



(Declared u/o 3 of UGC Act, 1956)

Department of Electronics and Communication Engineering

Sub Code/Name: **BET7L1 -DIGITAL CMOS VLSI LAB**

Name :

Reg No :

Branch :

Year & Semester :

LIST OF EXPERIMENTS

Sl No	Experiments	Page No
1	Design and implementation of logic gates	
2	Design and implementation of Half adder and full adder	
3	Design and implementation of Half subtractor and full subtractor	
4	Design and implementation of Boolean expressions	
5	Design and implementation of simple logic circuits	
6	Design and implementation of MUX & DEMUX – 4x1 and 8x1	
7	Encoder and decoder – 2x4 and 3x8	
8	Magnitude comparator	
9	Code converters	
10	Design and implementation of counters	
11	Design and implementation of flipflops	
12	FPGA implementation of ADC	
13	PGA implementation of traffic light controller	

INDEX

[illegible]

Introduction to VHDL

It is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. The system may be a single gate to a complete digital electronic system.

VHDL is a hardware description language used in electronic design automation to describe digital and mixed-signal system such as field-programmable gate arrays and integrated circuits. VHDL can also be used as a general purpose parallel programming language.

It can be considered a combination of following languages as:

- a) Sequential language
- b) Concurrent language
- c) Net list language
- d) Timing language
- e) Waveform Generation language

Need of VHDL

The requirement for it was generated in 1981 under VHSIC program. In this program a number of US companies were involved in designing VHSIC chips for DoD (defense department).

Most of the companies were using different hardware description to describe and develop their IC, as a result different vendors could not efficiently exchange designing with one another. Also they were provided DoD, descriptions of their chips in different hardware description language. Reuse was also an issue, thus a need for a standard language for design and documentation of the digital system was generated.

Capabilities of VHDL

1. It is used as an exchange medium between different chip vendors and CAD tool users.
2. It can be used for communication medium between different CAD and CAE tools.
3. Digital system can be modeled a set of interconnected components. Each component in turn can be modeled a set of interconnected components further.
4. It supports flexible design methodologies: Top-down Bottom-up mixed
5. It is not technology specific but it is capable of supported technology specific features.
6. It supports both synchronous and asynchronous timing modules.
7. It is an IEEE and ANSI standard.
8. It supports three basic different description styles.
9. It supports a wide range of abstraction levels ranging from abstract behavioral descriptors to vary precise gate level descriptions.
10. It has element that make large scale design modeling easier such as components, functions and procedure and package.
11. It is publically available, human readable and above all, it is not proprietary.

Package:

It provides convenient mechanism to store and share declarations that are common across many design units standard package used-IEEE std_logic_1164. It is decided by IEEE and ANSI.

Hardware abstraction:

VHDL is used to describe a model for digital hardware device. This model specifies the external view of device and one or more internal views. The internal views of the device specify the functionality or the structure while the external view specifies the interface of device through which it communicates with other modes in environment. Figure 1.1 shows hardware device and corresponding software of the device.

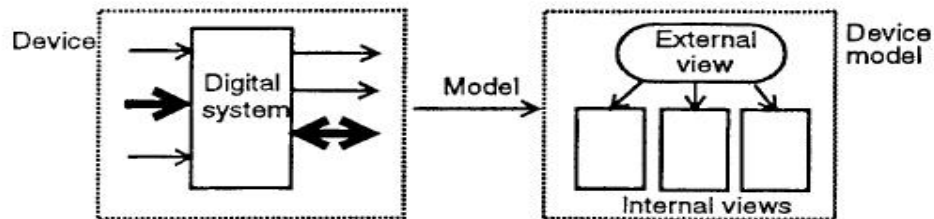


Figure 1.1 Device versus device model.

In VHDL each device model is treated as a distinct representation of unique device, called an entity. Figure 1.2, shows VHDL view of hardware device that has multiple device models, with each device model representing an entity. Each entity is described using one model, which contains one external view and one or more internal view.

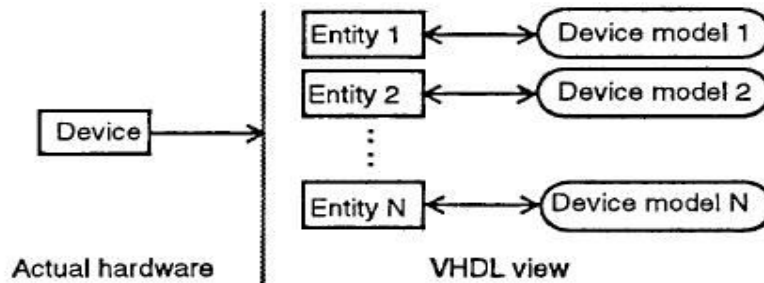


Figure 1.2 A VHDL view of a device.

VHDL provides 5 different primary constructs called the design units. They are-

1. Entity Declaration
2. Architecture Body
3. Configuration Declaration
4. Package Declaration
5. Package Body

1. Entity Declaration:

It describes the external view of

entity. Ex.-Input-output signal

names

2. Architecture Body:

It contains internal description of entity.

Ex.- A set of inter connected components that represents the structure of entity or set of concurrent or sequential statements that represent the behavior of entity.

3. Configuration Declaration:

It is used to create an entity; it specifies the binding of one architecture body from many architecture bodies that may be associated with the entity. It may also specify the binding of components used in selected architecture body to other entities. An entity may have number of different configuration.

4. Package Declaration:

A package declaration is used to store a set of common declarations like components, types, procedures, and functions. These declarations can then be imported into other design units using a context clause.

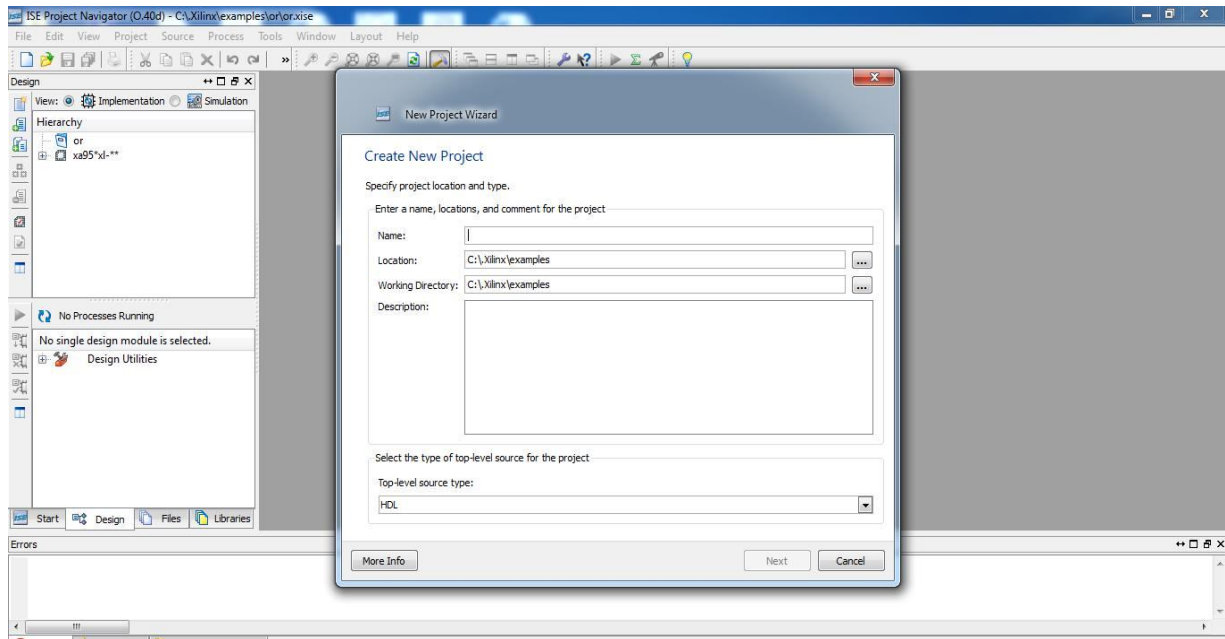
5. Package Body:

A package body is primarily used to store the definitions of functions and procedures that were declared in the corresponding package declaration, and also the complete constant declarations for any deferred constants that appear in the package declaration. Therefore, a package body is always associated with a package declaration.

Steps to implement the design

Step 1: Start the Xilinx project navigator by Stat->programs->Xilinx ISE->Project Navigator

Step 2: In the project navigator window click on new project->give file name->next.



Step 3: In the project navigator window right click on project name-> new source->VHDL module->give file name->define ports->finish.

Step 4: Write the VHDL code for any gate or

circuit. Step 5: Check Syntax and remove error if

present. Step 6: Simulate design using

Modelsim.

Step 7: In the project navigator window click on simulation->click on simulate behavioral

model. Step 8: Give inputs by right click on any input->force constant

Step 9: Run simulation

Step 10: Analyze the waveform.

To study the Simulation tools.

PROCEDURE

Create a New Project

Create a new ISE project which will target the FPGA device on the Spartan-3E Trainer Kit.

To create a new project:

2. Select **File > New Project...** The New Project Wizard appears.
3. Type **tutorial** in the Project Name field.
4. Enter or browse to a location (directory path) for the new project. A tutorial subdirectory is created automatically.
5. Verify that **HDL** is selected from the Top-Level Source Type list.
6. Click **Next** to move to the device properties page.
7. Fill in the properties in the table as shown below:

☒ Product Category: **All**

☒ Family: **Spartan3E**

☒ Device: **XC3S250**

☒ Package: **PQ208**

☒ Speed Grade: **-4**

☒ Top-Level Source Type: **HDL**

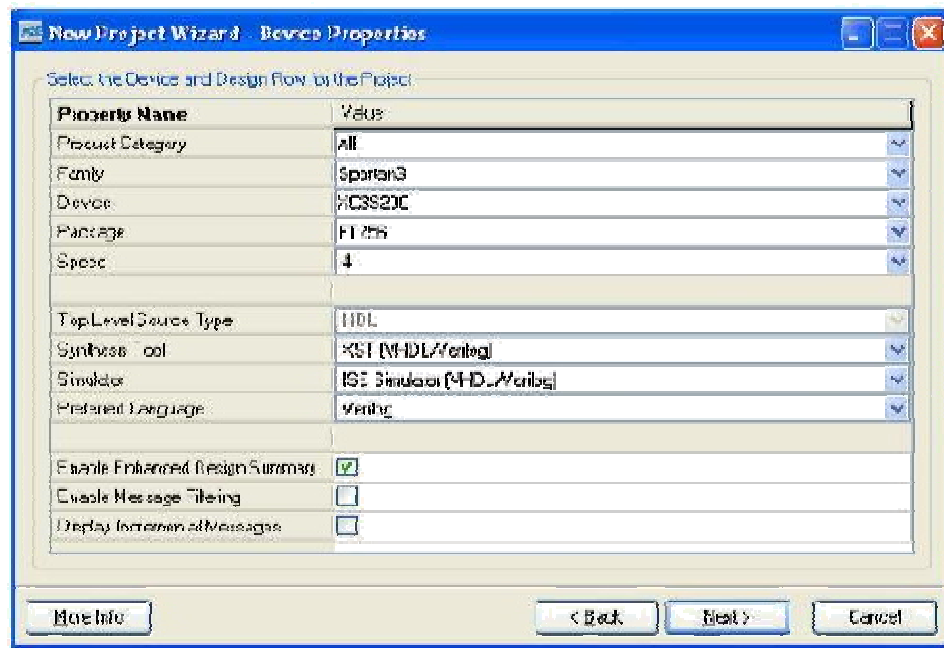
☒ Synthesis Tool: **XST (VHDL/Verilog)**

☒ Simulator: **ISE Simulator (VHDL/Verilog)**

☒ Preferred Language: **Verilog (or VHDL)**

☒ Verify that **Enable Enhanced Design Summary** is selected.

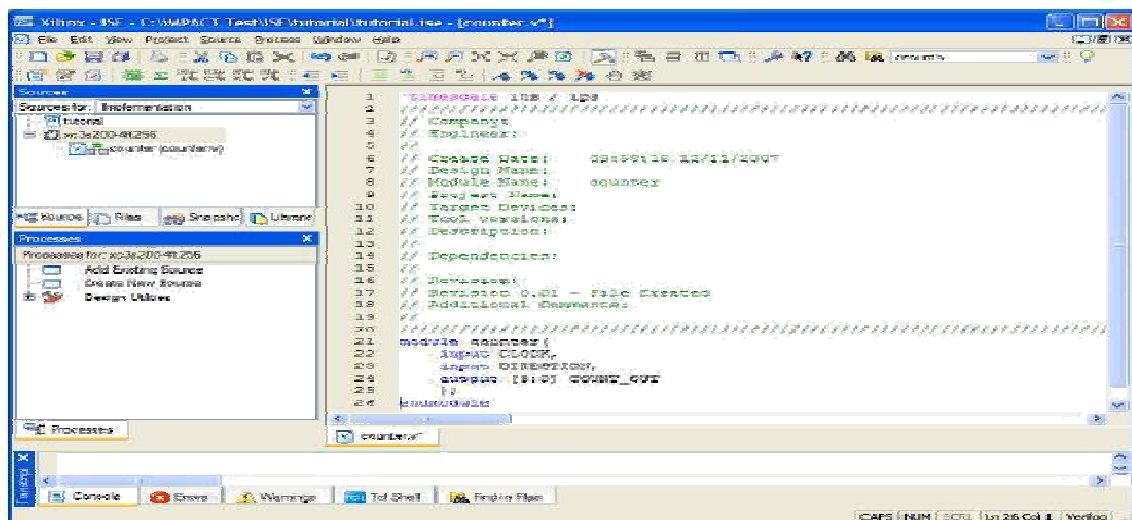
Leave the default values in the remaining fields.



6. Click **Next** to proceed to the Create New Source window in the New Project Wizard. At the end of the next section, your new project will be complete

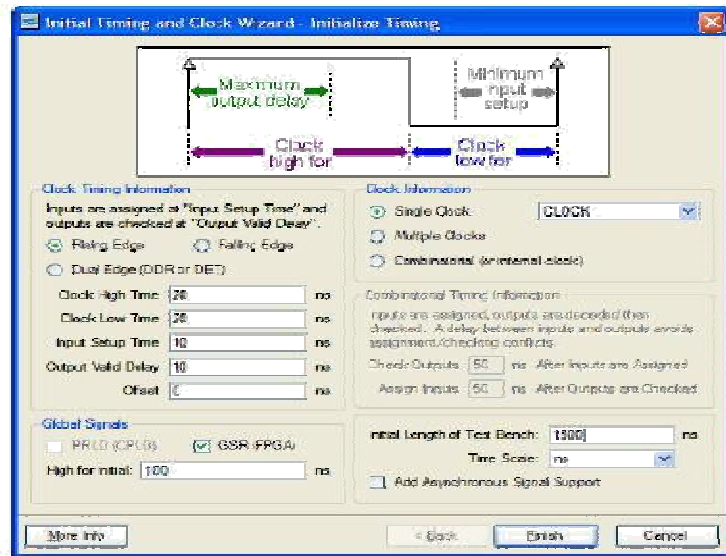
7. Click **Next**, then **Next**, then **Finish**.

The source file containing the counter module displays in the Workspace, and the counter displays in the Sources tab, as shown below.



ii. Creating a Test Bench for Simulation:

A test bench waveform contains input stimulus used to simulate the counter module. This test bench waveform is a graphical view of a test bench. It is used with a simulator to verify that the counter design meets both behavioral and timing design requirements.



1. Select the **counter** HDL file in the Sources in Project window.
2. Create a new source by selecting **Project _ New Source**.
3. In the New Source window, select **Test Bench Waveform** as the source type, and type **test bench** in the File Name field.
4. Click **Next**.
5. The Source File dialog box shows that you are associating the test bench with the source file: counter. Click **Next**.
6. Click **Finish**. You need to set initial values for your test bench waveform in the Initialize Timing dialog box before the test bench waveform editing window opens.
7. Fill in the fields in the Initialize Timing dialog box using the information below:

☐ Clock Time High: **20** ns.

Computer usage

Modelsim is used to simulate the design and to verify the functionality.

1. Modelsim 5.7
2. Xilinx ISE 7.1
3. Keil C

Ex No:1

Date:

DESIGN AND IMPLEMENTATION OF LOGIC GATES

Aim:

To design and implement logic gates.

Software tools Requirement:

- PC with Windows XP.
- Xilinx - 6.1i Software package
- Modelsim - 5.7c

Specifications:

HP Computer P4 Processor – 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -> <filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

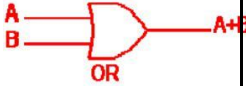
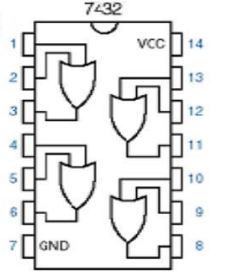
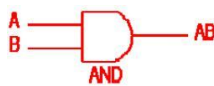
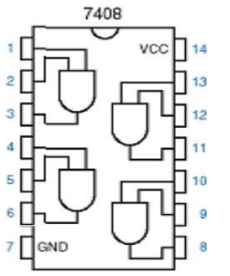
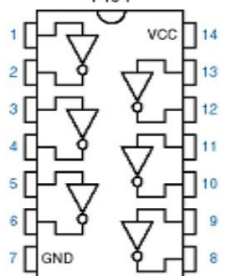
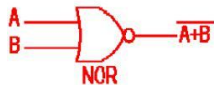
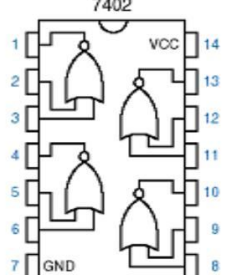
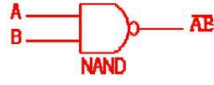
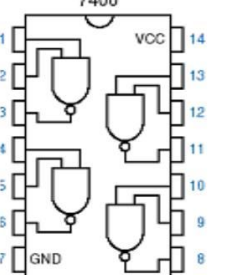
STEP 9: View -> Signals

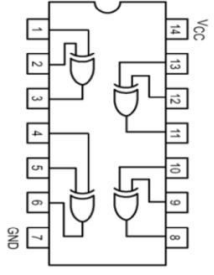
STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

Logic Gates and their Properties

Gate	Description	Truth Table			Logic Symbol	Pin Diagram
OR	The output is active high if any one of the input is in active high state, Mathematically, $Q = A+B$	A	B	Output Q		
AND	The output is active high only if both the inputs are in active high state, Mathematically, $Q = A.B$	A	B	Output Q		
NOT	In this gate the output is opposite to the input state, Mathematically, $Q = A$	A		Output Q		
NOR	The output is active high only if both the inputs are in active low state, Mathematically, $Q = (A+B)'$	A	B	Output Q		
NAND	The output is active high only if any one of the input is in active low state, Mathematically, $Q = (A.B)'$	A	B	Output Q		

XOR	The output is active high only if any one of the input is in active high state, Mathematically, $Q = A.B' + B.A'$	A	B	Output Q		
		0	0	0		
		0	1	1		
		1	0	1		
		1	1	0		

Pre lab Questions

1. What is truth table?
2. Which gates are called universal gates?
3. A basic 2-input logic circuit has a HIGH on one input and a LOW on the other input, and the output is HIGH. What type of logic circuit is it?
4. A logic circuit requires HIGH on all its inputs to make the output HIGH. What type of logic circuit is it?
5. Develop the truth table for a 3-input AND gate and also determine the total number of possible combinations for a 4-input AND gate.

VERILOG Program

a) AND Gate

Structural Model	Data Flow Model	Behavioural Model
<pre>module andstr(x,y,z); input x,y; output z; and g1(z,x,y); endmodule</pre>	<pre>module anddf(x,y,z); input x,y; output z; assign z=(x&y); endmodule</pre>	<pre>module andbeh(x,y,z); input x,y; output z; reg z; always @(x,y) z=x&y; endmodule</pre>

b) NAND Gate

Structural Model	Data Flow Model	Behavioural Model
<pre> modulenandstr(x,y,z); input x,y; output z; nand g1(z,x,y); endmodule </pre>	<pre> modulenanddf(x,y,z); input x,y; output z; assign z= !(x&y); endmodule </pre>	<pre> module nandbeh(x,y,z); input x,y; output z; reg z; always @(x,y) z=!(x&y); endmodule </pre>

c) OR Gate

Structural Model	Data Flow Model	Behavioural Model
<pre> module orstr(x,y,z); input x,y; output z; or g1(z,x,y); endmodule </pre>	<pre> module ordf(x,y,z); input x,y; output z; assign z=(x y); endmodule </pre>	<pre> module orbeh(x,y,z); input x,y; output z; reg z; always @(x,y) z=x y; endmodule </pre>

d) NOR Gate

Structural Model	Data Flow Model	Behavioural Model
<pre> modulenorstr(x,y,z); input x,y; output z; nor g1(z,x,y); endmodule </pre>	<pre> modulenordf(x,y,z); input x,y; output z; assign z= !(x y); endmodule </pre>	<pre> Modulenorbeh(x,y,z); input x,y; output z; reg z; always @(x,y) z=!(x y); endmodule </pre>

e) XOR Gate

Structural Model	Data Flow Model	Behavioural Model
<pre> module xorstr(x,y,z); input x,y; output z; xor g1(z,x,y); endmodule </pre>	<pre> module xordf(x,y,z); input x,y; output z; assign z=(x^y); endmodule </pre>	<pre> module xorbeh(x,y,z); input x,y; output z; reg z; always @(x,y) z=x^y; endmodule </pre>

f) XNOR Gate

Structural Model	Data Flow Model	Behavioural Model
<pre> module xnorstr(x,y,z); input x,y; output z; xnor g1(z,x,y); endmodule </pre>	<pre> module xnordf(x,y,z); input x,y; output z; assign z=!(x^y); endmodule </pre>	<pre> module xnorbeh(x,y,z); input x,y; output z; reg z; always @(x,y) z=!(x^y); endmodule </pre>

g) NOT Gate

Structural Model	Data Flow Model	Behavioural Model
<pre> module notstr(x,z); input x; output z; not g1(z,x); endmodule </pre>	<pre> module notdf(x,z); input x; output z; assign z= !x; endmodule </pre>	<pre> module notbeh(x,z); input x; output z; reg z; always @(x) z=!x; endmodule </pre>

Out put waveforms

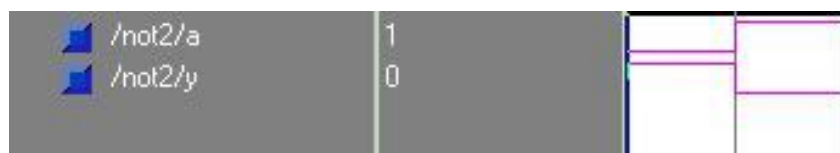
AND Gate:



OR Gate:



NOT Gate:



NOR Gate:



NAND Gate:



XOR Gate:



Result:

Logic gates were Designed and implemented.

Ex No: 2

Date:

DESIGN AND IMPLEMENTATION OF HALF ADDER AND FULL ADDER

Aim:

Design and implementation of Half adder and full adder

Software tools Requirement:

- PC with Windows XP.
- Xilinx - 6.1i Software package
- Modelsim - 5.7c

Specifications:

HP Computer P4 Processor – 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -> <filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

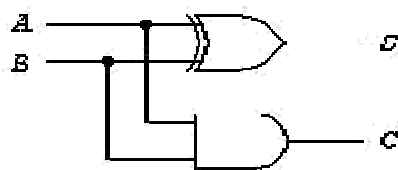
STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

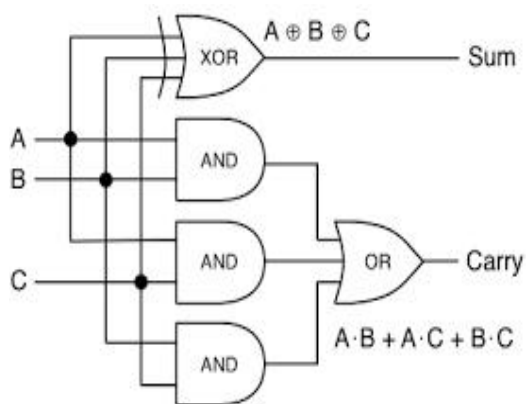
STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

Logic Diagram



Half adder



Full adder

VERILOG Program

HALF ADDER:

Structural model	Dataflow model	Behavioural model
<pre>module halfaddstr(sum, carry, a, b); output sum, carry; input a, b; xor(sum, a, b); and(carry, a, b); endmodule</pre>	<pre>module halfadddf(sum, carry, a, b); output sum, carry; input a, b; assign sum = a ^ b; assign carry = a & b; endmodule</pre>	<pre>module halfaddbeh(sum, carry, a, b); output sum, carry; input a, b; reg sum, carry; always @(a, b) sum = a ^ b; carry = a & b; endmodule</pre>

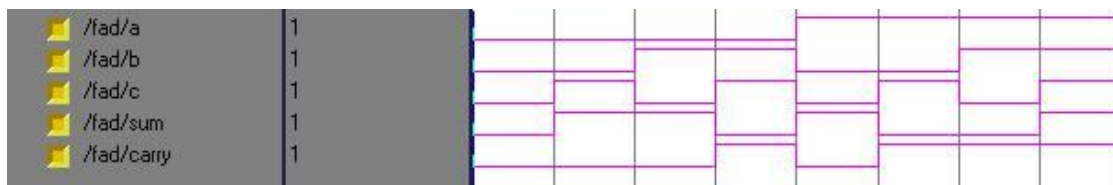
FULL ADDER:

Structural model	Dataflow model	Behavioural model
<pre> module fulladdstr(sum,carry,a,b,c); outputsum,carry; inputa,b,c; xor g1(sum,a,b,c); and g2(x,a,b); and g3(y,b,c); and g4(z,c,a); or g5(carry,x,z,y); endmodule </pre>	<pre> modulefulladddf(sum,carry,a,b,c); outputsum,carry; inputa,b,c; assign sum = a ^ b^c; assign carry=(a&b) (b&c) (c&a); endmodule </pre>	<pre> modulefulladdbeh(sum,carry,a,b,c); outputsum,carry; inputa,b,c; regsum,carry; always @ (a,b,c) sum = a ^ b^c; carry=(a&b) (b&c) (c&a); endmodule </pre>

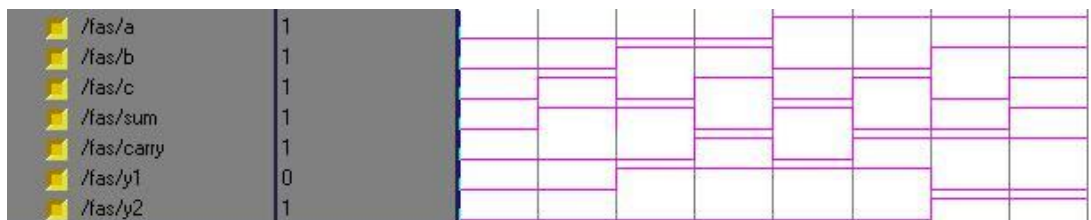
Half Adder:



Full adder Dataflow modeling:



Full adder structural modeling:



Result: Half adder and Full adder's were Designed and implemented.

Ex No: 3

Date:

DESIGN AND IMPLEMENTATION OF HALF SUBTRACTOR AND FULL SUBTRACTOR

Aim:

Design and implementation of half subtractor and full subtractor

Software tools Requirement:

- PC with Windows XP.
- Xilinx - 6.1i Software package
- Modelsim - 5.7c

Specifications:

HP Computer P4 Processor – 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -> <filename.v>

STEP 7: Compile the program

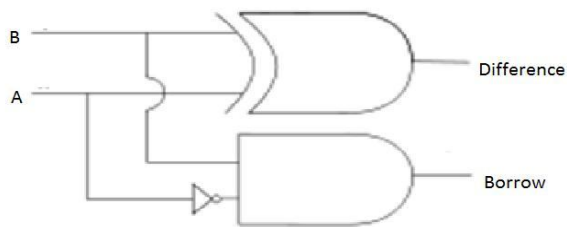
STEP 8: Simulate -> expand work -> select file -> ok

STEP 9: View -> Signals

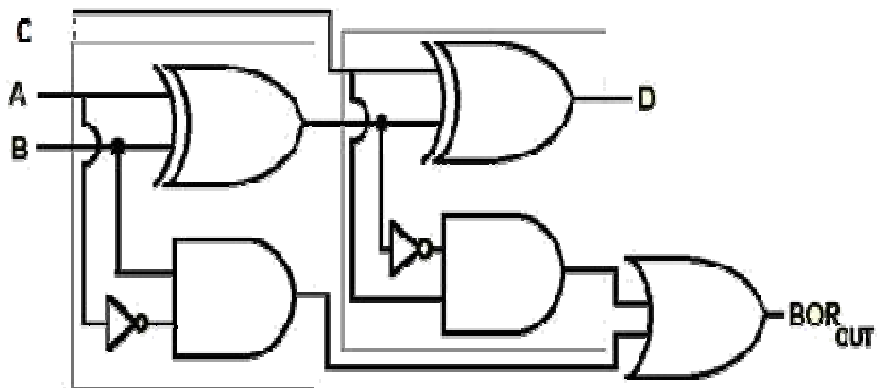
STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again



Halfsubtractor



Full subtractor

HALF SUBTRACTOR:

Structural model	Dataflow Model	Behavioural Model
<pre> module halfsubtstr(diff,borrow,a,b); output diff,borrow; input a,b; xor(diff,a,b); and(borrow,~a,b); endmodule </pre>	<pre> module halfsubtdf(diff,borrow,a,b); output diff,borrow; input a,b; assign diff = a ^ b; assign borrow = (~a & b); endmodule </pre>	<pre> module halfsubtbeh(diff,borrow,a,b); output diff,borrow; input a,b; reg diff,borrow; always @(a,b) diff = a ^ b; borrow = (~a & b); endmodule </pre>

FULL SUBTRACTOR:

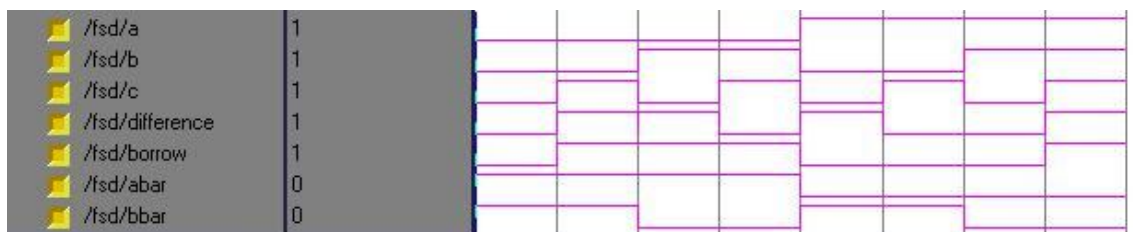
Structural model	Dataflow Model	Behavioural Model
<pre> module fullsubtstr(diff,borrow,a,b,c); outputdiff,borrow; inputa,b,c; wire a0,q,r,s,t; not(a0,a); xor(x,a,b); xor(diff,x,c); and(y,a0,b); and(z,~x,c); or(borrow,y,z); endmodule </pre>	<pre> modulefullsubtdf(diff,borrow,a,b,c); outputdiff,borrow; inputa,b,c; assign diff = a^b^c; assign borrow=~a&b) (~(a^b)&c); endmodule </pre>	<pre> modulefullsubtbeh(diff,borrow,a,b,c); outputdiff,borrow; inputa,b,c; outputdiff,borrow; always@(a,b,) diff = a^b^c; borrow=~a&b) (~(a^b)&c); endmodule </pre>

Output waveforms:

Half subtractor:



Full Subtractor Dataflow modeling:



Result: Half subtractor and Full subtractor's were Designed and implemented

Ex No: 5

Date:

DESIGN AND IMPLEMENTATION OF SIMPLE LOGIC CIRCUITS

AIM:

Design and implementation of simple logic circuits

APPARATUS REQUIRED:

S.No	Name of the equipment/ software	Quantity
1.	PC with Windows	1
2.	Xilinx Project navigator	1

PROCEDURE:

- Start the Xilinx ISE by using Start  Program files  Xilinx ISE  project navigator
- Click File  New Project
- Enter the Project Name and select the location then click next
- Select the Device and other category and click next twice and finish.
- Click on the symbol of FPGA device and then right click  click on new source.
- Select the Verilog Module and give the file name  click next and define ports  click next and finish.
- Writing the Verilog Code in Verilog Editor.
- Run the Check syntax  Process window  synthesize  double click check syntax. If any errors found then remove the errors with proper syntax & coding.
- Click on the symbol of FPGA device and then right click  click on new source.
- Select the Test Bench Waveform and give the file name  select entity click next and finish.
- Select the desired parameters for simulating your design. In this case combinational circuit and simulation time click finish.
- Assign all input signal using just click on graph and save file.
- From the source process window. Click Behavioral simulation from drop-down menu
- Select the test bench file (.tbw) and click process button  double click the Simulation Behavioral Model
- Verify your design in wave window by seeing behavior of output signal with respect to input signal.

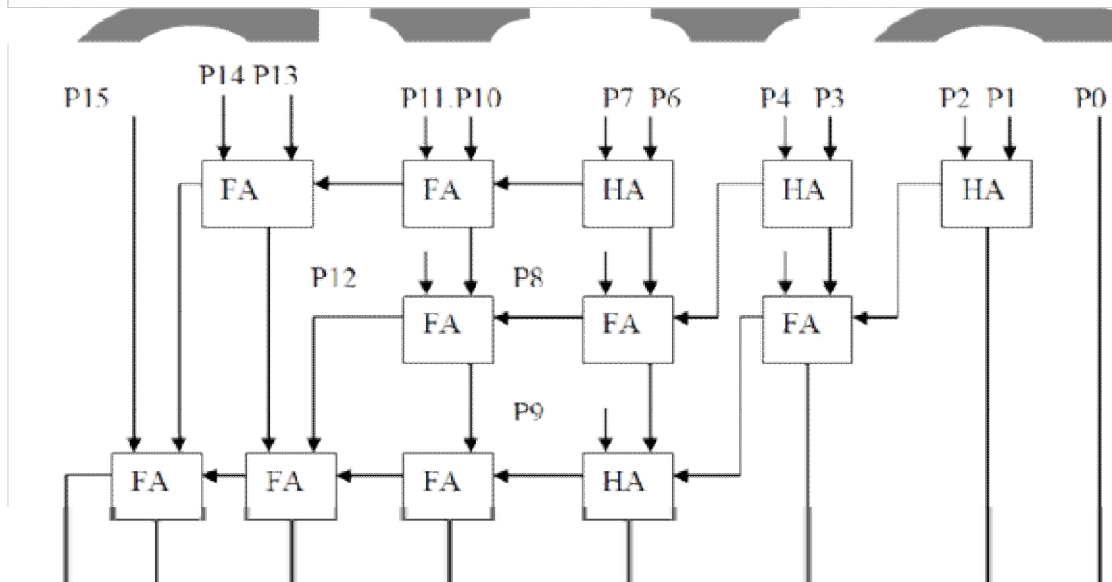
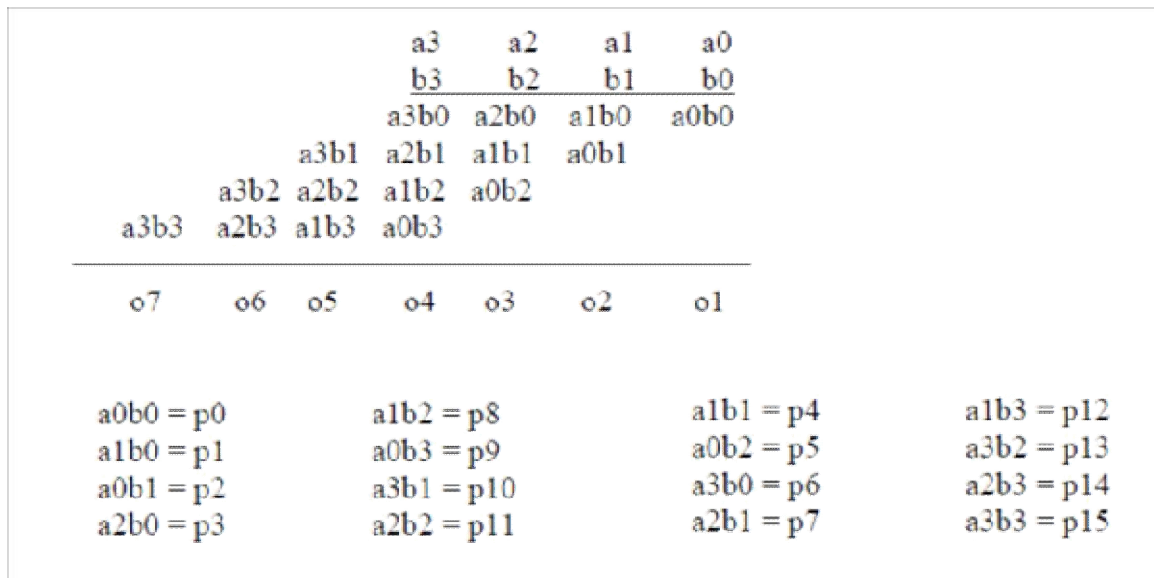
CODING:

BASIC LOGIC GATES:

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

  output
  c,d,e,f,g,h,i;
  and(c,a,b);
  or(d,a,b);
  nor(e,a,b);
  nand(f,a,b);
  xor(g,a,b);
  xnor(h,a,b);
```

4X 4 MULTIPLIER



LOGIC DIAGRAM

```
not(i,a); endmodule
```

8 BIT ADDER

```
module adder(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; xor  
(s,a,b,cin);  
assign cout = ((a & b )|(b& cin)|( a & cin)) ;  
endmodule
```

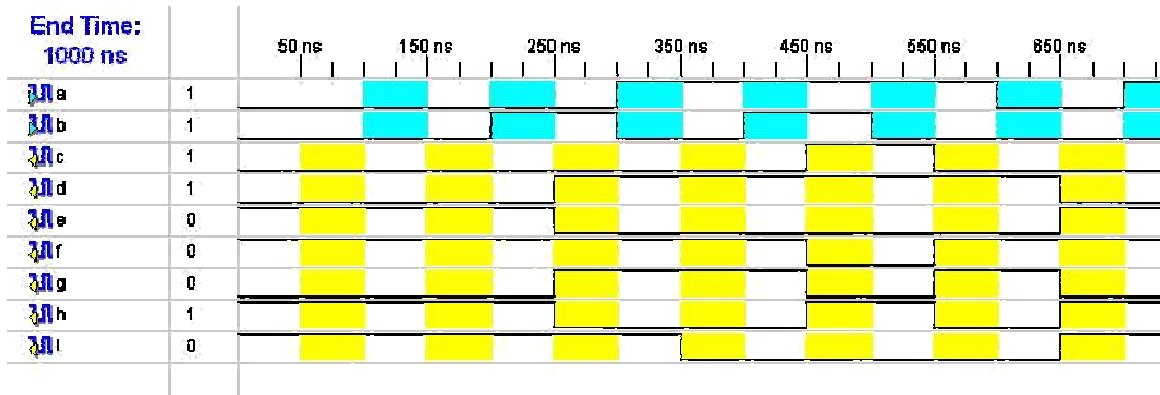
4 BIT MULTIPLIER module

```
multi(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]);
```

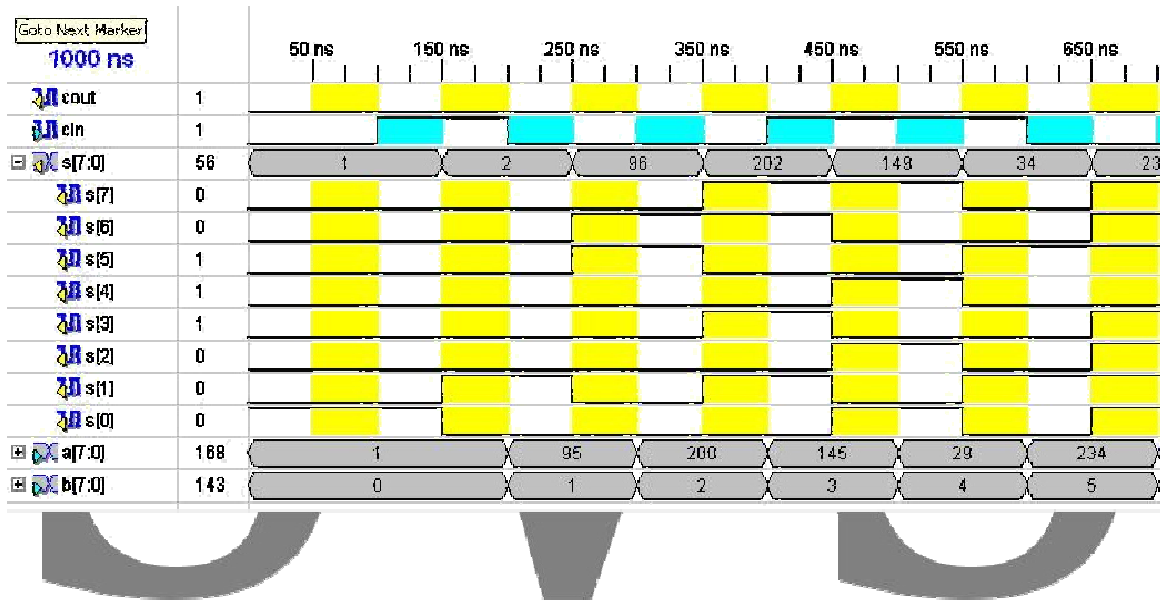
```
and(p[11],a[2],b[2]);
and(p[12],a[1],b[3]);
and(p[13],a[3],b[2]);
and(p[14],a[2],b[3]);
and(p[15],a[3],b[3]);
half ha1(s[1],c[1],p[1],p[2]);
```

OUTPUT

LOGIC GATES



8 BIT ADDER



```
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; xor
(s,x,y); and
(co,x,y);
endmodul
e
module
full(s,co,x,y,ci); input
x,y,ci;
output s,co;
wire
s1,d1,d2;
half
ha_1(s1,d1,x,y);
half
ha_2(s,d2,s1,ci); or
or_gate(co,d2,d1);
endmodule

```

RESULT:

Simple logic circuits were Designed and implemented.

Ex No : 6

Date:

DESIGN AND IMPLEMENTATION OF MULTIPLEXERS AND DEMULTIPLEXERS

Aim:

Design and implementation of MUX & DEMUX – 4x1 and 8x1

Software tools Requirement:

- PC with Windows XP.
- Xilinx - 6.1i Software package
- Modelsim - 5.7c

Specifications:

HP Computer P4 Processor – 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -> <filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

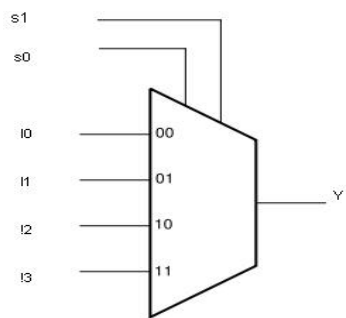
STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

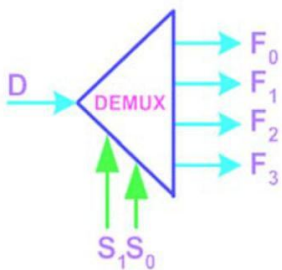
Logic Diagram



Function Table

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

4:1 Multiplexer Block diagram

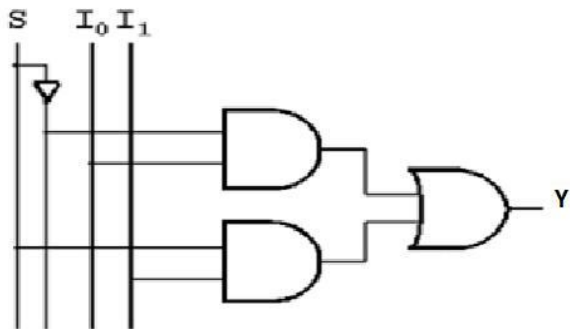


s_1	s_0	F_0	F_1	F_2	F_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

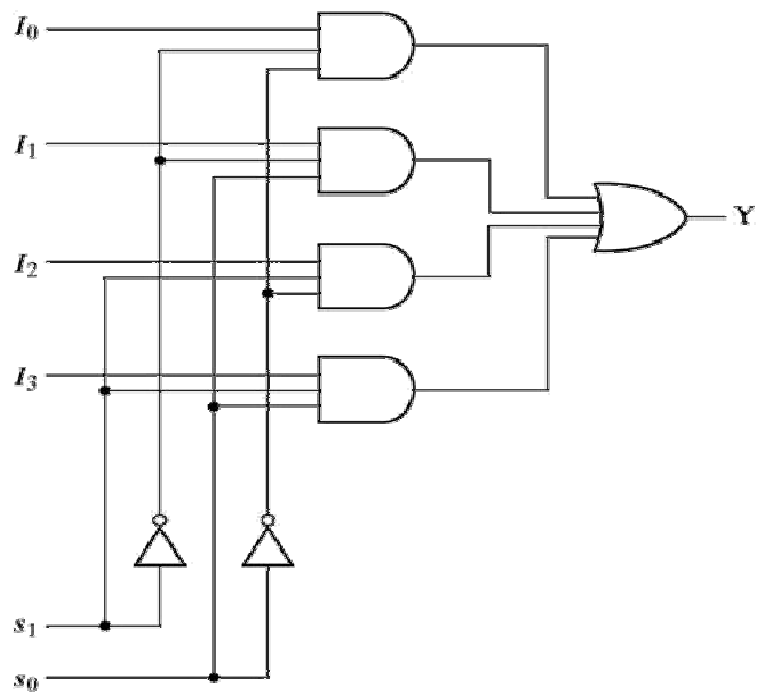
1:4 Demux Symbol

Function Table

Logic Diagram



2:1 Multiplexer



4:1 Multiplexer

Pre lab Questions

1. Define mux and demux.
2. Write their applications.
3. What is the relationship b/w input lines and select lines.
4. Design 4:1 mux and 1:4 demux.
5. Write brief notes on case statement.

VERILOG Program

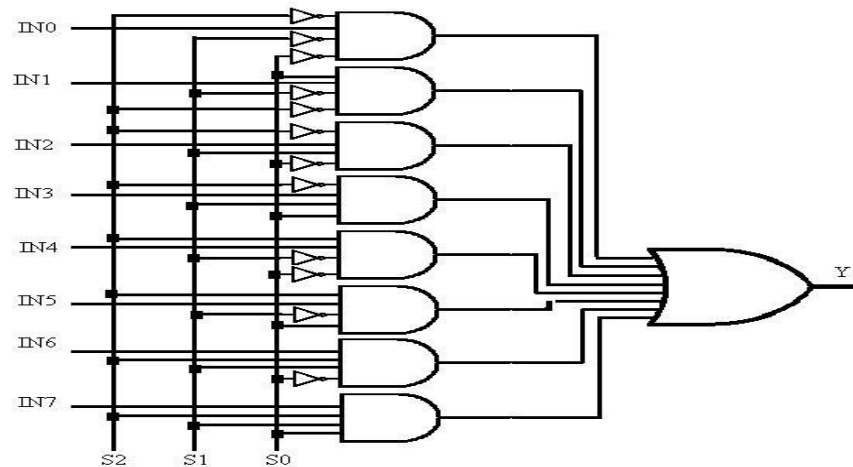
Multiplexers 2:1 MUX

Structural Model	Dataflow Model	BehaviouralModel
<pre> module mux21str(i0,i1,s,y); input i0,i1,s; output y; wire net1,net2,net3; not g1(net1,s); and g2(net2,i1,s); and g3(net3,i0,net1); or g4(y,net3,net2); endmodule </pre>	<pre> module mux21df(i0,i1,s,y); input i0,i1,s; output y; assign y =(i0&(~s)) (i1&s); endmodule </pre>	<pre> module mux21beh(i0,i1,s,y); input i0,i1,s; output y; reg y; always@(i0,i1) begin if(s==0) y=i1; if(s==1)y=i0; end endmodule </pre>

4:1 MUX

Structural Model	Dataflow Model	BehaviouralModel
<pre> module mux41str(i0,i1,i2,i3,s0,s1,y); input i0,i1,i2,i3,s0,s1; wire a,b,c,d; output y; and g1(a,i0,s0,s1); and g2(b,i1,(~s0),s1); and g3(c,i2,s0,(~s1)); and g4(d,i3,(~s0),(~s1)); or(y,a,b,c,d); </pre>	<pre> module mux41df(i0,i1,i2,i3,s0,s1,y); input i0,i1,i2,i3,s0,s1; output y; assign y=((i0&(~(s0))&(~(s1))) (i1&(~(s0))&s1) (i2&s0&(~(s1))) (i3&s0&s1); endmodule </pre>	<pre> module mux41beh(in,s,y); output y ; input [3:0] in ; input [1:0] s ; reg y; always @ (in,s) begin if (s[0]==0&s[1]==0) y = in[3]; else if (s[0]==0&s[1]==1) y = in[2]; else if (s[0]==1&s[1]==0) y = in[1]; else y = in[0]; end endmodule </pre>

Logic Diagram



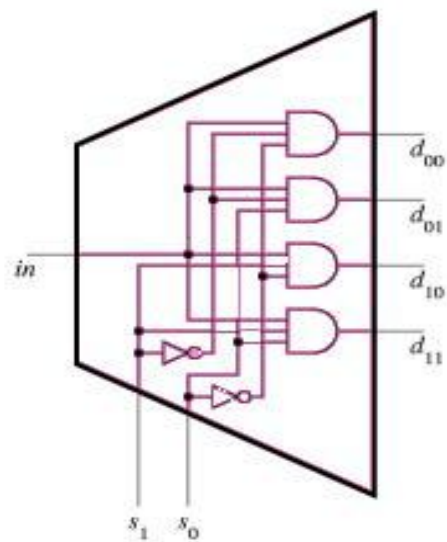
8:1 Multiplexer

VERILOG Program

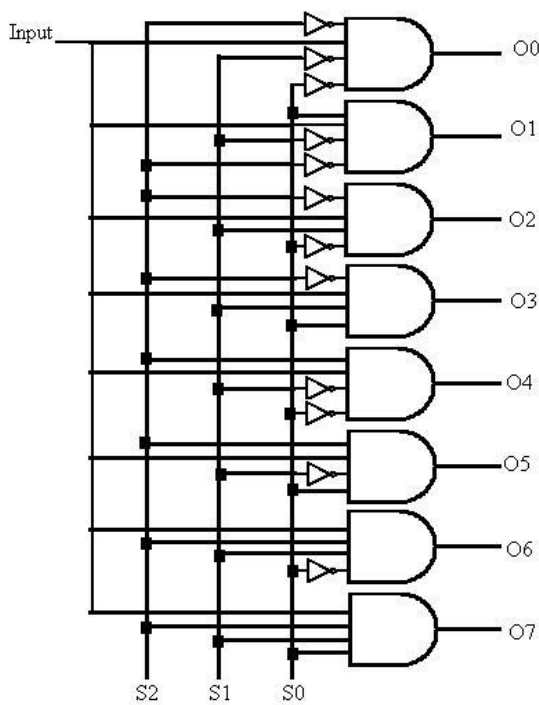
8:1 MUX

Structural Model	Dataflow Model	Behavioural Model
<pre> module mux81str(i0,i1,i2,i3,i4,i5,i6,i7,s0,s1,s2,y); input i0,i1,i2,i3,i4,i5,i6,i7,s0,s1,s2; wire a,b,c,d,e,f,g,h; output y; and g1(a,i7,s0,s1,s2); and g2(b,i6,(~s0),s1,s2); and g3(c,i5,s0,(~s1),s2); and g4(d,i4,(~s0),(~s1),s2); and g5(e,i3,s0,s1,(~s2)); and g6(f,i2,(~s0),s1,(~s2)); and g7(g,i1,s0,(~s1),(s2)); and g8(h,i0,(~s0),(~s1),(~s2)); or(y,a,b,c,d,e,f,g,h); endmodule </pre>	<pre> module mux81df(y,i,s); output y; input [7:0] i; input [2:0] s; wire se1; assign se1=(s[2]*4) (s[1]*2) s[0]; assign y=i[se1]; endmodule </pre>	<pre> module mux81beh(s,i0,i1,i2,i3,i4,i5,i6,i7,y); input [2:0] s; input i0,i1,i2,i3,i4,i5,i6,i7; reg y; always@(i0,i1,i2,i3,i4,i5,i6,i7,s) begin case(s) begin 3'd0: MUX_OUT=i0; 3'd1: MUX_OUT=i1; 3'd2: MUX_OUT=i2; 3'd3: MUX_OUT=i3; 3'd4: MUX_OUT=i4; 3'd5: MUX_OUT=i5; 3'd6: MUX_OUT=i6; 3'd7: MUX_OUT=i7; endcase end endmodule </pre>

Logic Diagram



1:4 Demultiplexer



1:8 Demultiplexer

VERILOG Program

1:4 DEMUX

Structural Model	Dataflow Model	Behavioural Model
<pre>module demux14str(in,d0,d1,d2,d3,s0,s1); output d0,d1,d2,d3; input in,s0,s1; and g1(d0,in,s0,s1); and g2(d1,in,(~s0),s1); and g3(d2,in,s0,(~s1)); and g4(d3,in,(~s0),(~s1)); endmodule</pre>	<pre>module demux14df(in,d0,d1,d2,d3,s0,s1); output d0,d1,d2,d3; input in,s0,s1; assign s0 = in & (~s0) & (~s1); assign d1= in & (~s0) & s1; assign d2= in & s0 & (~s1); assign d3= in & s0 & s1; endmodule</pre>	<pre>module demux14beh(din,sel,dout); output [3:0] dout ; reg [3:0] dout ; input din ; wire din ; input [1:0] sel ; wire [1:0] sel ; always @ (din or sel) begin case (sel) 0 : dout = {din,3'b000}; 1 : dout = {1'b0,din,2'b00}; 2 : dout = {2'b00,din,1'b0}; default : dout = {3'b000,din}; endcase end endmodule</pre>

1:8 DEMUX

Structural Model	Dataflow Model	Behavioural Model
<pre> module demux18str(in,s0,s1,s2,d0,d1,d2 ,d3,d4,d5,d6,d7); input in,s0,s1,s2; output d0,d1,d2,d3,d4,d5,d6,d7; and g1(d0,in,s0,s1,s2); and g2(d1,in,(~s0),s1,s2); and g3(d2,in,s0,(~s1),s2); and g4(d3,in,(~s0),(~s1),s2); and g5(d4,in,s0,s1,(~s2)); and g6(d5,in,(~s0),s1,(~s2)); and g7(d6,in,s0,(~s1),(~s2)); and g8(d7,in,(~s0),(~s1),(~s2)); endmodule </pre>	<pre> module demux18df(in,s0,s1,s2,i0,d1,d2,d3,d4,d5 ,d6,d7); input in,s0,s1,s2; output d0,d1,d2,d3,d4,d5,d6,d7; assign d0 = in & s0 & s1 & s2; assign d1 = in & (~s0) & s1 & s2; assign d2 = in & s0 & (~s1) & s2; assign d3 = in & (~s0) & (~s1) & s2; assign d4 = in & s0 & s1 & (~s2); assign d5 = in & (~s0) & s1 & (~s2); assign d6 = in & s0 & (~s1) & (~s2); assign d7 = in & (~s0) & (~s1) & (~s2); endmodule </pre>	<pre> module demux18beh(i, sel, y); input i; input [2:0] sel; output [7 :0] y ; reg [7:0] y; always@(i,sel) begin y=8'd0; case(sel) 3'd0:y[0]=i; 3'd1:y[1]=i; 3'd2:y[2]=i; 3'd3:y[3]=i; 3'd4:y[4]=i; 3'd5:y[5]=i; 3'd6:y[6]=i; default:y[7]=i; endcase end endmodule </pre>

Output Wave forms:

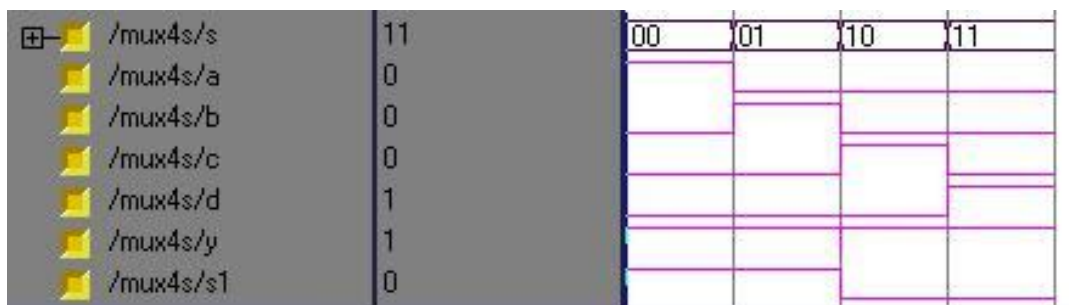
2 X 1 MUX:



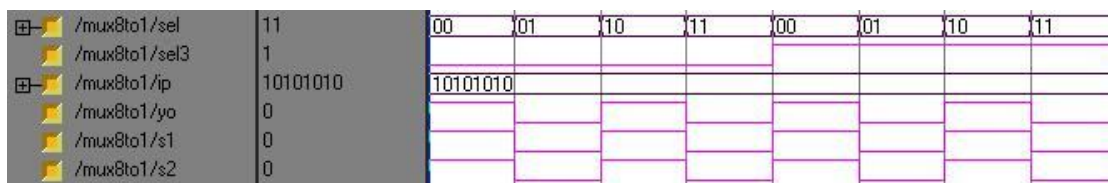
4 X 1 MUX DATAFLOW MODELING



4 X 1 MUX STRUCTURAL MODELING



MUX 8:1 Using 4:1 & 2:1



1 TO 4 DEMUX BEHAVIOURAL MODELING



Post Lab questions

1. Implement the function $f(A,B,C)=\sum m(0,1,3,5,7)$ by using Mux.
2. Write the VERILOG code for the above design
3. Write the VERILOG code for full subtractor using Demux.

Result:

Multiplexer and DeMultiplexer's were designed and implemented.

Ex No:7

Date:

DESIGN AND IMPLEMENTATION OF ENCODERS AND DECODERS

Aim:

Design and implementation of encoders and decoders

Software tools Requirement:

- PC with Windows XP.
- Xilinx - 6.1i Software package
- Modelsim - 5.7c

Specifications:

HP Computer P4 Processor – 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -> <filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

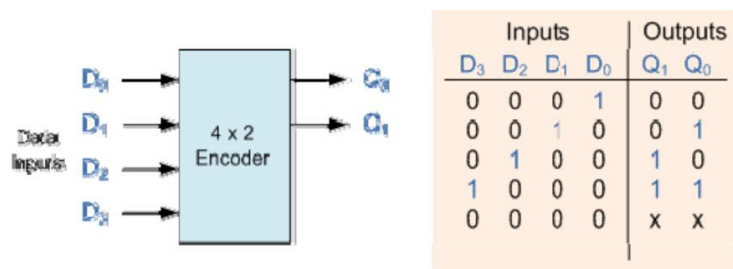
STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

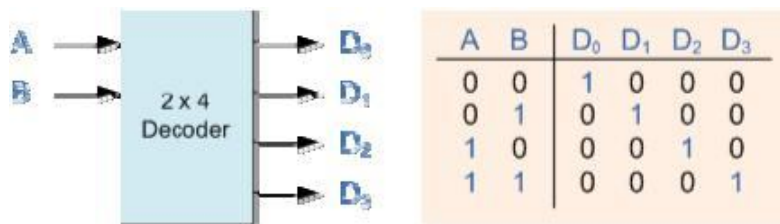
STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

Logic Diagram



4-to-2 bit Encoder

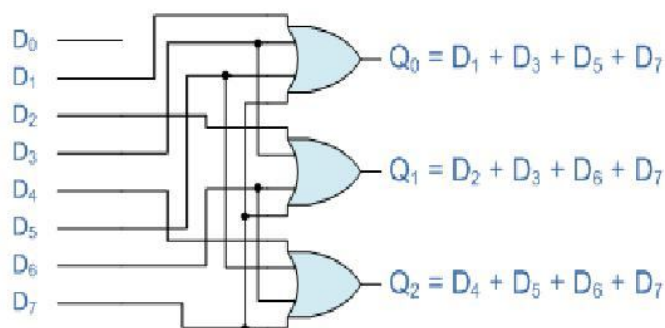


2-to-4 Binary Decoders

Pre lab Questions

1. What is difference b/w encoder and data selector?
2. What is the difference b/w decoder and data distributor?
3. Give the applications of encoder and decoder.
4. Write short notes on “with – select” statement.
5. What are the different logic state systems in Verilog?

Logic Diagram



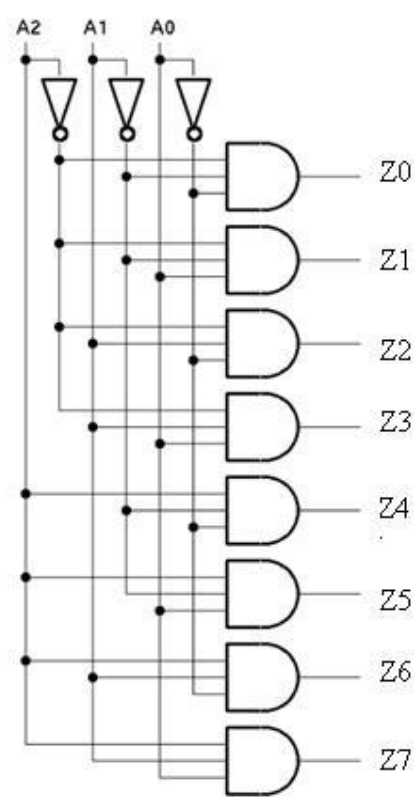
8:3 Encoder

VERILOG Program

8:3 Encoder

Structural Model	Data Flow Model	Behavioural Model
<pre>Module enc83str(d0,d1,d2,d3,d4,d5,d 6,d7,q0,q1,q2); Input d0,d1,d2,d3,d4,d5,d6,d7; Output q0,q1,q2; Or g1(q0,d1,d3,d5,d7); Or g2(q1,d2,d3,d6,d7); Or g3(q2,d4,d5,d6,d7); Endmodule</pre>	<pre>Module enc83df(d0,d1,d2,d3,d4,d5,d6, d7,q0,q1,q2); Input d0,d1,d2,d3,d4,d5,d6,d7; Output q0,q1,q2; Assign q0=d1 d3 d5 d7; Assign q1=d2 d3 d6 d7; Assign q2=d4 d5 d6 d7; Endmodule</pre>	<pre>module enc83beh (din,a,b,c); input [0:7]din; output a,b,c; reg a,b,c; always@(din) case(din) 8'b10000000:begin a=1'b0;b=1'b0,c=1'b0;end 8'b01000000:begin a=1'b0;b=1'b0;c=1'b1;end 8'b00100000:begin a=1'b0;b=1'b1;c=1'b0;end 8'b00010000:begin a=1'b0;b=1'b1;c=1'b1;end 8'b10001000:begin a=1'b1;b=1'b0,c=1'b0;end 8'b10000100:begin a=1'b1;b=1'b0,c=1'b1;end 8'b10000010:begin a=1'b1;b=1'b1,c=1'b0;end 8'b10000001:begin a=1'b1;b=1'b1,c=1'b1;end endcase endmodule</pre>

Logic Diagram



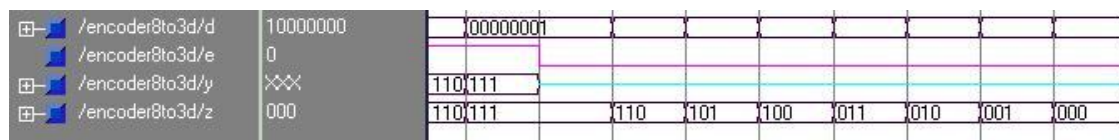
3:8 Decoder

VERILOG Program

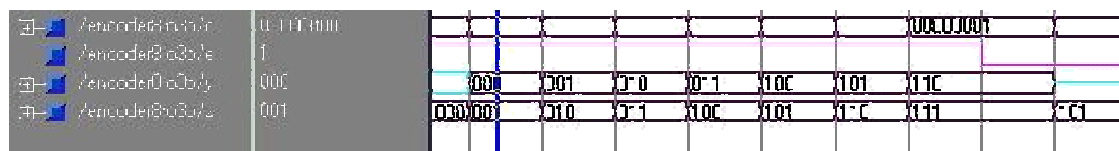
3:8 Decoder

Structural Model	Data Flow Model	Behavioural Model
<pre> module decoder38str(z0,z1,z2,z3,z4,z5,z 6,z7,a0,a1,a2); output z0,z1,z2,z3,z4,z5,z6,z7; input a0,a1,a2; not (s0,a0); not (s1,a1); not (s2,a2); and (z0,s0,s1,s2); and (z1,a0,s1,s2); and (z2,s0,a1,s2); and z3,a0,a1,s2); and (z4,s0,s1,a2); and (z5,a0,s1,a2); and (z6,s0,a1,a2); and (z7,a0,a1,a2); endmodule </pre>	<pre> module decoder38df(z,a0,a1,a2); output [7:0] z; input a0,a1,a2; assign z[0] = ~a0 & ~a1 & ~a2; assign z[1] = ~a0& ~a1& a2; assign z[2] = ~a0& a1& ~a2; assign z[3] = ~a0& a1& a2; assign z[4] = a0& ~a1& ~a2; assign z[5] = a0& ~a1& a2; assign z[6] = a0& a1& ~a2; assign z[7] = a0& a1& a2; endmodule </pre>	<pre> module decoder38beh(s el,out1); input [2:0] sel; outputreg [7:0] out1; always @(sel,out1) case (sel) 3'b000 : out1 = 8'b00000001; 3'b001 : out1 = 8'b00000010; 3'b010 : out1 = 8'b00000100; 3'b011 : out1 = 8'b00001000; 3'b100 : out1 = 8'b00010000; 3'b101 : out1 = 8'b00100000; 3'b110 : out1 = 8'b01000000; default : out1 = 8'b10000000; endcase endmodule </pre>

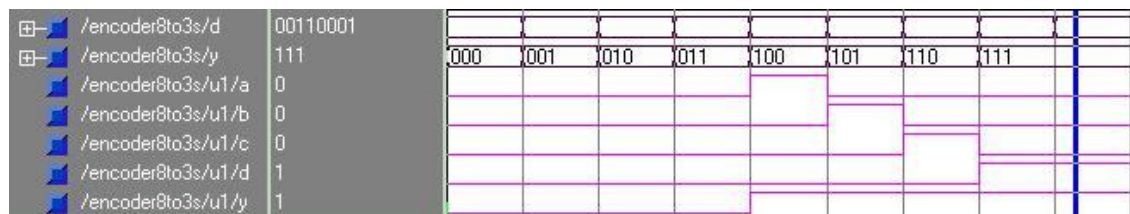
Output Wave forms:



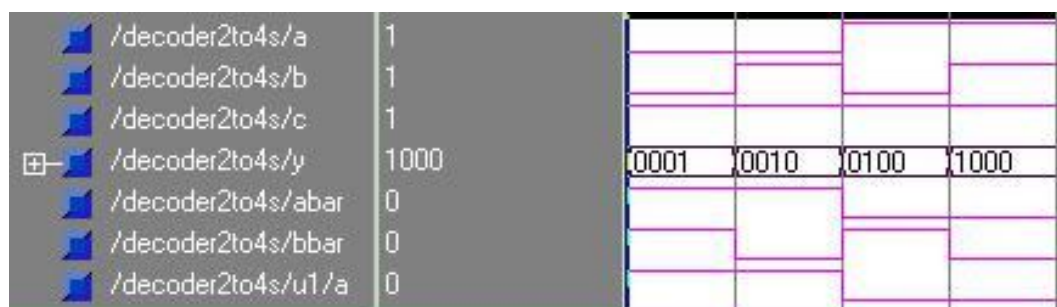
ENCODER 8:3 Data Flow Modeling



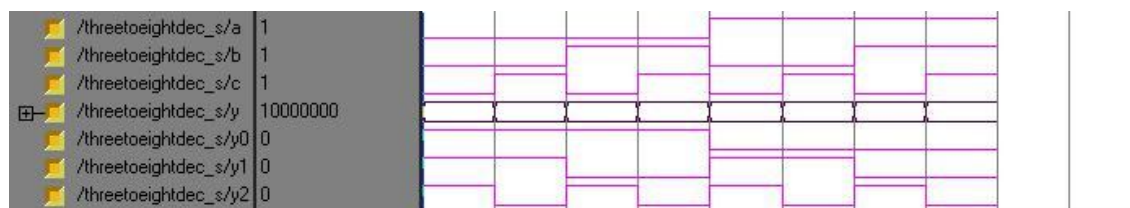
ENCODER 8:3 Behavioral Modeling



ENCODER 8:3 Structural Modeling



DECODER 2:4 Structural Modeling



DECODER 3:8 Structural Modeling

Result:

Encoder and Decoder's were designed and implemented

Ex No:8

Date:

MAGNITUDE COMPARATOR

Aim: To design and implement Magnitude comparator.

Apparatus Required:

- PC with Windows XP.
- Xilinx - 6.1i Software package
- Modelsim - 5.7c

Specifications:

HP Computer P4 Processor – 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -> <filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

Program:

VHDL CODE :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity comparator is
port (a_in, b_in: in std_logic_vector (3 downto 0) ;
      g_op, e_op, L_op: out std_logic);
```

end comparator;

architecture behavioral of comparator is begin

```
    process (a_in, b_in) begin
        if (a_in<b_in) then
            L_op<= '1';
        else L_op<=
            '0'; end if;
        if (a_in>b_in) then
            g_op<= '1';
        else g_op<=
            '0'; end if;
        if (a_in = b_in) then
            e_op<= '1';
        else e_op<=
            '0'; end if;
    end process;
end;
```

VERILOG CODE :

```
module comparater(a_in, b_in, L_op,g_op,e_op);
    input [3:0] a_in;
    input [3:0] b_in;
    output L_op;
    output g_op;
    output e_op;
    reg L_op,g_op,e_op;
    always @ (a_in,b_in) begin
        if (a_in<b_in)
            L_op=1'b1;
        else
            L_op=1'b0;
            if (a_in>b_in)
                g_op=1'b1;
        else
            g_op=1'b0;
            if (a_in==b_in)
                e_op=1'b1;
        else
            e_op=1'b0;
    end
```

```
end  
endmodule
```

Expression:

```
sum = a_in  $\oplus$  b_in  $\oplus$  c_in;  
carry= (a_in.b_in)+(b_in.c_in)+(a_in.b_in);
```

Result:

Thus the design and implementation of Magnitude comparator done.

Ex No:10

Date: **DESIGN AND IMPLEMENTATION OF COUNTERS**

Aim:

Design and implementation of Counters.

Software tools Requirement:

- PC with Windows XP.
- Xilinx - 6.1i Software package
- Modelsim - 5.7c

Specifications:

HP Computer P4 Processor – 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -> <filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

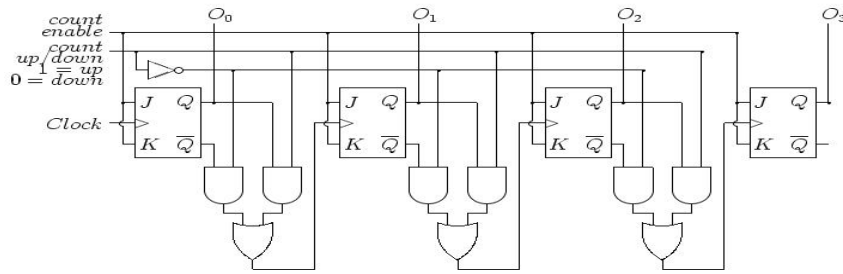
STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

STEP 12: Change input values and run again

Logic Diagram



Updown Counter

Pre Lab questions

1. How does synchronous counter differ from asynchronous counter?
2. How many flip-flops do you require to design Mod-6 counter.
3. What are the different types of counters?
4. What are the different types of shift registers?
5. How many f/fs are needed for n-bit counter?
6. What is meant by universal shift register?

VERILOG Program

Up Down Counter

```
module updown(out, clk, reset, updown);
    output [3:0] out;

    input clk, reset, updown;

    reg [3:0] out;

    always @(posedge clk)
        if(reset) begin
            out <= 4'b0;
        end else if(updown)
            begin out <= out + 1;
```

```
end else

begin

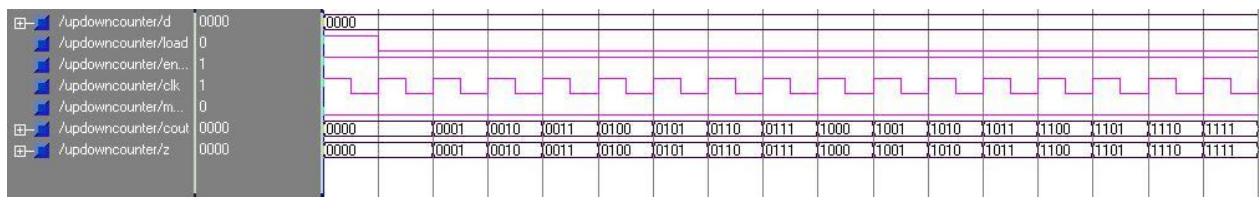
out<=out-1;

end

endmodule
```

Output Waveform:

UP COUNTER



DOWN COUNTER



Result:

Counters were designed and implemented.

Ex No:11

Date:

DESIGN AND IMPLEMENTATION OF FLIP FLOPS

Aim:

Design and implementation of flip flops

Software tools Requirement:

- PC with Windows XP.
- Xilinx - 6.1i Software package
- Modelsim - 5.7c

Specifications:

HP Computer P4 Processor – 2.8 GHz, 2GB RAM, 160 GB Hard Disk

Algorithm

STEP 1: Open ModelSim XE II / Starter 5.7C

STEP 2: File -> Change directory -> D:\<register number>

STEP 3: File -> New Library -> ok

STEP 4: File -> New Source -> Verilog

STEP 5: Type the program

STEP 6: File -> Save -> <filename.v>

STEP 7: Compile the program

STEP 8: Simulate -> expand work -> select file -> ok

STEP 9: View -> Signals

STEP 10: Select values -> Edit -> Force -> input values

STEP 11: Add -> Wave -> Selected signals -> Run

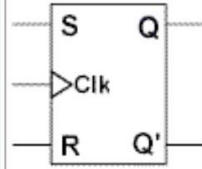
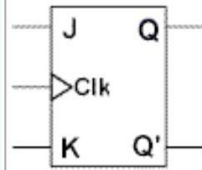
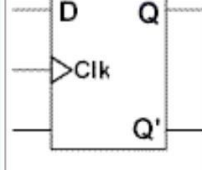
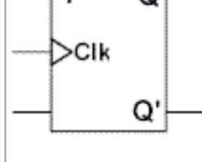
STEP 12: Change input values and run again

Flip-Flops Logic diagram and their properties

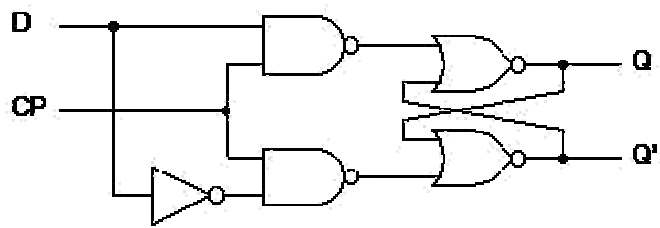
Flip-flops are synchronous bitable devices. The term synchronous means the output changes state only when the clock input is triggered. That is, changes in the output occur in synchronization with the clock.

A flip-flop circuit has two outputs, one for the normal value and one for the complement value of the stored bit. Since memory elements in sequential circuits are usually flip-flops, it is worth summarizing the behavior of various flip-flop types before proceeding further.

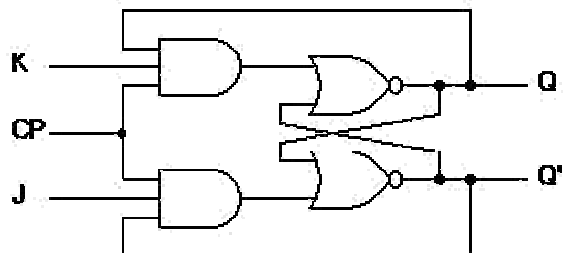
All flip-flops can be divided into four basic types: SR, JK, D and T. They differ in the number of inputs and in the response invoked by different value of input signals. The four types of flip-flops are defined in the Table 5.1. Each of these flip-flops can be uniquely described by its graphical symbol, its characteristic table, its characteristic equation or excitation table. All flip-flops have output signals Q and Q'.

Flip-Flop Name	Flip-Flop Symbol	Characteristic Table	Characteristic Equation	Excitation Table																																			
SR		<table><tr><th>S</th><th>R</th><th>Q(next)</th></tr><tr><td>0</td><td>0</td><td>Q</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>?</td></tr></table>	S	R	Q(next)	0	0	Q	0	1	0	1	0	1	1	1	?	$Q(\text{next}) = S + R'Q$ $SR = 0$	<table><tr><th>Q</th><th>Q(next)</th><th>S</th><th>R</th></tr><tr><td>0</td><td>0</td><td>0</td><td>X</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>X</td><td>0</td></tr></table>	Q	Q(next)	S	R	0	0	0	X	0	1	1	0	1	0	0	1	1	1	X	0
S	R	Q(next)																																					
0	0	Q																																					
0	1	0																																					
1	0	1																																					
1	1	?																																					
Q	Q(next)	S	R																																				
0	0	0	X																																				
0	1	1	0																																				
1	0	0	1																																				
1	1	X	0																																				
JK		<table><tr><th>J</th><th>K</th><th>Q(next)</th></tr><tr><td>0</td><td>0</td><td>Q</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>Q'</td></tr></table>	J	K	Q(next)	0	0	Q	0	1	0	1	0	1	1	1	Q'	$Q(\text{next}) = JQ' + K'Q$	<table><tr><th>Q</th><th>Q(next)</th><th>J</th><th>K</th></tr><tr><td>0</td><td>0</td><td>0</td><td>X</td></tr><tr><td>0</td><td>1</td><td>1</td><td>X</td></tr><tr><td>1</td><td>0</td><td>X</td><td>1</td></tr><tr><td>1</td><td>1</td><td>X</td><td>0</td></tr></table>	Q	Q(next)	J	K	0	0	0	X	0	1	1	X	1	0	X	1	1	1	X	0
J	K	Q(next)																																					
0	0	Q																																					
0	1	0																																					
1	0	1																																					
1	1	Q'																																					
Q	Q(next)	J	K																																				
0	0	0	X																																				
0	1	1	X																																				
1	0	X	1																																				
1	1	X	0																																				
D		<table><tr><th>D</th><th>Q(next)</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	D	Q(next)	0	0	1	1	$Q(\text{next}) = D$	<table><tr><th>Q</th><th>Q(next)</th><th>D</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	Q	Q(next)	D	0	0	0	0	1	1	1	0	0	1	1	1														
D	Q(next)																																						
0	0																																						
1	1																																						
Q	Q(next)	D																																					
0	0	0																																					
0	1	1																																					
1	0	0																																					
1	1	1																																					
T		<table><tr><th>T</th><th>Q(next)</th></tr><tr><td>0</td><td>Q</td></tr><tr><td>1</td><td>Q'</td></tr></table>	T	Q(next)	0	Q	1	Q'	$Q(\text{next}) = TQ' + T'Q$	<table><tr><th>Q</th><th>Q(next)</th><th>T</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	Q	Q(next)	T	0	0	0	0	1	1	1	0	1	1	1	0														
T	Q(next)																																						
0	Q																																						
1	Q'																																						
Q	Q(next)	T																																					
0	0	0																																					
0	1	1																																					
1	0	1																																					
1	1	0																																					

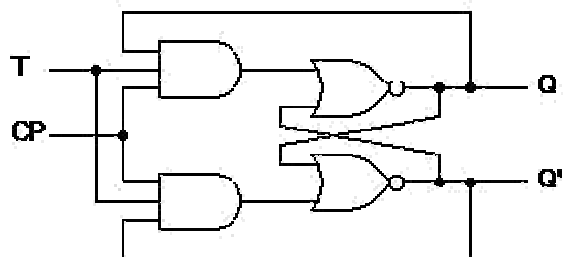
Flip-flops and their properties



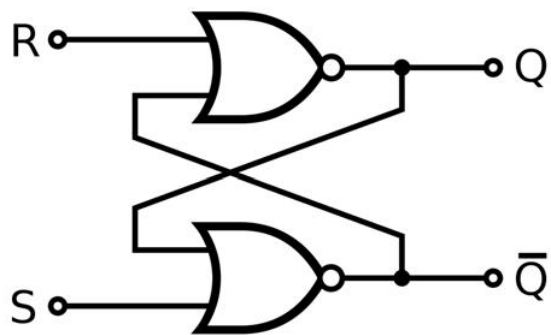
D- Flip Flop



JK Flip Flop



T Flip Flop



T Flip Flop

Pre-lab Questions

1. Describe the main difference between a gated S-R latch and an edge-triggered S-R flip-flop.
2. How does a JK flip-flop differ from an SR flip-flop in its basic operation?
3. Describe the basic difference between pulse-triggered and edge-triggered flip-flops.
4. What is use of characteristic and excitation table?
5. What are synchronous and asynchronous circuits?
6. How many flip flops due you require storing the data 1101?

S-R Flip Flop

Dataflow Modelling

```
module sr_df (s, r, q, q_n);  
    input s, r;  
    output q, q_n;  
    assign q_n = ~(s | q);  
    assign q = ~(r | q_n);  
endmodule
```

Structural Modelling

```
module sr_st(s,r,q,q_n);  
    input s, r;  
    output q, q_n;  
    or g1(q_n,~s,~q);  
    or g2(q,~r,~q_n);  
endmodule
```

Behavioral

```
module sr_beh(s,r,q,q_n);  
    input s, r;  
    output q, q_n;  
    reg q, q_n;  
    always@(s,r)  
    begin  
        q,n = ~(s|q);  
        assign q = ~(r | q_n);  
    endmodule
```

T Flip Flop

Behavioral Modelling

```
module t_beh(q,q1,t,c);  
    output q,q1;  
    input t,c;  
    reg q,q1;  
    initial  
    begin  
        q=1'b1;  
        q1=1'b0;  
    end  
    always @ (c)  
    begin  
        if(c) begin  
            if (t==1'b0) begin q=q; q1=q1; end else  
                begin q=~q; q1=~q1; end  
        end  
    end  
endmodule
```

Structural Modelling

```
module t_st(q,q1,t,c);  
    output q,q1;  
    input t,c;  
    wire w1,w2;  
    assign w1=t&c&q;  
    assign w2=t&c&q1;  
    assign q=~(w1|q1);  
    assign q1=~(w2|q);  
endmodule
```

Dataflow Modelling

```
module t_df(q,q1,t,c);  
    output q,q1;  
    input t,c;  
    and g1(w1,t,c,q);  
    and g2(w2,t,c,q1);  
    nor g3(q,w1,q1);  
    nor g4(q1,w2,q);  
endmodule
```

D Flip Flop

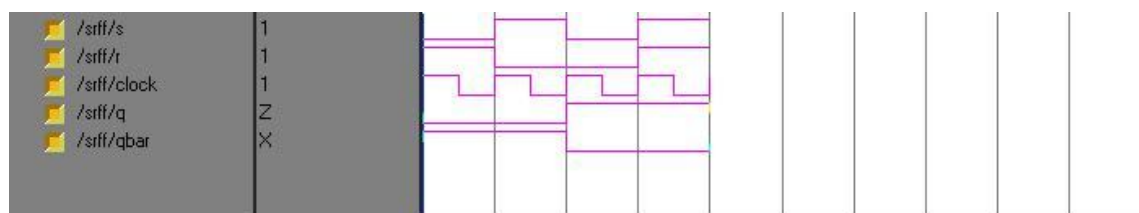
<i>Behavioral Modelling</i>	<i>Dataflow Modelling</i>	<i>Structural Modelling</i>
<pre>Module dff_async_reset(data, clk, reset ,q); input data, clk, reset ; output q; reg q; always @ (posedgeclk or negedge reset) if (~reset) begin q <= 1'b0; end else begin q <= data; end endmodule</pre>	<pre>module dff_df(d,c,q,q1); input d,c; output q,q1; assign w1=d&c; assign w2=~d&c; q=~(w1 q1); q1=~(w2 q); endmodule</pre>	<pre>module dff_df(d,c,q,q1); input d,c; output q,q1; and g1(w1,d,c); and g2(w2,~d,c); nor g3(q,w1,q1); nor g4(q1,w2,q); endmodule</pre>

JK Flip Flop

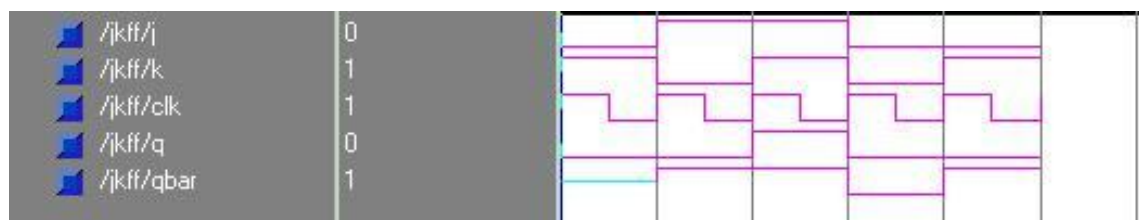
Behavioral Modelling	Dataflow Modelling	Structural Modelling
<pre> module jk(q,q1,j,k,c); output q,q1; input j,k,c; reg q,q1; initial begin q=1'b0; q1=1'b1; end always @ (posedge c) begin case({j,k}) {1'b0,1'b0}:begin q=q; q1=q1; end {1'b0,1'b1}: begin q=1'b0; q1=1'b1; end {1'b1,1'b0}:begin q=1'b1; q1=1'b0; end {1'b1,1'b1}: begin q=~q; q1=~q1; end endcase end endmodule </pre>	<pre> module jkflip_df (j,k,q,qn); input j,k,q; output qn; wire w1,w2; assign w1=~q; assign w2=~k; assign qn=(j & w1 w2 & q); endmodule </pre>	<pre> module jkflip_st(j,k,q,qn); input j,k,q; output qn; and g1(w1,j,~q); and g2(w2,~k,q); or g3(qn,w1,w2); endmodule </pre>

Output Waveforms :

SR flip flop



JK flip flop



D flip flop



T flip flop



Result:

Flip flops were Designed and implemented.

Ex. No: 13

Date:

FPGA IMPLEMENTATION OF TRAFFIC LIGHT CONTROLLER

AIM:

To design a Traffic Light Controller using Verilog code and to test its working on the FPGA Board.

TOOLS REQUIRED:

SOFTWARE:

XILINX ISE 6.1i

HARDWARE:

XILINX - Spartan kit XC3S400TQ144, Power supply Adapter, Parallel port cable, FRC connector, AU card - II

PROGRAM:

Verilog Code for Traffic Light Controller

```
module traffic_verilog(seg_1,seg_2,R1,R2,R3,G1,G2,G3,Y1,Y2,Y3,PR,PG,clk,rst); output
reg R1,R2,R3,G1,G2,G3,Y1,Y2,Y3,PR,PG;
output reg
[6:0]seg_1,seg_2; input
clk,rst;
integer timer_count1 = 0,timer_count2 =
0; reg clk_msec,clk_sec;
reg [7:0]count;
reg [1:0]state = 2'b0;

always@(posedge clk)
begin

if(timer_count1==3999)
begin
timer_count1=0;
clk_msec=1'b1;
end
else
begin
timer_count1=timer_count1+
1; clk_msec=1'b0;
end

end
```

```

always@(posedge clk_msec)
begin

    if(timer_count2==999)
    begin
        timer_count2=0;
        clk_sec=1'b1;
    end
    else
    begin
        timer_count2=timer_count2+
        1; clk_sec=1'b0;
    end
end

always@(posedge clk_sec)
begin
    if(~rst)
    begin
        R1 = 1'b1; G1 = 1'b0; Y1 = 1'b0;
        R2 = 1'b1; G2 = 1'b0; Y2 = 1'b0;
        R3 = 1'b1; G3 = 1'b0; Y3 = 1'b0;
        state=2'b00;
    end
    else
    begin
        case(state)
        2'b00://SIGNAL AT SIGNAL LIGHTS ONE
        begin

            if(count==8'b00100101)
            begin
                G1 = 1'b0;
                R1 = 1'b0;
                Y1 = 1'b1;
                R3 = 1'b0;
                Y3 = 1'b0;
                G3 = 1'b0;
            end
            if(count==8'b00101001)
            begin
                G1 = 1'b1;
                Y1 = 1'b0;
                R3 = 1'b1;
                state=2'b01;
            end
            else
            e
            state=2'b00
        ; end
    end
end

```



```

2'b01://SIGNAL AT SIGNAL LIGHTS TWO
begin
if(count==8'b00100101)
begin
Y1 = 1'b1;

G1 = 1'b0;
R3 = 1'b1;
R2 = 1'b0;
Y2 = 1'b1;
G2 = 1'b0;

end
if(count==8'b00101001)
begin
R1 = 1'b1;

Y1 = 1'b0;
Y2 = 1'b0;
G2 = 1'b1; state
= 2'b10;

end
else
state=2'b01;
end

```

```

2'b10://SIGNAL AT SIGNAL LIGHTS THREE begin
if(count==8'b00100101)
begin
Y2 = 1'b1;

G2 = 1'b0;
R3 = 1'b0;
Y3 = 1'b1;
G3 = 1'b0;

end
if(count==8'b00101001)
begin
R2 = 1'b1;

Y2 = 1'b0;
Y3 = 1'b0;
G3 = 1'b1;
state = 2'b11;

end
else
state=2'b10;
end
2'b11://ALL SIGNAL HIGH TO ALLOW PEDESTRIALS TO CROSS begin
if(count==8'b00100101)

begin
Y1= 1'b0;

Y3 = 1'b1;

```

```

                                G3 = 1'b0;

    end
    if(count==8'b00101001)
    begin

                                Y1 =1'b0;

        R3 = 1'b1;

                                Y3 = 1'b0;
                                state = 2'b00;

        en
        d
        else
            state=2'b11
        ; end
    endcas
    e end
end

always@(count,state)
begin
    if((state==2'b00)&&(count<=8'b00101001))
    begin
        PR = 1'b1;

                                PG = 1'b0;

        end
    else
        begin
            PR = 1'b0;

                                PG = 1'b1;

        en
    d end

always@(posedge clk_sec)
begin
    if(rst==1'b0)
        count=8'b00000000
    ; else if(clk_sec)
        begin
            if(count[3:0]==4'b1001)
            begin
                count[3:0]=4'b0000;

                                if(count[7:4]==4'b0010)
                                count[7:4]=4'b0000;

            else
                count[7:4]=count[7:4]+1;

            end
        end
    end
end

```

```

        else
            count[3:0]=count[3:0]+1;
        end
    end

    always@(count)
    begin
        case(count[3:0])
            4'b0000:seg_1 = 8'b11111110;
            //0
            4'b0001:seg_1 = 8'b0110000; //1
            4'b0010:seg_1 = 8'b1101101; //2
            4'b0011:seg_1 = 8'b1111001; //3
            4'b0100:seg_1 = 8'b0110011; //4
            4'b0101:seg_1 = 8'b1011011; //5
            4'b0110:seg_1 = 8'b1011111; //6
            4'b0111:seg_1 = 8'b1110000; //7
            4'b1000:seg_1 = 8'b1111111; //8
            4'b1001:seg_1 = 8'b1111011; //9
            default:seg_1 = 8'b0000000;
            //off
        endcase

        case(count[7:4])
            4'b0000:seg_2 = 8'b11111110; //0
            4'b0001:seg_2 = 8'b0110000; //1
            4'b0010:seg_2 = 8'b1101101; //2
            4'b0011:seg_2 = 8'b1111001; //3
            4'b0100:seg_2 = 8'b0110011; //4
            4'b0101:seg_2 = 8'b1011011; //5
            4'b0110:seg_2 = 8'b1011111; //6
            4'b0111:seg_2 = 8'b1110000; //7
            4'b1000:seg_2 = 8'b1111111; //8
            4'b1001:seg_2 = 8'b1111011; //9
            default:seg_2 = 8'b0000000;
            //off
        endcase
    end
endmodule

```

UCF file (User Constraint File)

```

NET "clk" LOC = "p52" ;
NET "G1" LOC = "p97" ;
NET "G2" LOC = "p98" ;
NET "G3" LOC = "p125" ;
NET "PG" LOC = "p132" ;
NET "PR" LOC = "p135" ;
NET "R1" LOC = "p93" ;
NET "R2" LOC = "p104" ;

```

```

NET "R3" LOC = "p129" ;
NET "rst" LOC = "p78" ;
NET "seg_1<6>" LOC = "p27" ;
NET "seg_1<5>" LOC = "p26" ;

NET "seg_1<4>" LOC = "p24" ;
NET "seg_1<3>" LOC = "p23" ;
NET "seg_1<2>" LOC = "p21" ;
NET "seg_1<1>" LOC = "p18" ;
NET "seg_1<0>" LOC = "p17" ;
NET "seg_2<6>" LOC = "p15" ;
NET "seg_2<5>" LOC = "p14" ;
NET "seg_2<4>" LOC = "p13" ;
NET "seg_2<3>" LOC = "p12" ;
NET "seg_2<2>" LOC = "p1" ;
NET "seg_2<1>" LOC = "p83" ;
NET "seg_2<0>" LOC = "p80" ;
NET "Y1" LOC = "p95" ;
NET "Y2" LOC = "p99" ;
NET "Y3" LOC = "p122" ;

```

PROCEDURE:

Software part

1. Click on the Xilinx ISE9.1i or Xilinx Project navigator icon on the desktop of PC.
2. Write the Verilog code, check syntax, view RTL schematic and note the device utilization summary by double clicking on the synthesis in the process window.
3. Open a new UCF file and lock the pins of the design with FPGA I/O pins.
4. Implement the design by double clicking on the implementation tool selection.
5. Create programming file (i.e., bit file) for downloading into the device.

Hardware part

Connect the power supply cable to the FPGA kit using power supply adapter.

Connect the FPGA kit to the parallel port of the PC through the cable provided along with the kit.

Connect FRC1 of main board to CN2 of AU card - II using FRC cable.

Connect FRC8 of main board to CN3 of AU card - II using FRC cable.

Connect FRC5 of main board to CN1 of AU card - II using FRC cable.

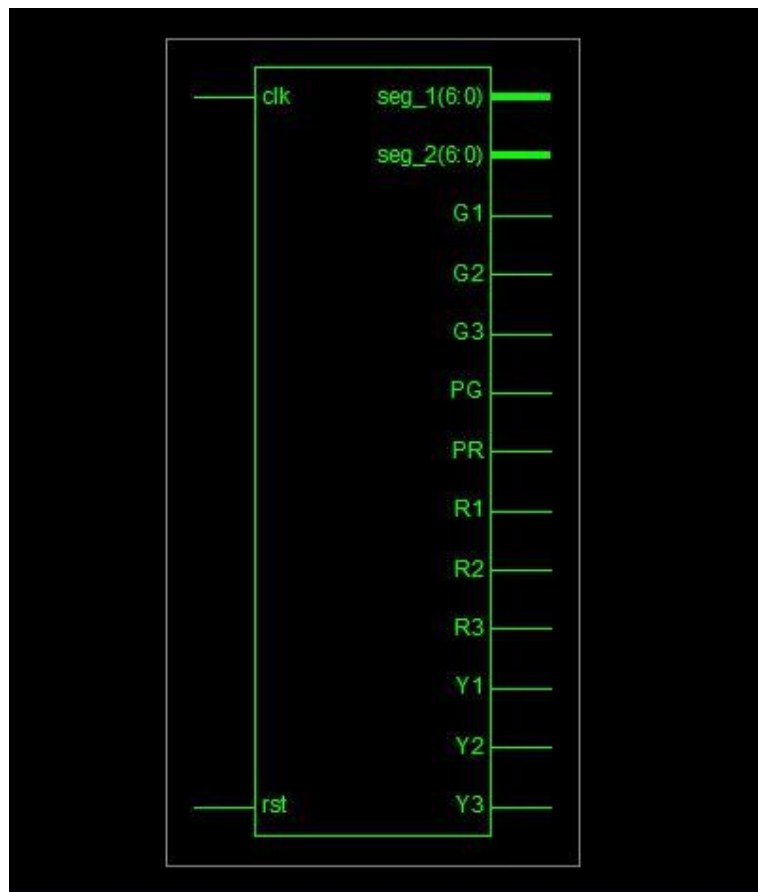
Working:

1. Connections are made as above and implement the design into the FPGA device and the Traffic Light Controller (AU card – II) will be ON.
2. Initially, the Pedestrians on all the three signals will be in GREEN for 30 seconds to allow the Pedestrians to pass through.
3. After 30 seconds, the pedestrian goes to RED and the GREEN in signal1 goes ON for 25 seconds.
4. After 25 seconds, the YELLOW will be ON for 5 seconds and then it goes to RED, simultaneously the GREEN in signal2 will be ON.
5. Similar procedure is repeated for all the signals. Then the procedure is repeated again by switching the GREEN in all the Pedestrians.

SYNTHESIS REPORT:**Device Utilization Summary (estimated values)**

Logic Utilization	Used	Available	Utilization
Number of Slices	78	3584	2%
Number of Slice Flip Flops	87	7168	1%
Number of 4 input LUTs	151	7168	2%
Number of bonded IOBs	27	97	27%
Number of GCLKs	3	8	37%

RTL Schematic Representation – Top Level



RESULT:

Thus the traffic light controller was designed using verilog code and its working was tested in FPGA board.