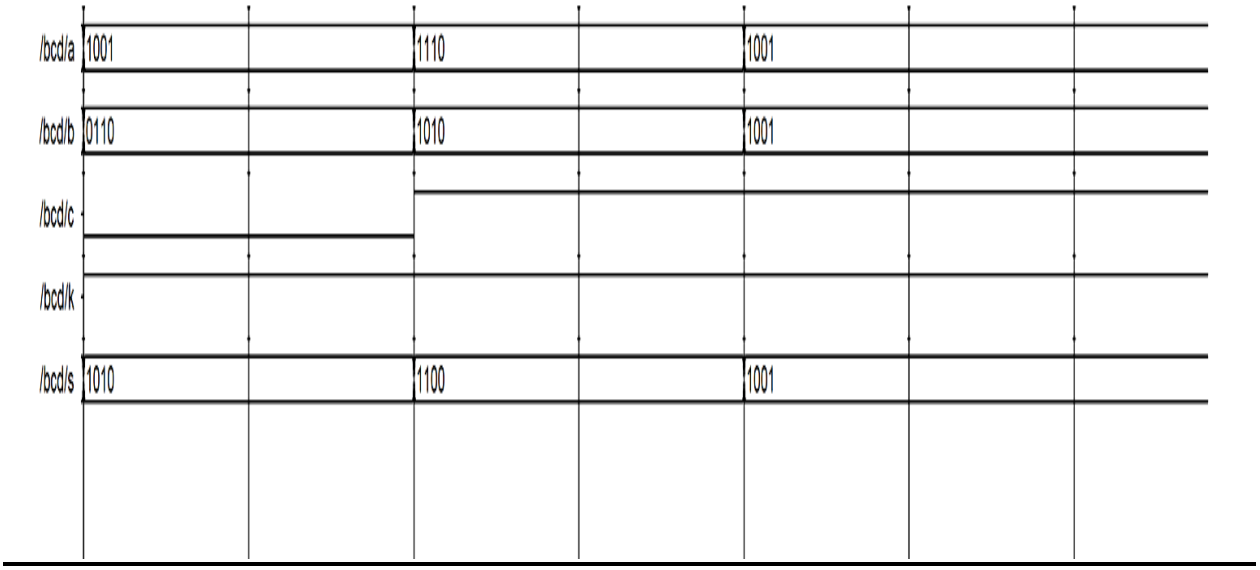
```
module bcdadder(s,k,a,b,c,d,e);
   output [4:7] s;
   inout k;
   input [0:3]a,b;
         input c,d,e;
   wire c1,c2,c3,c4,s0,s1,s2,s3,e1,e2,e3,e4;

fulladder f1(s0,c1,a[0],b[0],c);
fulladder f2(s1,c2,a[1],b[1],c1);
fulladder f3(s2,c3,a[2],b[2],c2);
fulladder f4(s3,c4,a[3],b[3],c3);
```

```
assign k=((s3 & s2) | (s3 & s1)| c4);

fulladder f5(s[4],e1,s0,d,e);
fulladder f6(s[5],e2,s1,k,e1);
fulladder f7(s[6],e3,s2,k,e2);
fulladder f8(s[7],e4,s3,d,e3);
endmodule

module fulladder(s,ca,a,b,c);
output s,ca;
input a,b,c;
xor(s,a,b,c);
assign ca=((a & b)|(b & c)| (c & a));
endmodule
```

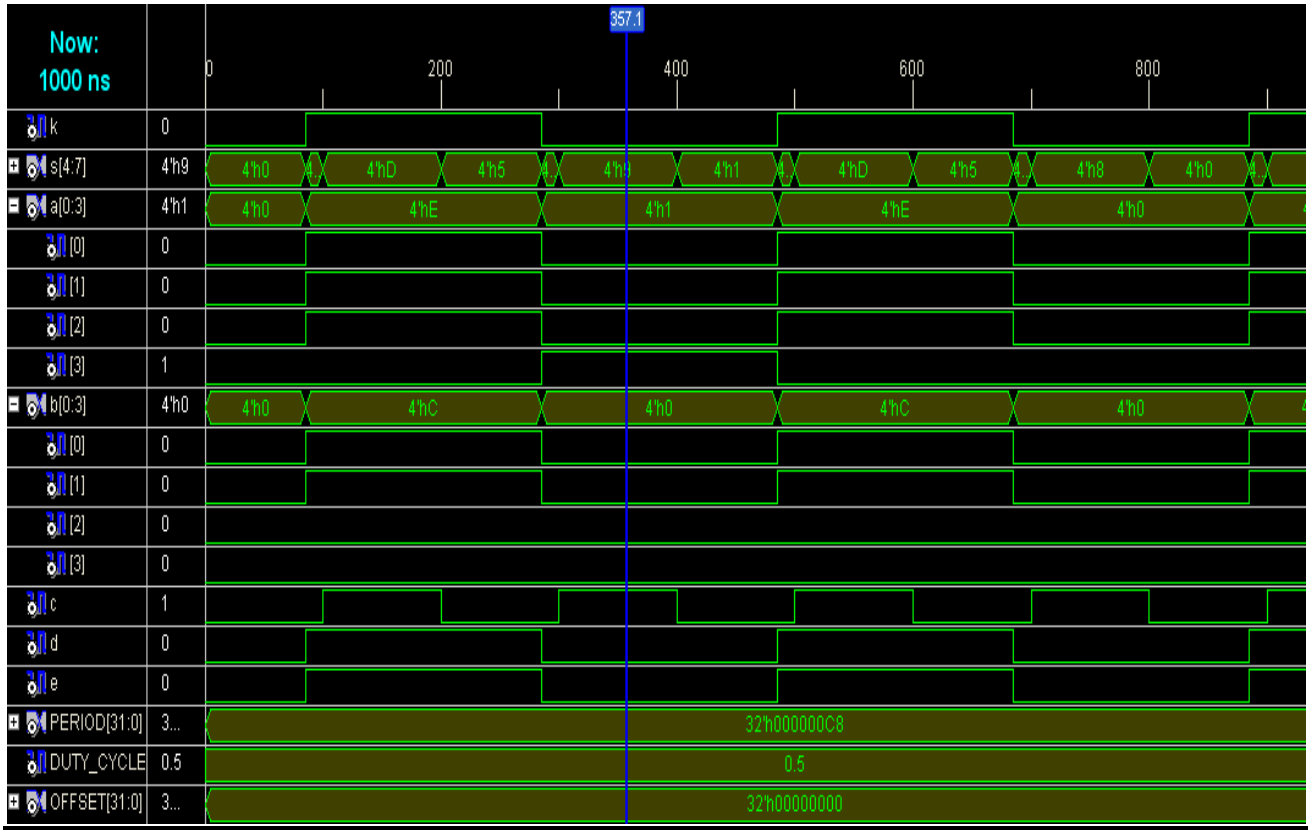
Logic Diagram:



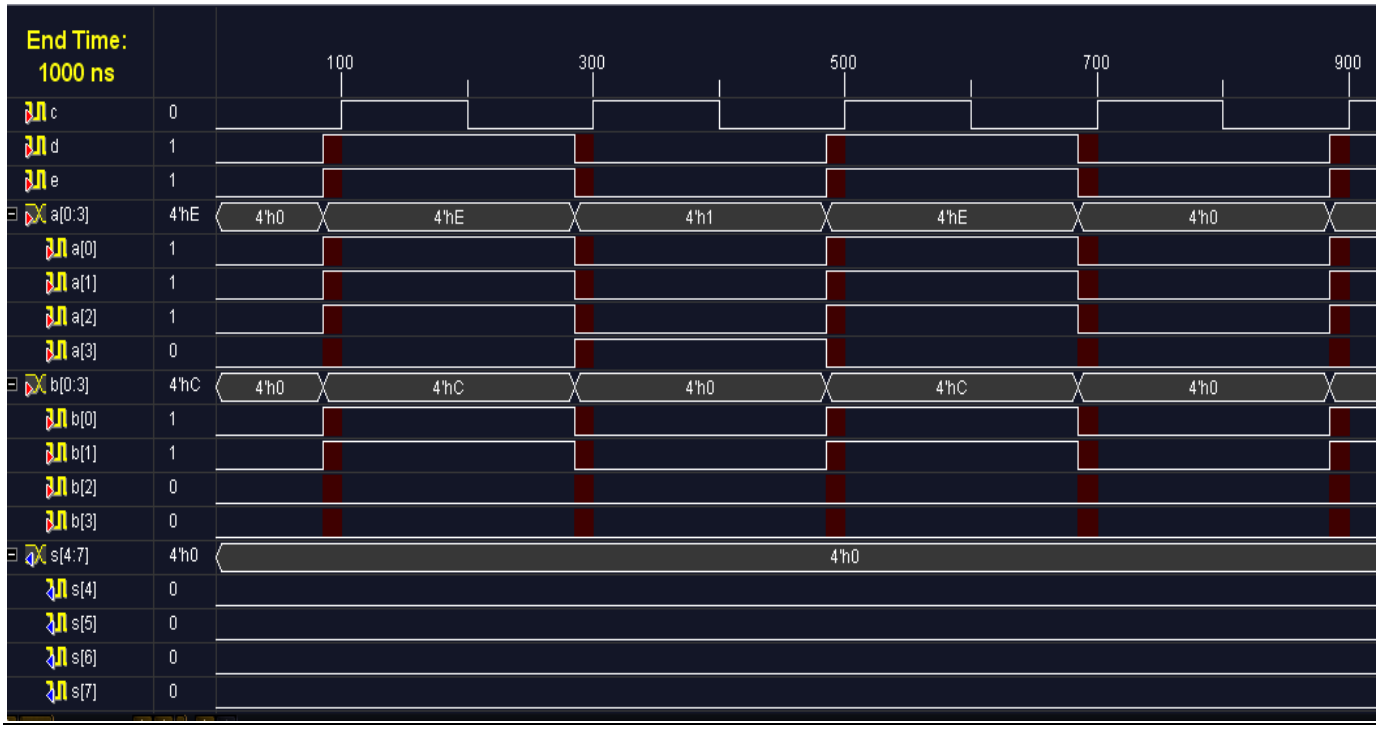
Waveform:



Waveform of bcd adder:



Test bench waveform of bcd adder:



RESULT:

Thus the logic circuit for the BCD adder is designed in Verilog HDL and the output is verified.

Expt . No: 6 Date :	# Design of Multiplexers

Aim:

Design a 4 to 1 multiplexer circuit in Verilog.

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

A digital multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. Multiplexing means transmitting a large number of information units over a smaller number of channels or lines. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected. A multiplexer is also called a data selector, since it selects one of many inputs and steers the binary information to the output lines. Multiplexer ICs may have an enable input to control the operation of the unit. When the enable input is in a given binary state (the disable state), the outputs are disabled, and when it is in the other state (the enable state), the circuit functions as normal multiplexer. The enable input (sometimes called strobe) can be used to expand two or more multiplexer ICs to digital multiplexers with a larger number of inputs.
The size of the multiplexer is specified by the number 2^n of its input lines and the single output line. In general, a 2^n – to – 1 line multiplexer is constructed from an n – to 2^n decoder by adding to it 2^n input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate to provide the 1 – line output.

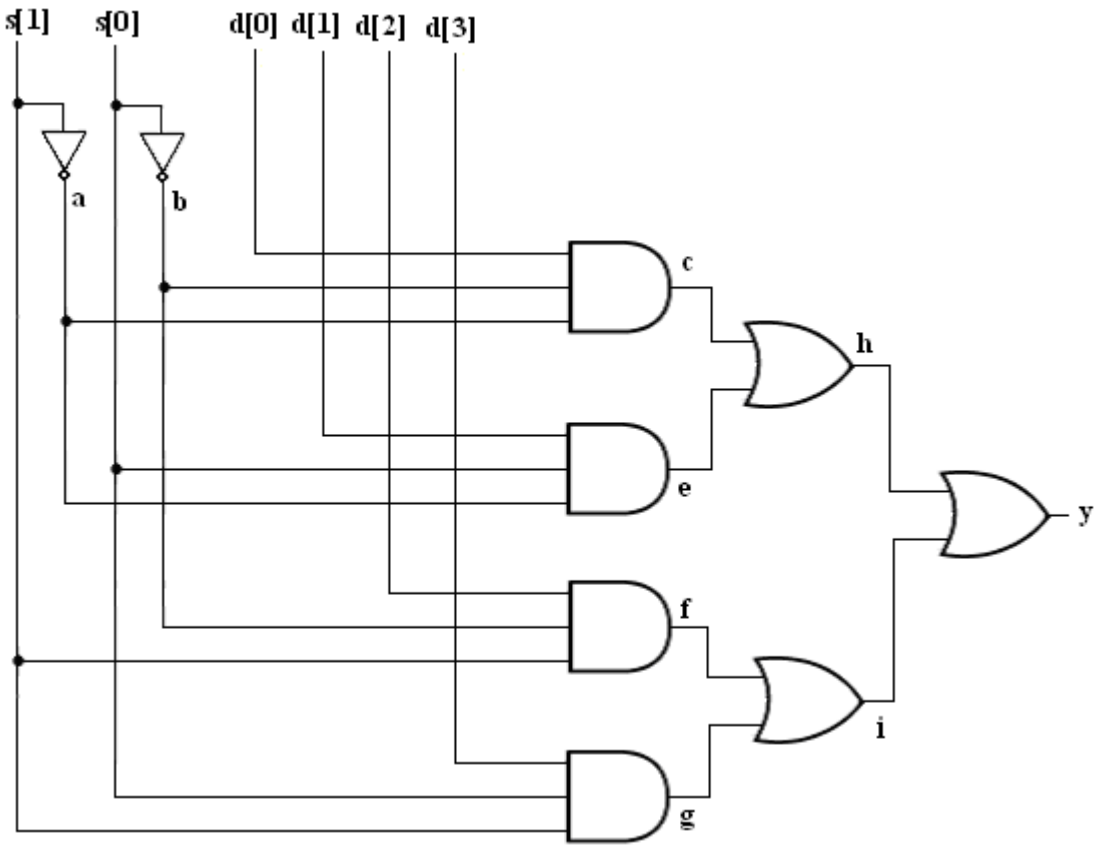
Procedure:

1. The multiplexer circuit is designed and the Boolean function is found out.
2. The Verilog Module Source for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

Truth table:

INPUT		OUTPUT
s[1]	s[0]	y
0	0	D[0]
0	1	D[1]
1	0	D[2]
1	1	D[3]

Logic Diagram:
4 to 1 Multiplexer:

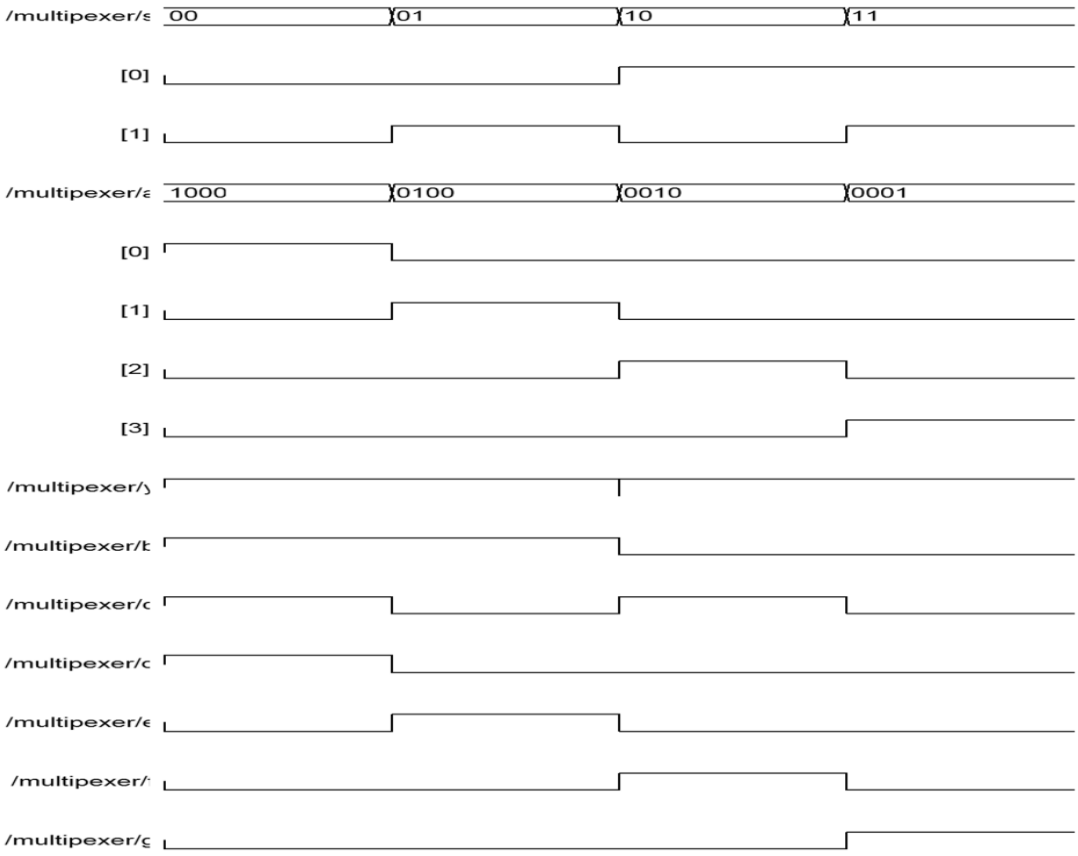


Multiplexer using verilog code:

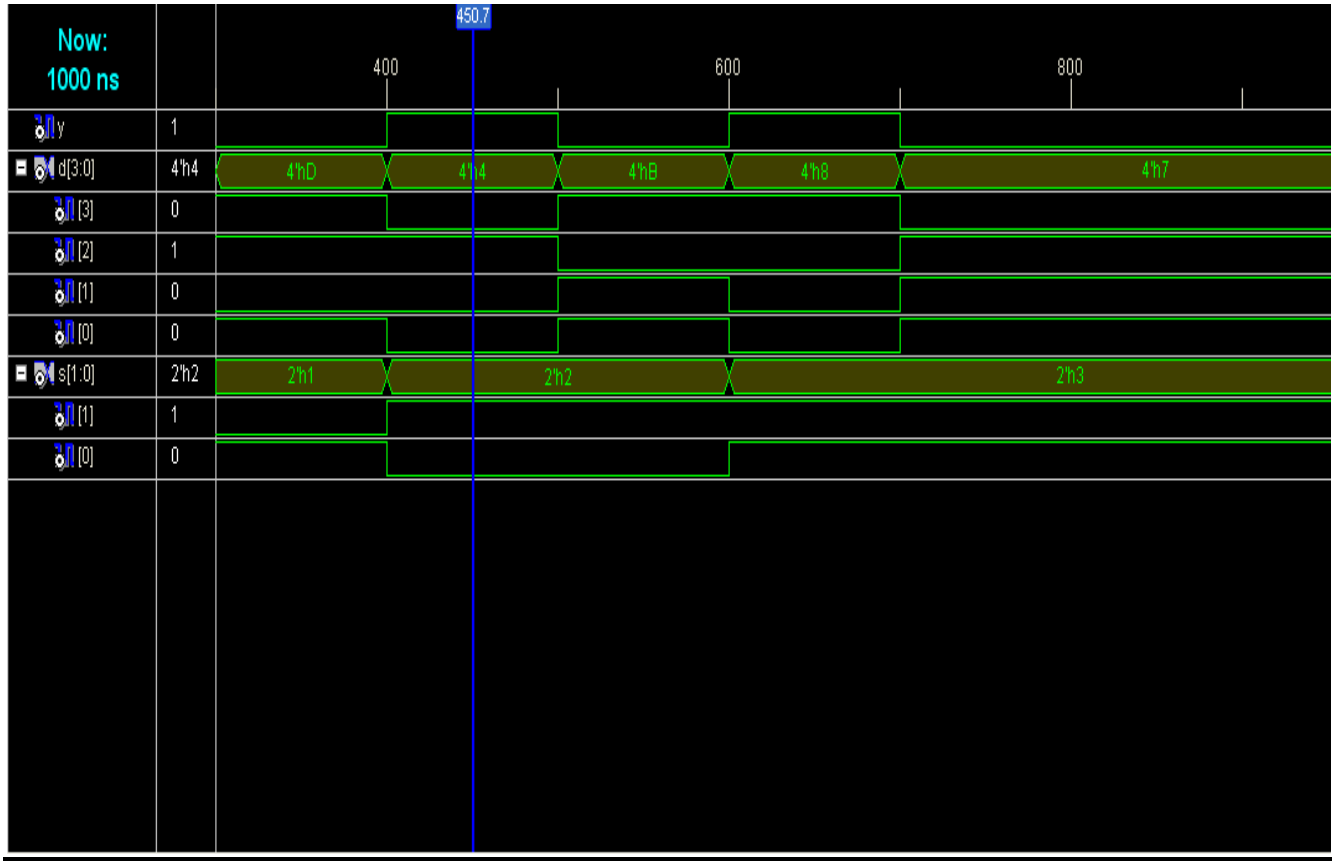
```
module multiplexer(y,d,s);
output y;
input [3:0] d;
input [1:0] s;
wire a,b,c,e,f,g,h,i;
//Instantiate Primitive gates
not (a,s[1]);
not (b,s[0]);
and (c,d[0],b,a);
and (e,d[1],s[0],a);
and (f,d[2],b,s[1]);
and (g,d[3],s[0],s[1]);
or (h,c,e);
or (i,f,g);
or (y,h,i);
endmodule
//Stimulus for testing 4 to 1 Multiplexer
module simulation;
reg [3:0]d;
reg [1:0]s;
wire y;
```

```
//Instantiate the 4 to 1 Multiplexer
multiplexer mux_t(y,d,s);
initial
begin
s=2'b00;d[0]=1'b1;d[1]= 1'b0;d[2]= 1'b0;d[3]= 1'b0;
#100
s=2'b00;d[0]= 1'b0;d[1]= 1'b1;d[2]= 1'b1;d[3]= 1'b1;
#100
s=2'b01;d[0]= 1'b0;d[1]= 1'b1;d[2]= 1'b0;d[3]= 1'b0;
#100
s=2'b01;d[0]= 1'b1;d[1]= 1'b0;d[2]= 1'b1;d[3]= 1'b1;
#100
s=2'b10;d[0]= 1'b0;d[1]= 1'b0;d[2]= 1'b1;d[3]= 1'b0;
#100
s=2'b10;d[0]= 1'b1;d[1]= 1'b1;d[2]= 1'b0;d[3]= 1'b1;
#100
s=2'b11;d[0]= 1'b0;d[1]= 1'b0;d[2]= 1'b0;d[3]= 1'b1;
#100
s=2'b11;d[0]= 1'b1;d[1]= 1'b1;d[2]= 1'b1;d[3]= 1'b0;
end
endmodule
```

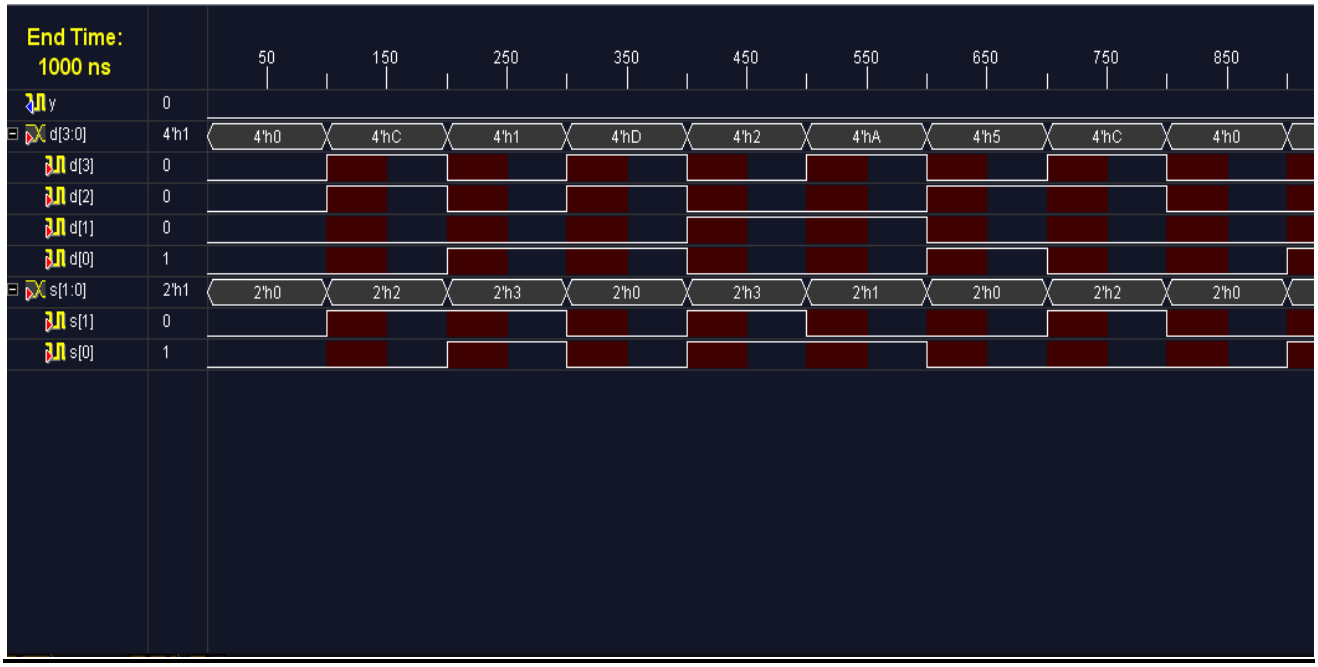
Waveform:



Waveform of multiplexers



Test bench waveform of multiplexers:



RESULT:

Thus the multiplexer is designed in Verilog HDL and the output is verified.

Expt . No: 7 Date :	# Design of Multipliers ## 1. ARRAY MULTIPLIER REALIZATION IN VERILOG HDL

Aim:

To design an array multiplier circuit for 4 inputs and 8 outputs using VHDL.

Apparatus required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

Binary multiplication can be accomplished by several approaches. The approach presented here is realized entirely with combinational circuits. Such a circuit is called an **array multiplier.**

The term array is used to describe the multiplier because the multiplier is organized as an array structure. Each row, called a *partial product*, is formed by a bit-by-bit multiplication of each operand.

For example, a partial product is formed when each bit of operand 'a' is multiplied by b0, resulting in a3b0, a2b0,a1b0, a0b0. The binary multiplication table is identical to the AND truth table.

Each product bit {o(x)}, is formed by adding partial product columns. The product equations, including the carry-in {c(x)}, from column c(x-1), are (the plus sign indicates addition not OR).

Each product term, p(x), is formed by AND gates and collection of product terms needed for the multiplier. By adding appropriate p term outputs, the multiplier output equations are realized, as shown in figure.

4X 4 Array Multiplier:

			a3	a2	a1	a0
			b3	b2	b1	b0
			a3b0	a2b0	a1b0	a0b0
		a3b1	a2b1	a1b1	a0b1	
	a3b2	a2b2	a1b2	a0b2		
a3b3	a2b3	a1b3	a0b3			
o7	o6	o5	o4	o3	o2	o1

a0b0 = p0
a1b0 = p1
a0b1 = p2
a2b0 = p3
a1b2 = p8
a0b3 = p9
a3b1 = p10
a2b2 = p11

a1b1 = p4 a1b3 = p12
a0b2 = p5 a3b2 = p13
a3b0 = p6 a2b3 = p14
a2b1 = p7 a3b3 = p15

Truth Table:

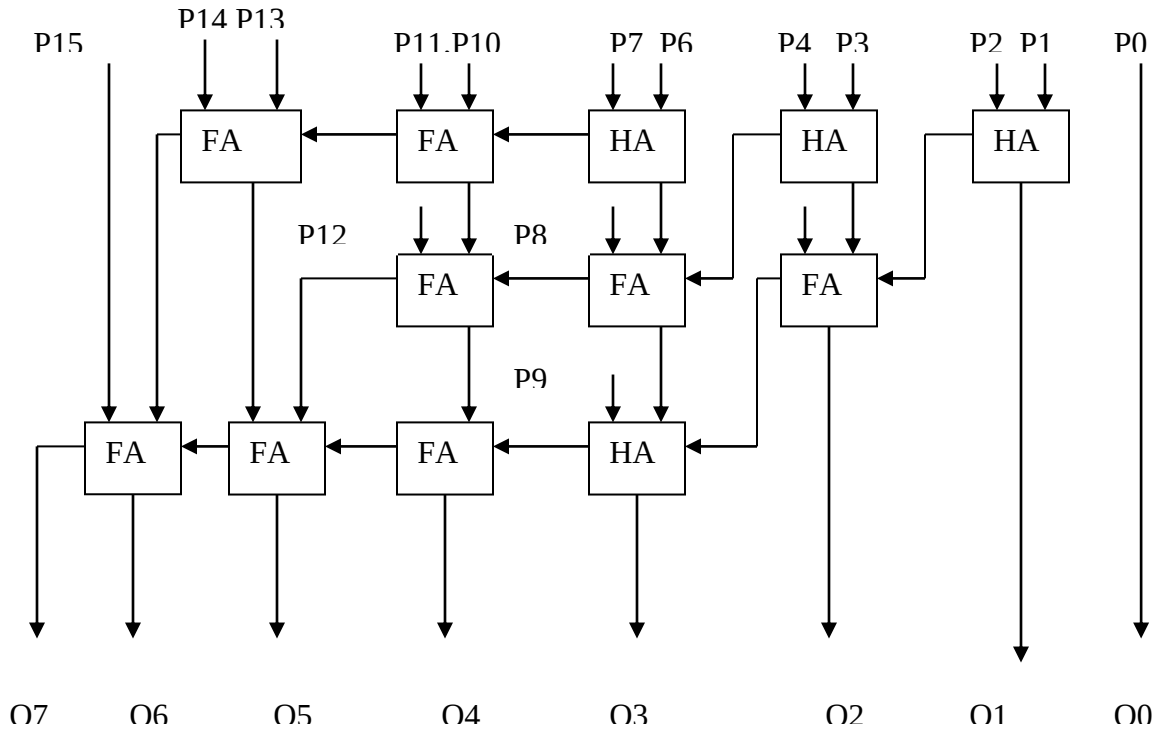
A	B	A X B
0	0	0
0	1	0
1	0	0
1	1	1

Program:

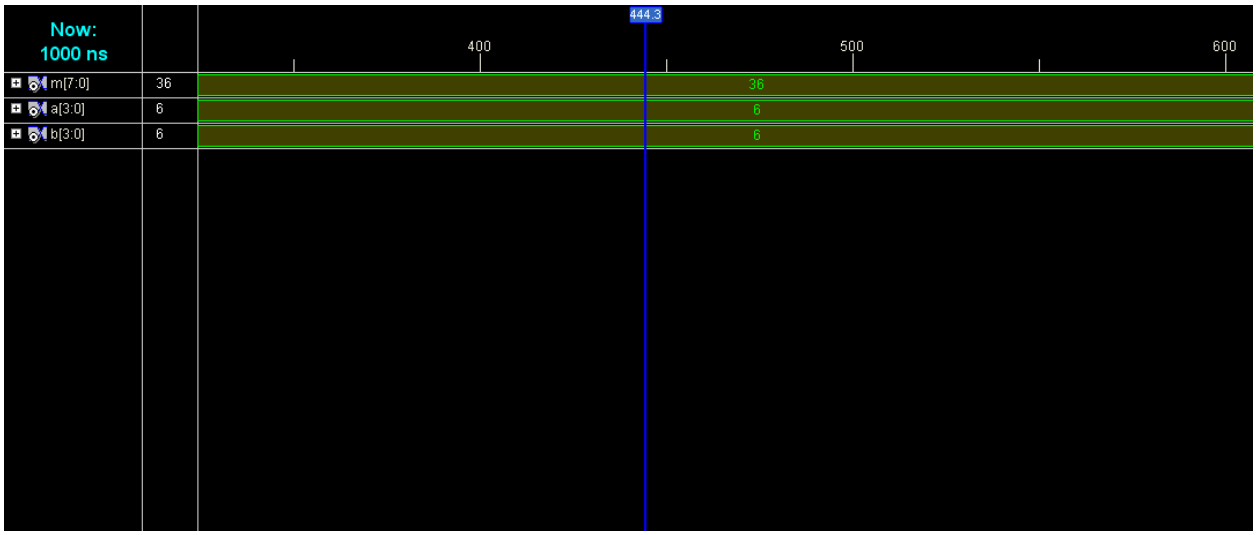
```
module mmmm(m,a,b);

input [3:0]a;
input  [3:0]b;
output [7:0]m;
wire  [15:0]p;
wire [12:1]s;
wire [12:1]c;

and(p[0],a[0],b[0]);
and(p[1],a[1],b[0]);
and(p[2],a[0],b[1]);
and(p[3],a[2],b[0]);
and(p[4],a[1],b[1]);
and(p[5],a[0],b[2]);
and(p[6],a[3],b[0]);
and(p[7],a[2],b[1]);
and(p[8],a[1],b[2]);
and(p[9],a[0],b[3]);
and(p[10],a[3],b[1]);
and(p[11],a[2],b[2]);
and(p[12],a[1],b[3]);
and(p[13],a[3],b[2]);
and(p[14],a[2],b[3]);
and(p[15],a[3],b[3]);
```

```
half ha1(s[1],c[1],p[1],p[2]);
half ha2(s[2],c[2],p[4],p[3]);
half ha3(s[3],c[3],p[7],p[6]);

full fa4(s[4],c[4],p[11],p[10],c[3]);
full fa5(s[5],c[5],p[14],p[13],c[4]);
full fa6(s[6],c[6],p[5],s[2],c[1]);
full fa7(s[7],c[7],p[8],s[3],c[2]);
full fa8(s[8],c[8],p[12],s[4],c[7]);
full fa9(s[9],c[9],p[9],s[7],c[6]);

half ha10(s[10],c[10],s[8],c[9]);
full fa11(s[11],c[11],s[5],c[8],c[10]);
full fa12(s[12],c[12],p[15],s[5],c[11]);

buf(m[0],p[0]);
buf(m[1],s[1]);
buf(m[2],s[6]);
buf(m[3],s[9]);
buf(m[4],s[10]);
buf(m[5],s[11]);
buf(m[6],s[12]);
buf(m[7],c[12]);

endmodule

module half(s,co,x,y);
input x,y;
output s,co;
//Instatiate primitive gates
xor (s,x,y);
and (co,x,y);
endmodule

//Description of Full Adder
module full(s,co,x,y,ci);
input x,y,ci;
output s,co;
wire s1,d1,d2;    //Outputs of first XOR and AND gates
//Instantiate Half Adder
half ha_1(s1,d1,x,y);
half ha_2(s,d2,s1,ci);
or or_gate(co,d2,d1);
endmodule
```

Logic Diagram:



Wave Form:



RESULT:

Thus an array multiplier circuit for 4 inputs and 8 outputs using VHDL is designed and the output is verified.

Expt . No: 8 Date :	# Design of Multipliers ## 2. BRAUN MULTIPLIER REALIZATION IN VERILOG HDL

Aim:

To design the Braun multiplier in verilog HDL

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

The entire partial product A.bk are computed in parallel, and then collected through a cascaded array of carry save order. At the bottom of the array, an adder is used to convert the carry save from the required form of output.

Completion time is fixed by the depth of the array, and by the carry propagation characteristics of the adder. In multiplier is suited only to positive operands.

4 X 4 Braun Multiplier:

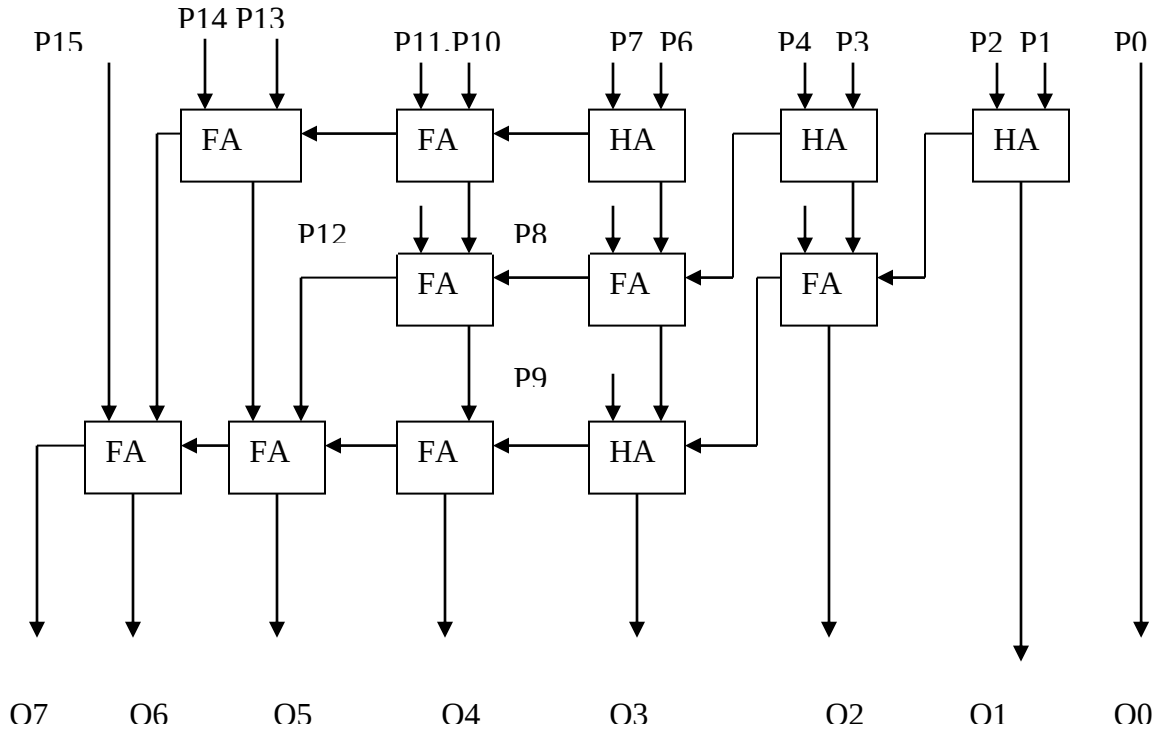
			a3	a2	a1	a0
			b3	b2	b1	b0
			a3b0	a2b0	a1b0	a0b0
		a3b1	a2b1	a1b1	a0b1	
	a3b2	a2b2	a1b2	a0b2		
a3b3	a2b3	a1b3	a0b3			
o7	o6	o5	o4	o3	o2	o1

a0b0 = p0
a1b0 = p1
a0b1 = p2
a2b0 = p3
a1b1 = p4
a0b2 = p5
a3b0 = p6
a2b1 = p7
a1b2 = p8
a0b3 = p9
a3b1 = p10
a2b2 = p11
a1b3 = p12
a3b2 = p13
a2b3 = p14
a3b3 = p15

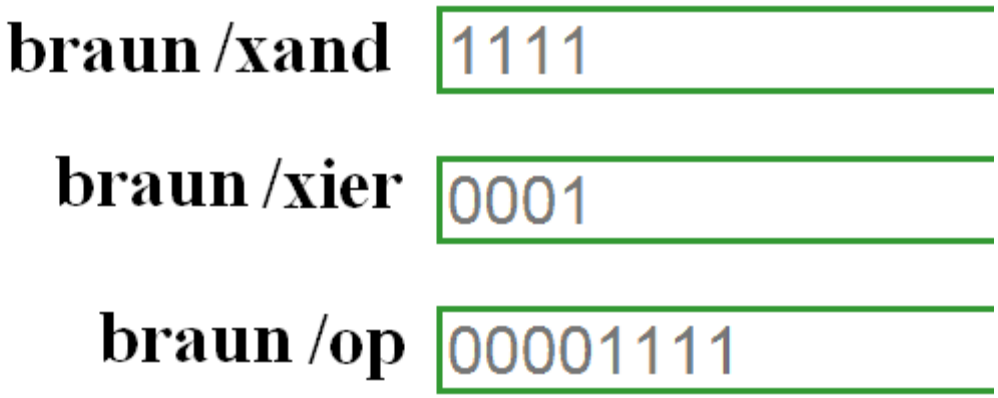
Truth Table:

A	B	A X B
0	0	0
0	1	0
1	0	0
1	1	1

Logic Diagram:



Wave Form:



RESULT:

Thus an Braun multiplier in verilog HDL is designed and the output is verified.

Expt . No: 9 Date :	# Design of Counters **1. RIPPLE COUNTER REALIZATION IN VERILOG HDL**

Aim:

To realize an asynchronous ripple counter in Verilog

Apparatus required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

In a ripple counter, the flip-flop output transition serves as a source for triggering other flip-flops. In other words, the Clock Pulse inputs of all flip-flops (except the first) are triggered not by the incoming pulses, but rather by the transition that occurs in other flip-flops. A binary ripple counter consists of a series connection of complementing flip-flops (JK or T type), with the output of each flip-flop connected to the Clock Pulse input of the next higher-order flip-flop. The flip-flop holding the LSB receives the incoming count pulses. All J and K inputs are equal to 1. The small circle in the Clock Pulse /Count Pulse indicates that the flip-flop complements during a negative-going transition or when the output to which it is connected goes from 1 to 0. The flip-flops change one at a time in rapid succession, and the signal propagates through the counter in a ripple fashion. A binary counter with reverse count is called a binary down-counter. In binary down-counter, the binary count is decremented by 1 with every input count pulse.

Procedure:

1. The 4 bit asynchronous ripple counter circuit is designed.
2. The Verilog Module Source for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

//Structural description of Ripple Counter

```
module ripplecounter(A0,A1,A2,A3,COUNT,RESET);
output A0,A1,A2,A3;
input COUNT,RESET;
//Instantiate  Flip-Flop
FF F0(A0,COUNT,RESET);
FF F1(A1,A0,RESET);
FF F2(A2,A1,RESET);
FF F3(A3,A2,RESET);
endmodule
```

//Description of Flip-Flop

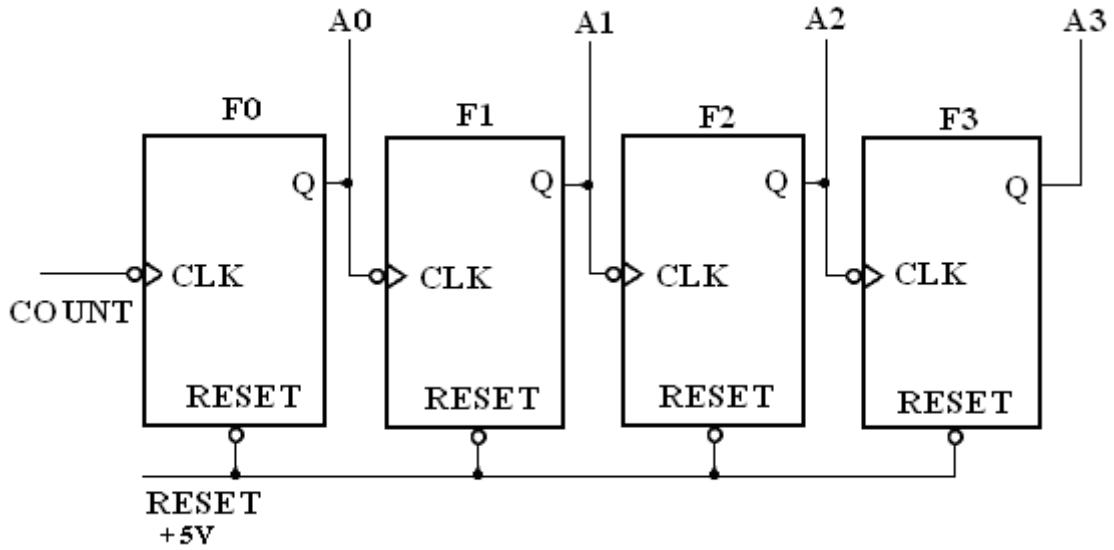
```
module FF(Q,CLK,RESET);
output Q;
input CLK,RESET;
reg Q;
always @(negedge CLK or negedge RESET)
if(~RESET)
Q=1'b0;
else
Q=(~Q);
endmodule
```

//Stimulus for testing Ripple Counter

```
module simulation;
reg COUNT;
reg RESET;
wire A0,A1,A2,A3;

//Instantiate Ripple Counter
ripplecounter rc_t(A0,A1,A2,A3,COUNT,RESET);
always
#5 COUNT=~COUNT;
initial
begin
COUNT=1'b0;
RESET=1'b0;
#10 RESET=1'b1;
end
endmodule
```

LOGIC DIAGRAM:

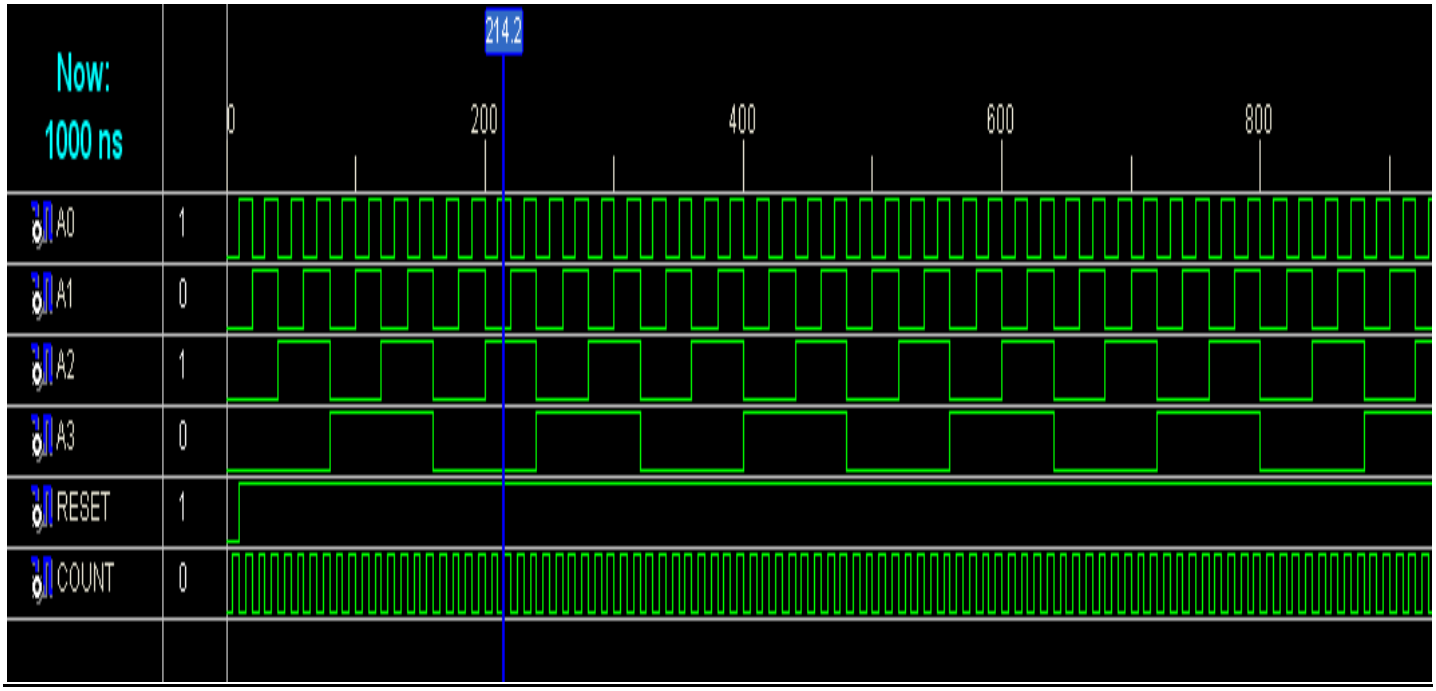
4-Bit Ripple Counter:



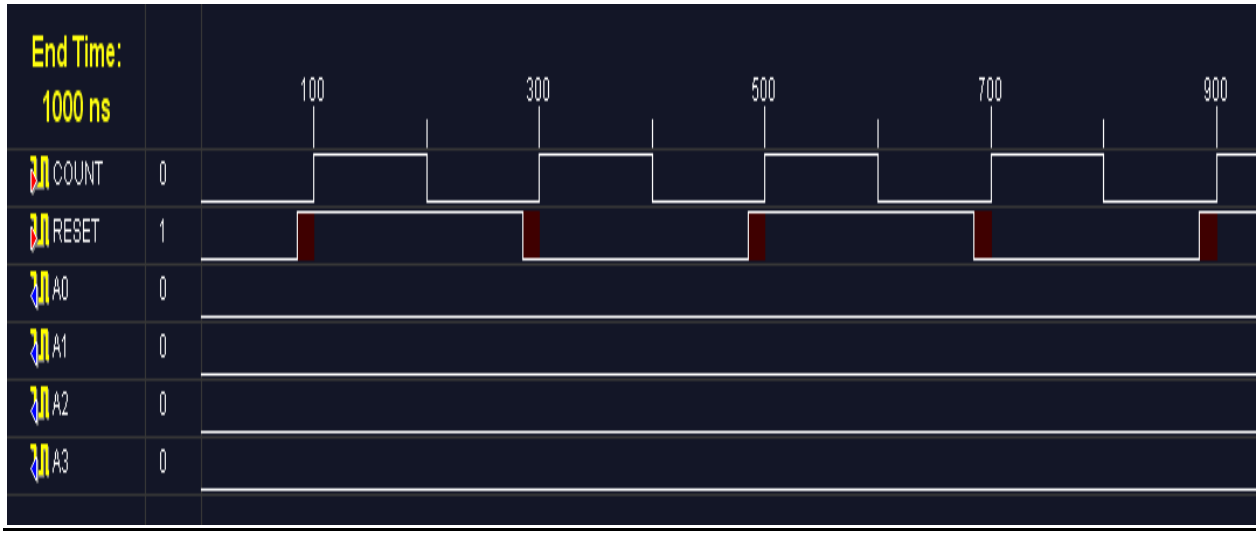
TRUTH TABLE:

COUNT	A0	A1	A2	A3
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	0
5	1	0	1	0
6	0	1	1	0
7	1	1	1	0
8	0	0	0	1
9	1	0	0	1
10	0	1	0	1
11	1	1	0	1
12	0	0	1	1
13	1	0	1	1
14	0	1	1	1
15	1	1	1	1

Waveform of ripple counter:

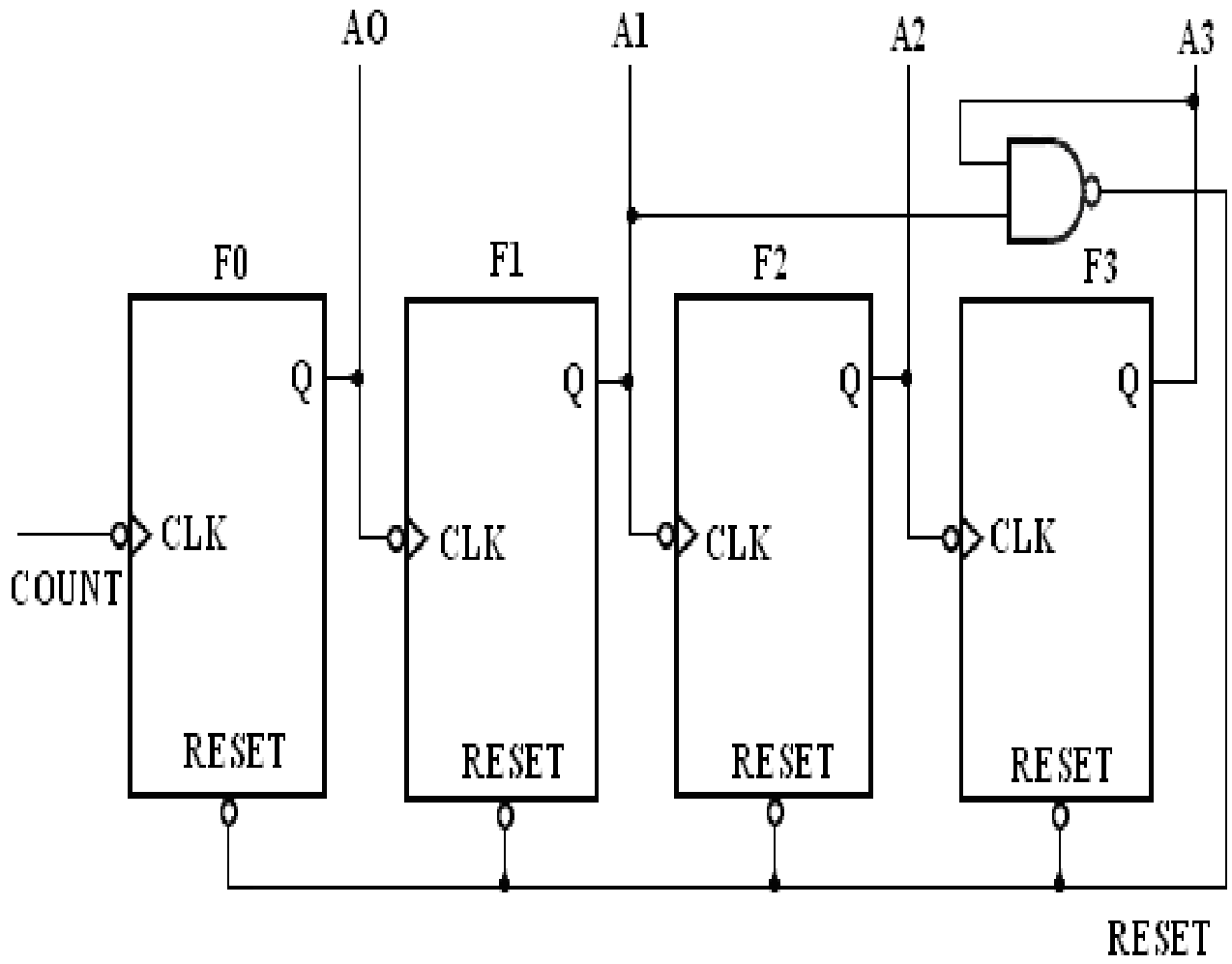


Testbenchwaveform of ripple counter:



LOGIC DIAGRAM:

MOD-10 Ripple Counter:



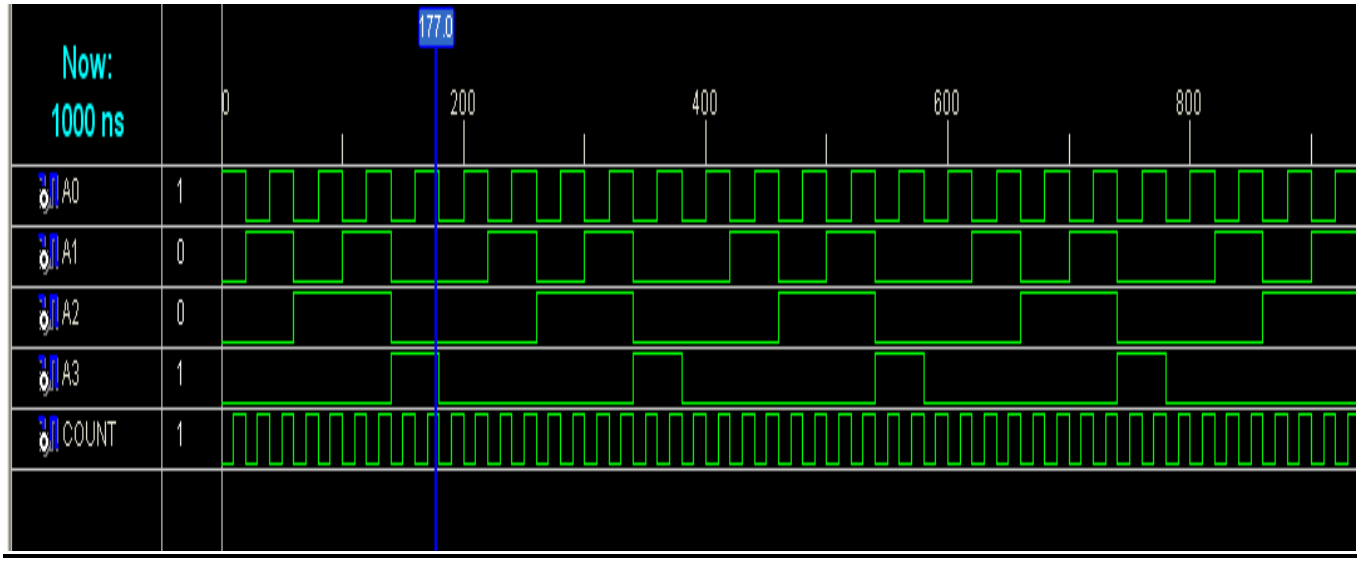
TRUTH TABLE:

COUNT	A0	A1	A2	A3
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	0
5	1	0	1	0
6	0	1	1	0
7	1	1	1	0
8	0	0	0	1
9	1	0	0	1
10	0	0	0	0

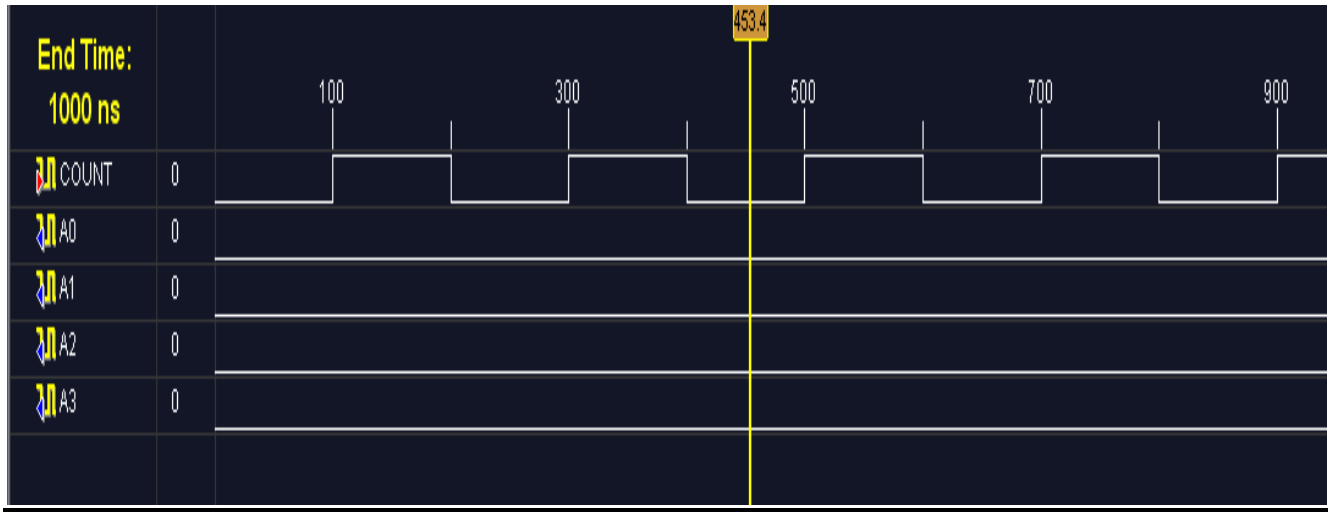
```
/Structural description of MOD10 Counter
module MOD10(A0,A1,A2,A3,COUNT);
output A0,A1,A2,A3;
input COUNT;
wire RESET;
//Instantiate Flip-Flop
FF F0(A0,COUNT,RESET);
FF F1(A1,A0,RESET);
FF F2(A2,A1,RESET);
FF F3(A3,A2,RESET);
//Instantiate Primitive gate
nand (RESET,A1,A3);
endmodule

//Description of Flip-Flop
module FF(Q,CLK,RESET);
output Q;
input CLK,RESET;
reg Q=1'b0;
always @(negedge CLK or negedge RESET)
if(~RESET)
Q=1'b0;
else
Q=(~Q);
endmodule
```

```
//Stimulus for testing MOD10 Counter
module simulation;
reg COUNT;
wire A0,A1,A2,A3;
//Instantiate MOD10 Counter
MOD10 MOD10_TEST(A0,A1,A2,A3,COUNT);
always
#10 COUNT=~COUNT;
initial
begin
COUNT=1'b0;
end
endmodule
```

Waveform of mod 10:

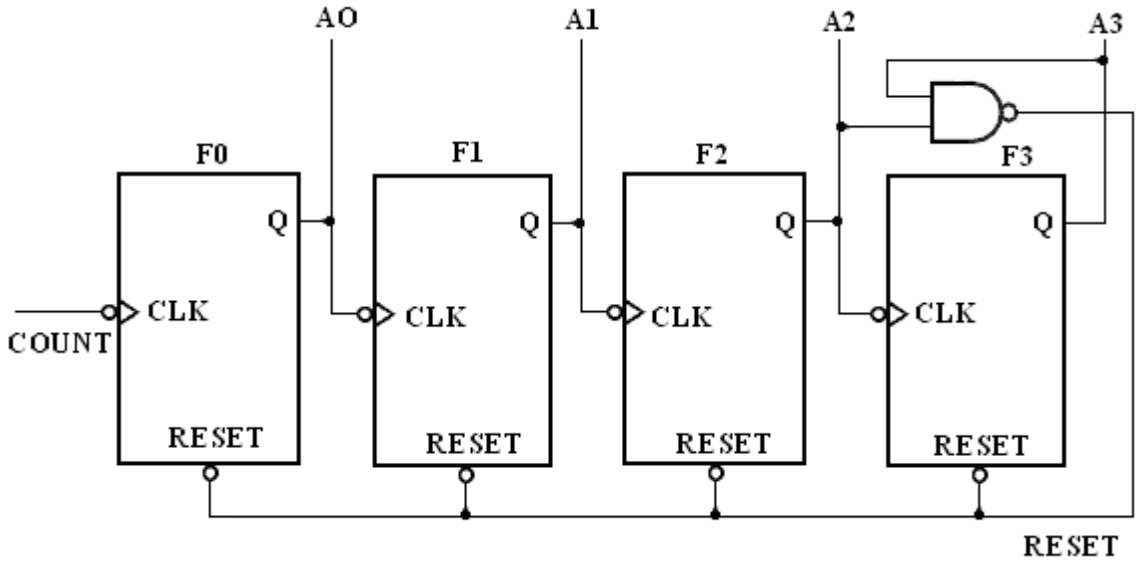


Testbenchwaveform of mod 10:



LOGIC DIAGRAM:

MOD-12 Ripple Counter:

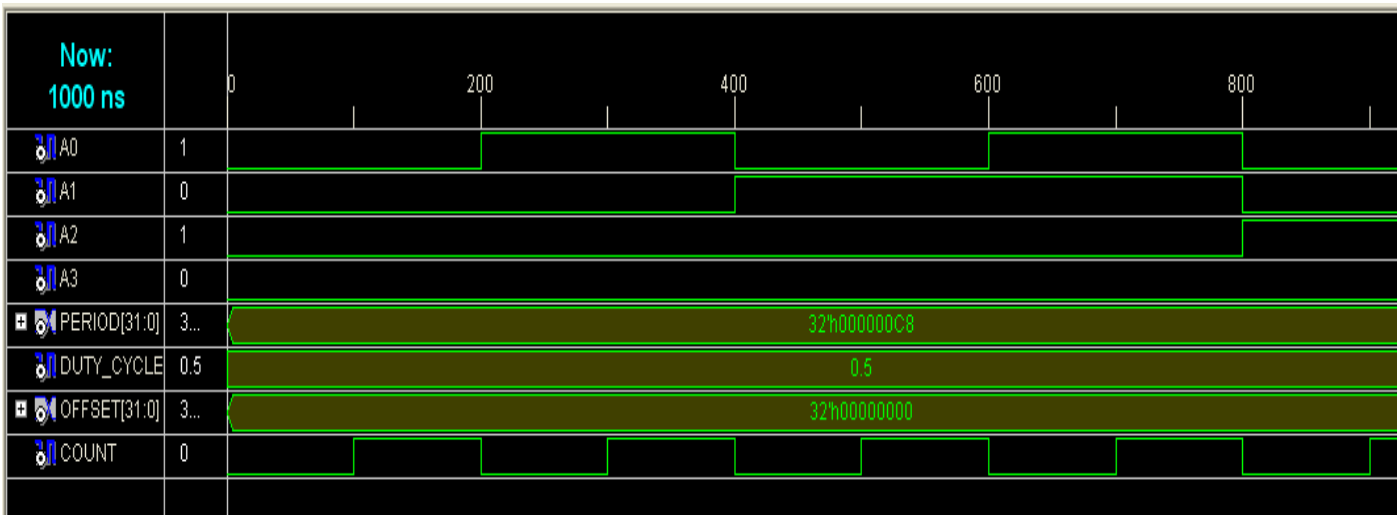


TRUTH TABLE:

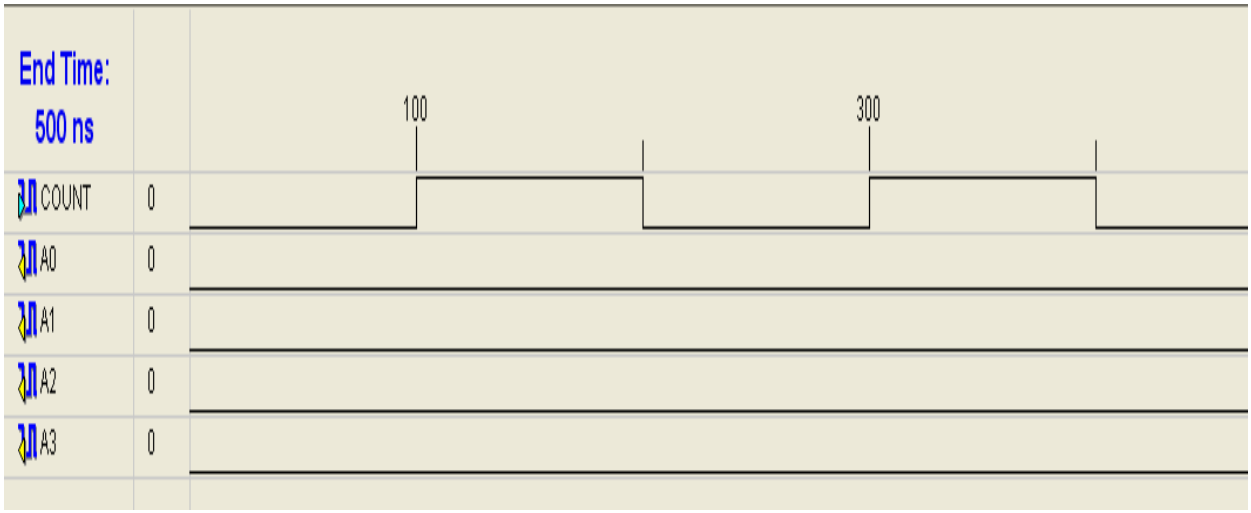
COUNT	A0	A1	A2	A3
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	0
5	1	0	1	0
6	0	1	1	0
7	1	1	1	0
8	0	0	0	1
9	1	0	0	1
10	0	1	0	1
11	1	1	0	1
12	0	0	0	0

```
//Structural description of MOD12 Counter

module MOD12(A0,A1,A2,A3,COUNT);
output A0,A1,A2,A3;
input COUNT;
wire RESET;
//Instantiate Flip-Flop
FF F0(A0,COUNT,RESET);
FF F1(A1,A0,RESET);
FF F2(A2,A1,RESET);
FF F3(A3,A2,RESET);
//Instantiate Primitive gates
nand (RESET,A2,A3);
endmodule

//Description of Flip-Flop

module FF(Q,CLK,RESET);
output Q;
input CLK,RESET;
reg Q=1'b0;
always @(negedge CLK or negedge RESET)
if(~RESET)
Q=1'b0;
else
Q=(~Q);
endmodule

//Stimulus for testing MOD12 Counter

module simulation;
reg COUNT;
wire A0,A1,A2,A3;
//Instantiate MOD12 Counter
MOD12 MOD12_TEST(A0,A1,A2,A3,COUNT);
always
#10 COUNT=~COUNT;
initial
begin
COUNT=1'b0;
end
endmodule
```

Waveform of mod 12 counter :



Testbenchwaveform of mod 12 counter:



RESULT:

Thus the ripple counter is designed in Verilog HDL and the output is verified.

Expt . No: 10 Date :	# Design of Counters **2. RING COUNTER REALIZATION IN VERILOG HDL**

AIM:

To realize a ring counter in Verilog and VHDL.

Apparatus required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

A ring counter is a circular shift register with only one flip-flop being set at ay particular time; all others are cleared. The single bit is shifted from one flip-flop tot the other to produced the sequence of timing signals.

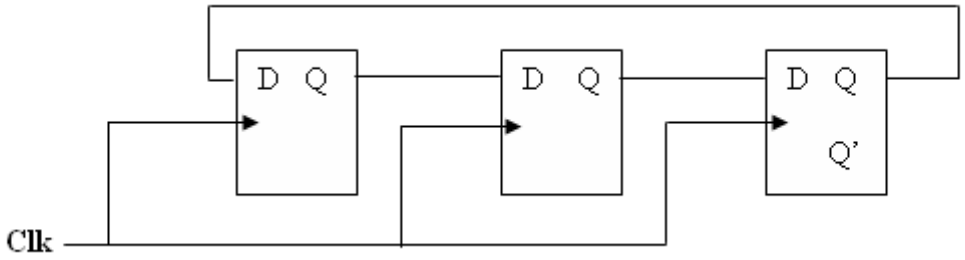
Procedure:

1. The 4 bit ring counter circuit is designed.
2. The Verilog Module Source for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

Binary Ring Counter Design in Verilog

```
module my_ringcntvlog (q,clk,reset);
output [0 : 3]q;
input clk,reset;
reg [0 : 3] q;
always @ (negedge clk or reset)
begin
        if (~reset)
                q = 4'b 1000;
        else if (reset)
        begin
                q[0] <= q[3];
                q[1] <= q[0];
                q[2] <= q[1];
                q[3]    <= q[2];
        end
end
endmodule
```

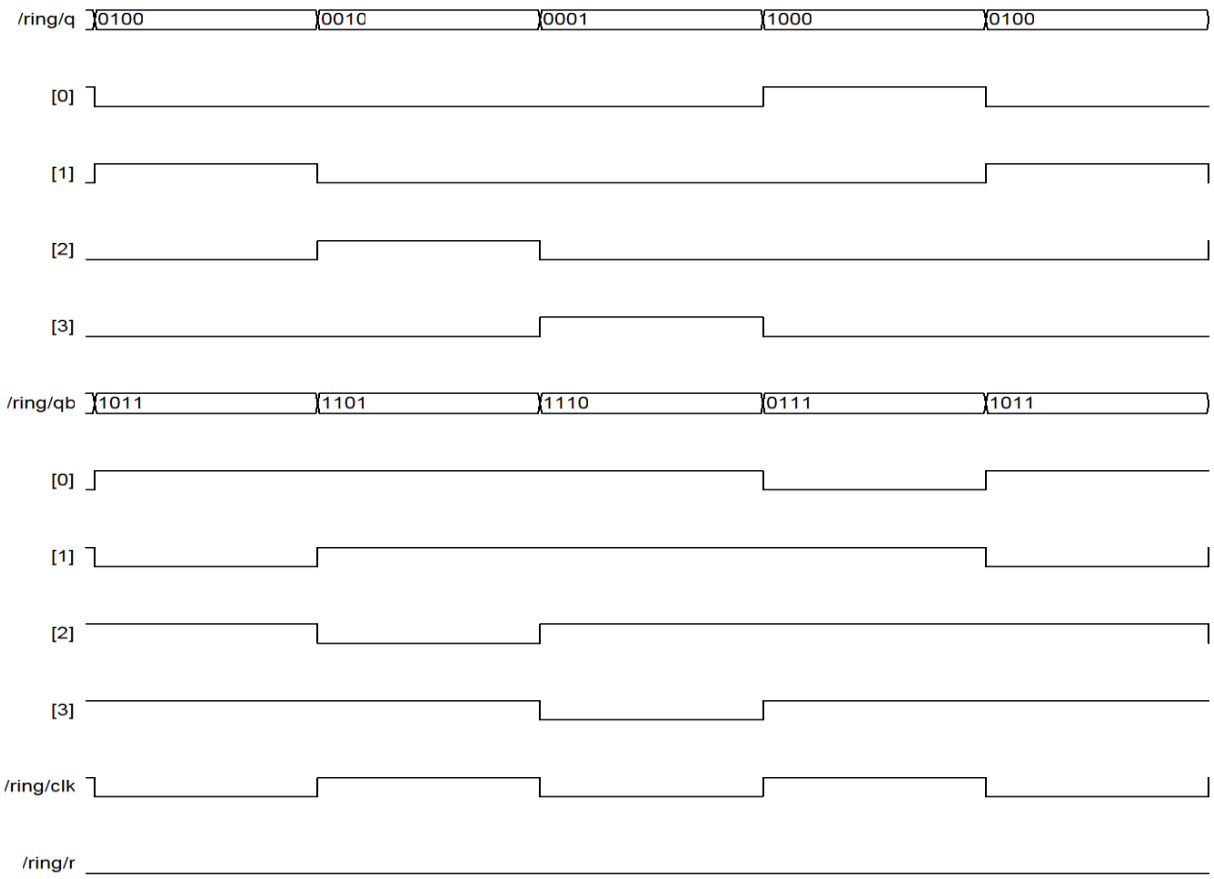
Logic Diagram:



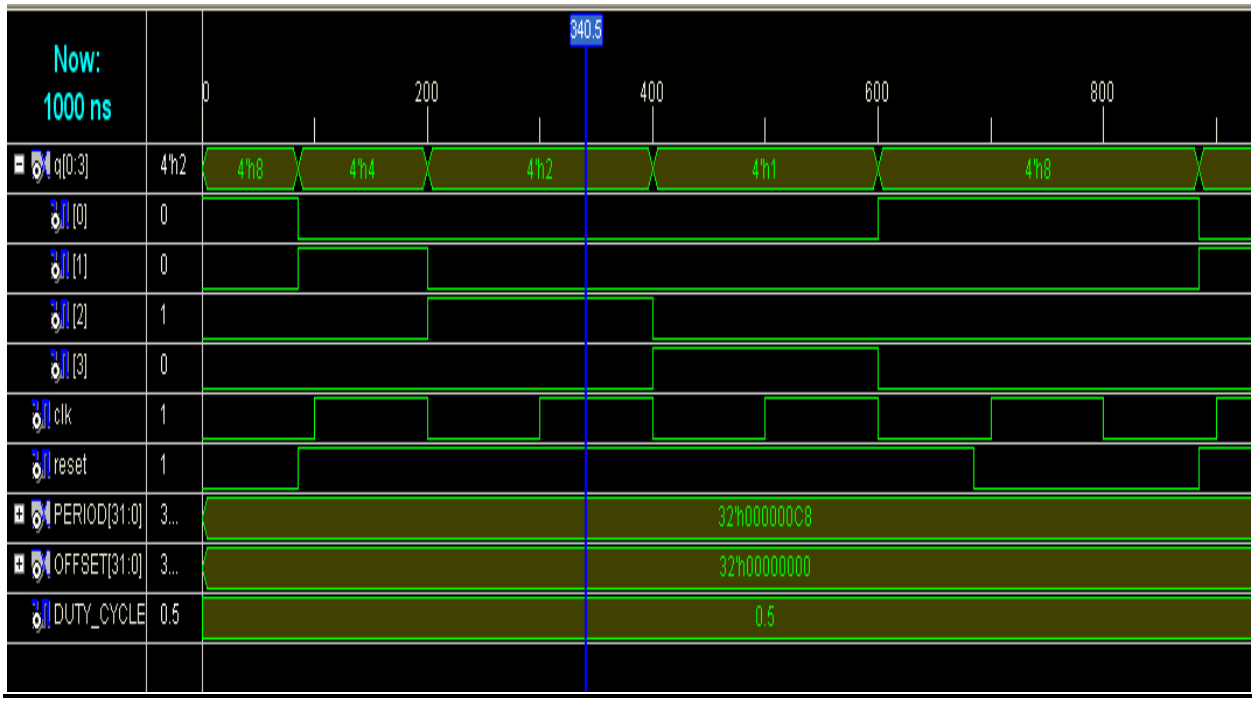
Truth Table:

Input		Output		
Clk	Reset	Qa	Qb	Qc
1	1	1	0	0
1	0	0	1	0
1	0	0	0	1
1	0	1	0	0
1	0	0	1	0

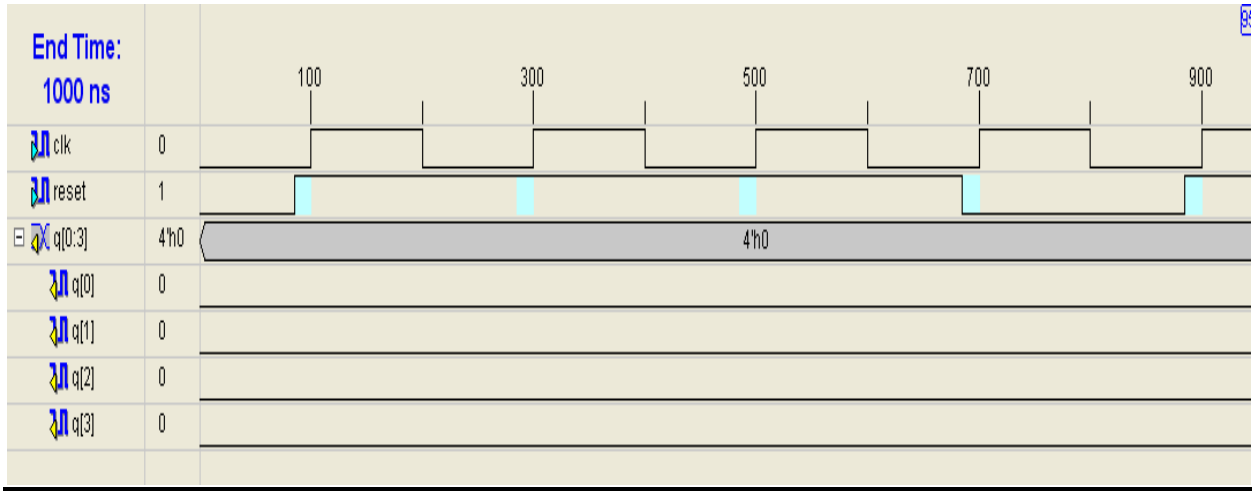
Waveforms



Waveform of ring counter:



Test bench waveform of ring counter:



RESULT:

Thus the ring counter is designed in Verilog HDL and the output is verified.

Expt . No: 11 Date :	**Design of Pseudo Random Binary Sequence**

Aim:

Realize the parity generator in Verilog HDL

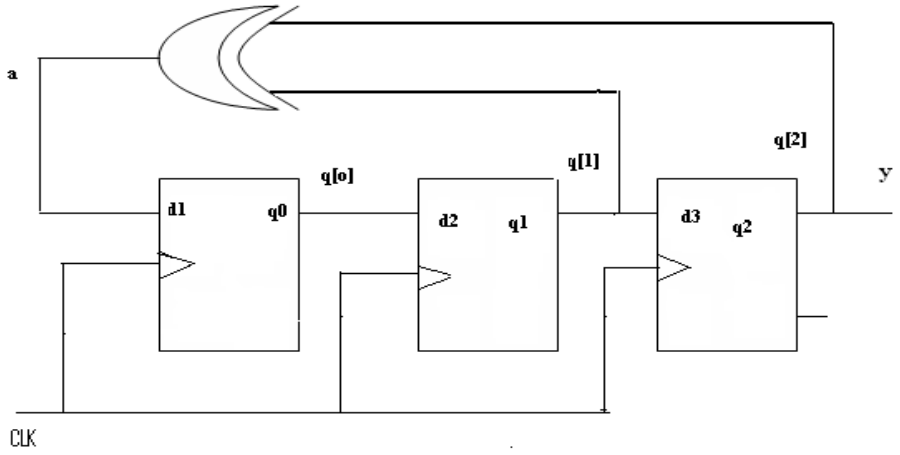
Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator;

Theory:

Random numbers for polynomial equations are generated by using the shift register circuit. The random number generator is nothing but the Linear Feedback Shift Register(LFSR). The shift registers are very helpful and versatile modules that facilitate the design of many sequential circuits whose design may otherwise appear very complex. In its simplest form, a shift register consists of a series of flip-flops having identical interconnection between two adjacent flip-flops. Two such registers are shift right registers and the shift left registers. In the shift right register, the bits stored in the flip-flops shift to the right when shift pulse is active. Like that, for a shift left register, the bits stored in the flip-flops shift left when shift pulse is active. In the shift registers, specific patterns are shifted through the register. There are applications where instead of specific patterns, random patterns are more important. Shioft registers can also built to generate such patterns , which are pseudorandom in nature. Called Linear Feedback Shift Registers (LFSR's), these are very useful for encoding and decoding the error control codes. LFSRs used as a generators of pseudorandom sequences have proved externally useful in the area of testing of VLSI chips.

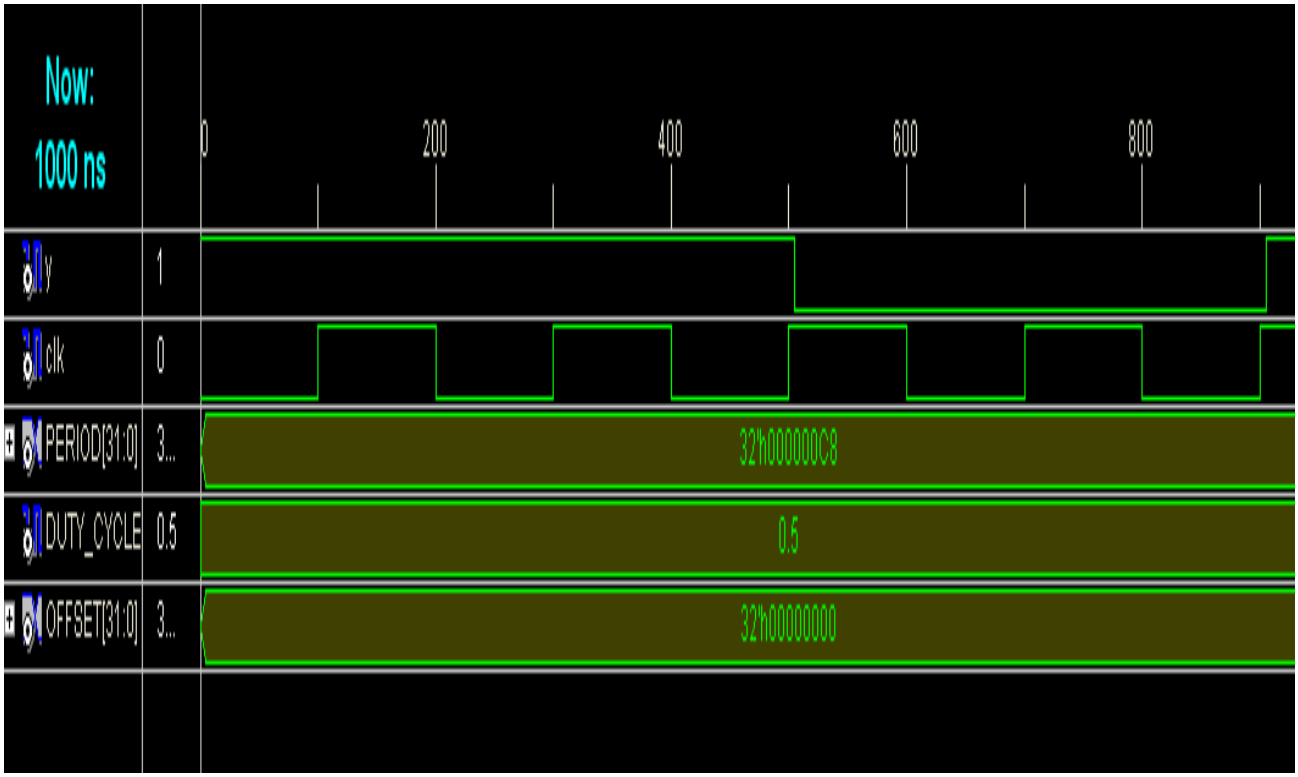
Circuit diagram:



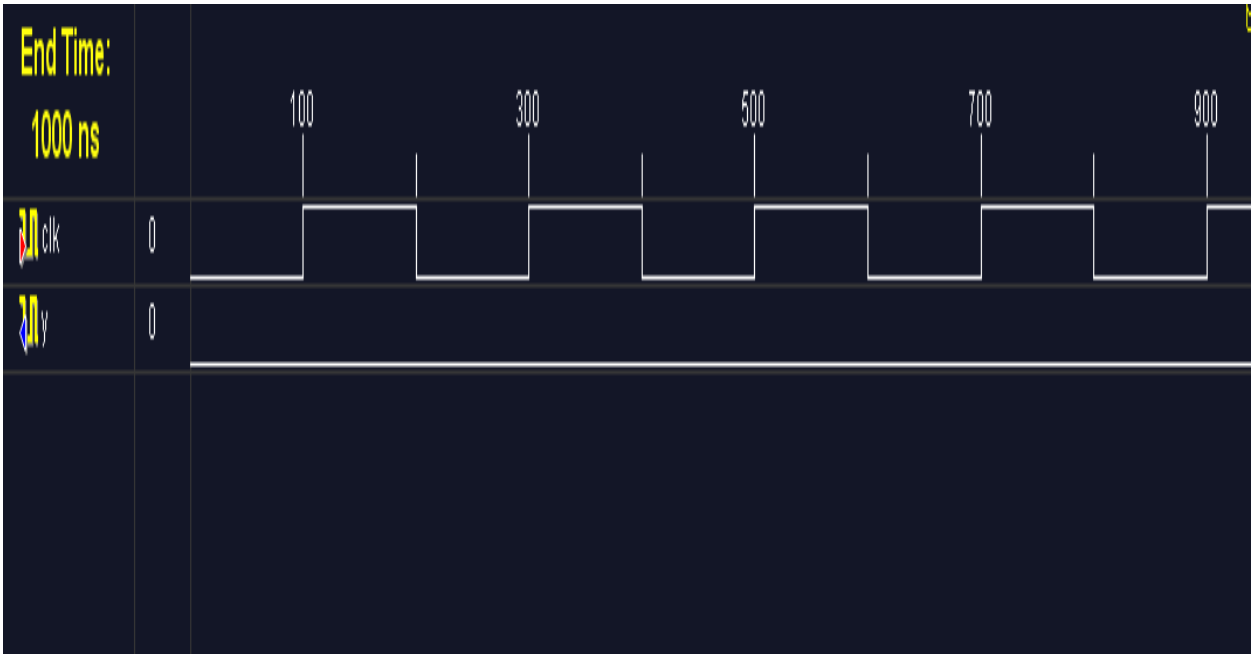
Verilog code

```
 module (y,clk);
output y;
input clk;
wire [1:0]q;
wire a;
dff df1(q[0],a,clk);
dff  df2(q[1],q[0],clk;
dff  df3(y,q[1],clk);
xor(a,y,q[1]);
endmodule

module dff(q,d,clk);
output q;
input clk;
input d,clk;
reg q=1'b0;
always@(posedge clk)
q=#5d;
endmodule
```

Waveform of prbs:



Testbenchwaveform of prbs :



RESULT:

Thus the parity generator is designed in Verilog HDL and the output is verified.

Expt . No: 12 Date :	# Design of Accumulator

Aim:

Realize the accumulator in Verilog HDL

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator

Theory:

An accumulator differs from a counter in the nature of the operands of the add and subtract operation:
• In a counter, the destination and first operand is a signal or variable and the other operand is a constant equal to 1: A <= A + 1.
• In an accumulator, the destination and first operand is a signal or variable, and the second operand is either:
♦ A signal or variable: A <= A + B
♦ A constant not equal to 1: A <= A + Constant
An inferred accumulator can be up, down or updown. For an updown accumulator, the accumulated data may differ between the up and down mode:

```
...
if updown = '1' then
a <= a + b;
else
        a <= a - c;
```

Procedure:

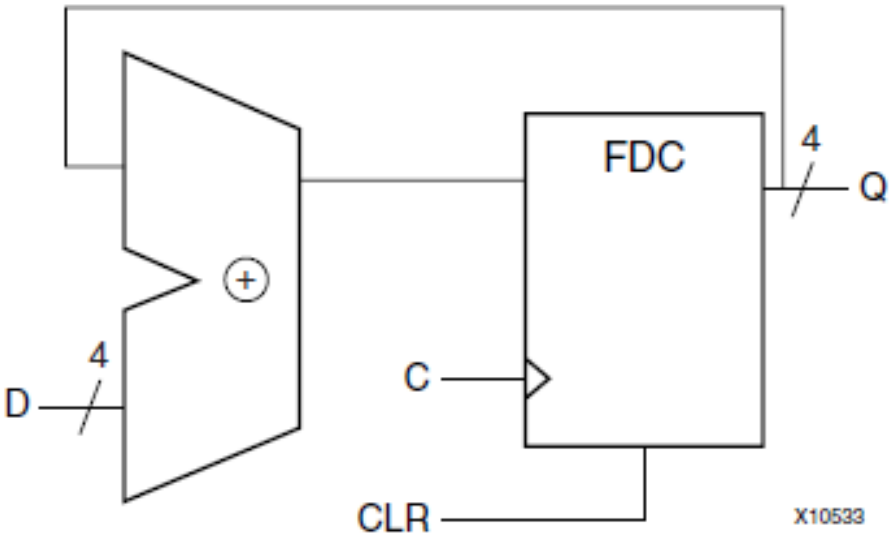
1. the accumulator circuit is designed.
2. The Verilog Module Source for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

Program:

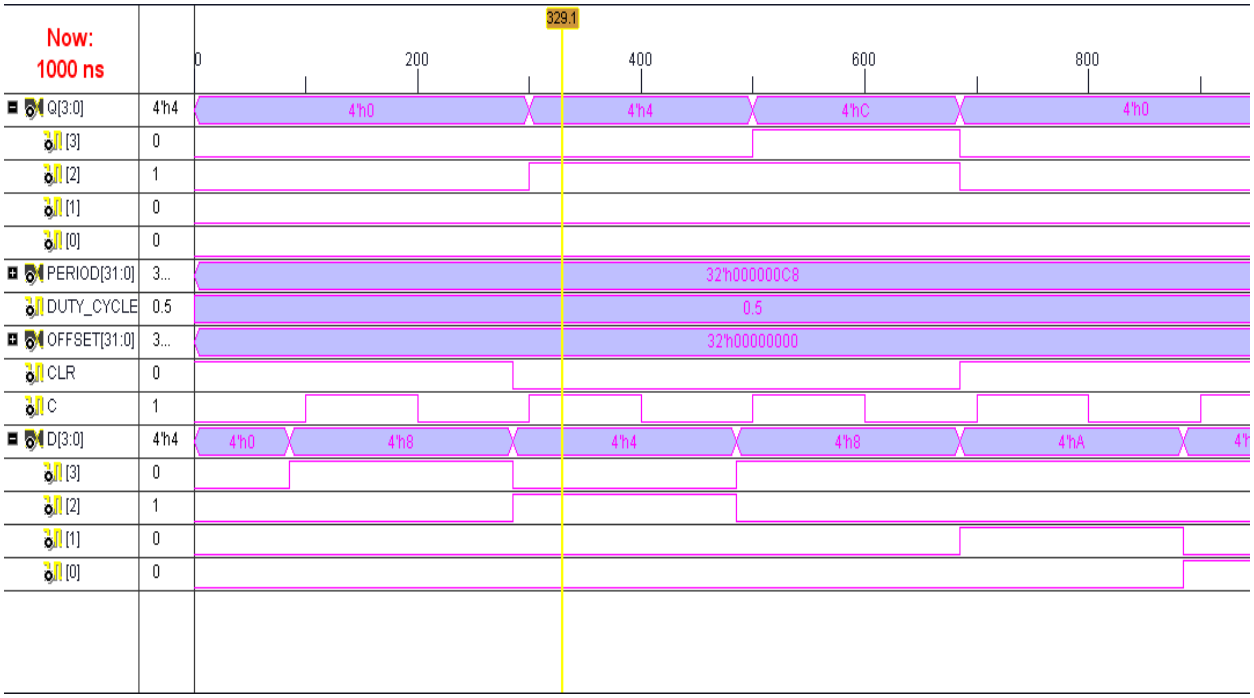
```
module accum (C, CLR, D, Q);
input C, CLR;
input  [3:0] D;
output [3:0] Q;
reg    [3:0] tmp;

  always @(posedge C or posedge CLR)
    begin
      if (CLR)
        tmp = 4'b0000;
```

```
    else
      tmp = tmp + D;
  end
assign Q = tmp;
endmodule
```

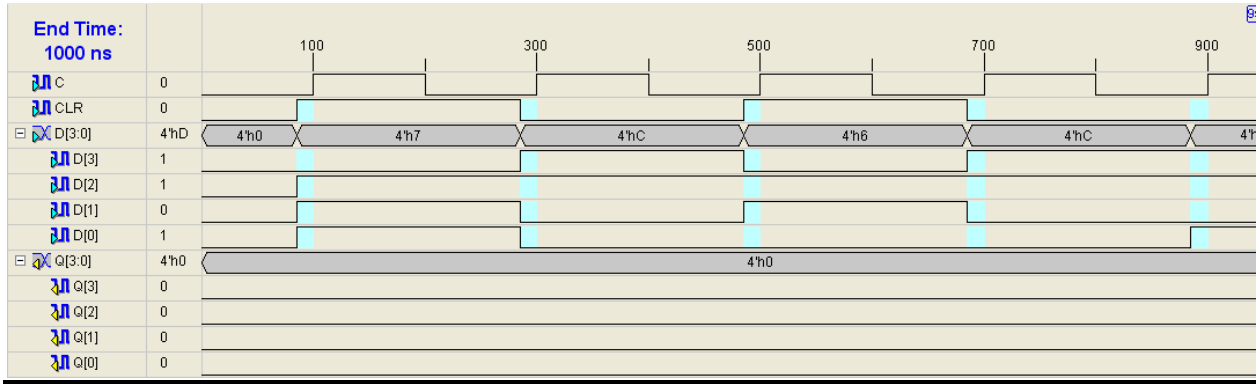
Circuit diagram:



Wave form of accumulator :



Testbenchwaveform of accumulator:



RESULT:

Thus the logic circuit for the Accumulator is designed in Verilog HDL and the output is verified.

Expt . No: 13 Date :	# Design of Decoder

Aim:

Realize the 3 to 8 Decoder in Verilog HDL.

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ModelSim Simulator.

Theory:

A decoder is a combinational circuit that converts binary information from 'n' input lines to a maximum of 2^n unique output lines. It performs the reverse operation of the encoder. If the n-bit decoded information has unused or don't-care combinations, the decoder output will have fewer than 2^n outputs. The decoders are represented as n-to-m line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables. The name decoder is also used in conjunction with some code converters such as BCD-to-seven-segment decoders. Most, if not all, IC decoders include one or more enable inputs to control the circuit operation. A decoder with an enable input can function as a de-multiplexer.

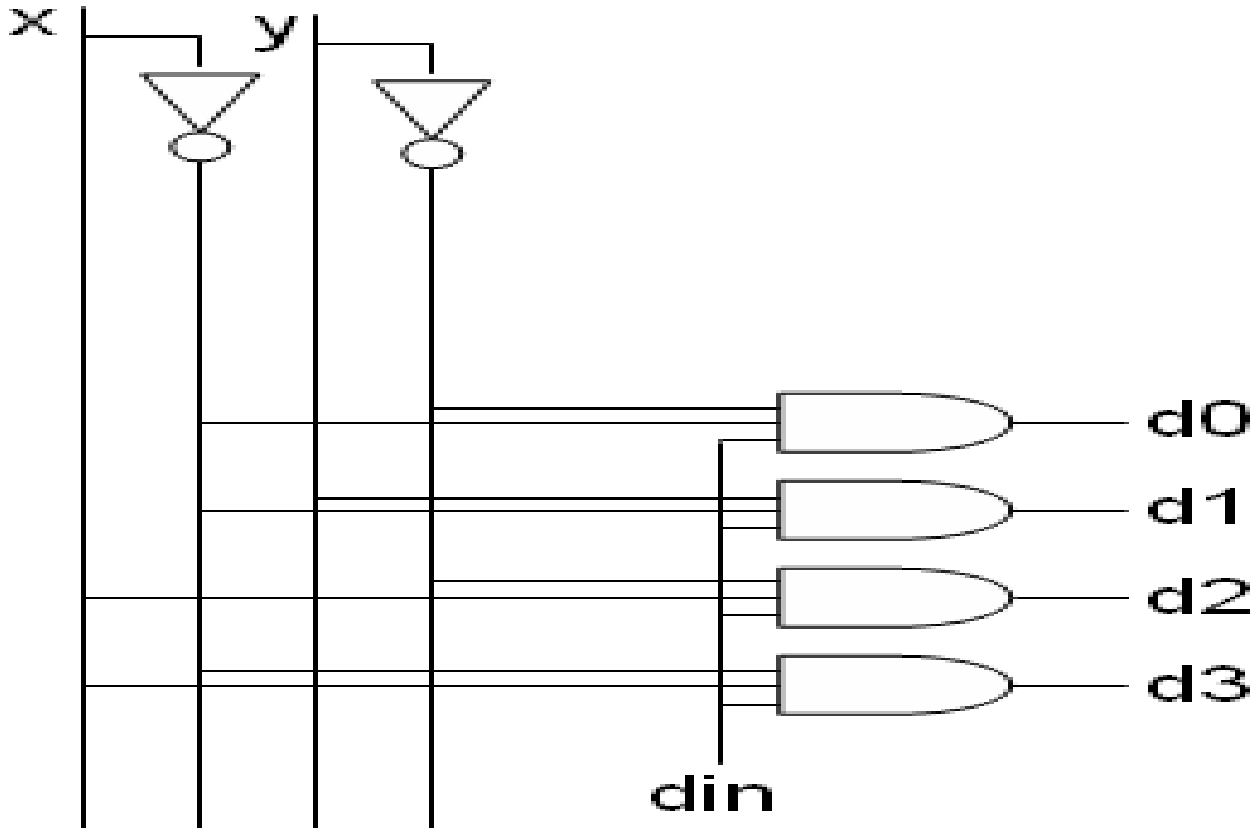
Procedure:

1. The decoder circuit is designed and the Boolean function is found out.
2. The Verilog Module Source for the circuit is written.
3. It is implemented in Model Sim and Simulated.
4. Signals are provided and Output Waveforms are viewed.

Decoder using verilog code

```
module my_decodr(d,x);
output [0:7] d;
input [0:2] x;
wire [0:2] temp;
not n1(temp[0],x[0]);
not n2(temp[1],x[1]);
not n3(temp[2],x[2]);
and a0(d[0],temp[0],temp[1],temp[2]);
and a1(d[1],temp[0],temp[1],x[2]);
and a2(d[2],temp[0],x[1],temp[2]);
and a3(d[3],temp[0],x[1],x[2]);
and a4(d[4],x[0],temp[1],temp[2]);
and a5(d[5],x[0],temp[1],x[2]);
and a6(d[6],x[0],x[1],temp[2]);
and a7(d[7],x[0],x[1],x[2]);
endmodule
```

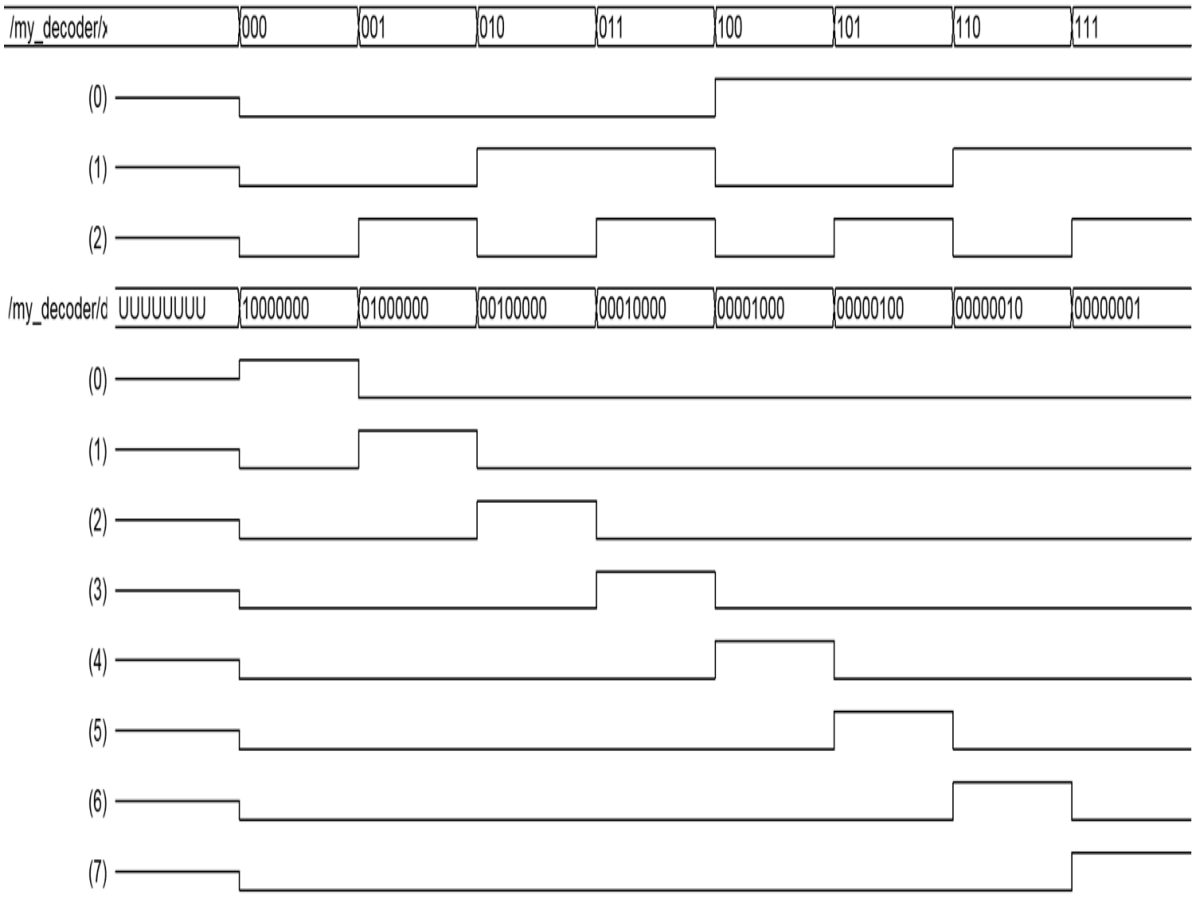
Logic diagram:



Truth Table:

INPUTS			OUTPUTS			
D_{IN}	X	Y	D_0	D_1	D_2	D_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Waveform:



RESULT:

Thus the logic circuit for the 3 to 8 decoder is designed in Verilog HDL and the output is verified.