Lab-10

Introduction to Verilog HDL

COMBINATIONAL CIRCUITS IN VERILOG HDL

10.1 Lab Objectives:

- Understand the fundamental concepts of Verilog Hardware Description Language (HDL).
- Learn the syntax and structure of Verilog code for designing digital circuits.
- Implement basic logic gates (AND, OR, NOT) using Verilog and simulate their behavior.
- Design and simulate combinational circuits such as multiplexers, decoders, and encoders in Verilog.
- Gain proficiency in using Verilog modeling techniques for sequential circuits.
- Develop skills in writing testbenches to verify the functionality of Verilog designs.
- Understand the concept of waveform simulation and analyze simulation results.
- Learn how to use Verilog for simple arithmetic and data manipulation tasks.
- Explore the concept of gate-level modeling and its applications in Verilog.
- Practice documenting Verilog code and creating clear design descriptions for future reference and collaboration.

10.2 Verilog:

Verilog is the combination of the terms *"Verification"* and *"Logic"*. It is hardware description language or a special type of programming language which describes the hardware implementations of digital systems and circuits. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits.

Verilog module:

Verilog provides the concept of a module. A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks.

In Verilog, a module is declared by the keyword module. A corresponding keyword endmodule must appear at the end of the module definition. Each module must have a module_name, which

is the identifier for the module, and a module_terminal_list, which describes the input and output terminals of the module.

module <module_name> (<module_terminal_list>); //outputs first then inputs

...

<module internals> // Also called module body.

...

endmodule

Verilog is case sensitive.

Reserved keywords are written in lower cases.

A semicolon is used to terminate the statement.

The comment rule is same as C Programming Language.

Purpose of a module:

A module represents a design unit that implements certain behavioral characteristics and well get converted into a digital circuit during synthesis. Any combination of the inputs can be given to the module, and it will provide a corresponding output. This allows the same module to be reused to form bigger modules that implement more complex hardware.

Single line comment starts with "//".

For example – //Example of a Verilog single line comment

Multiline comments start with – ' /*' and end with '*/'.

For example –

/* Example

Of Verilog multiple

Line comment*/

Verilog Data Types:

The primary intent of data types in the Verilog language is to represent data storage elements like bits in a flipflop and transmission elements like wires in that connects logic gates and sequential circuits.

Verilog Operators:

Verilog has three fundamental operators for Verilog HDL. They are given below.

Unary Verilog operators: These types of Verilog operators come first of the operands.

For example: x = ~ y; Here '~' is a unary operator

Binary Verilog operators: These types of Verilog operators come in between two operands.

For example: x = y || z; Here '||' is a binary operator.

Ternary Verilog operators: These types of Verilog operators use two different operators to differentiate three operators.

For example: x = y? z: w; here '?' and ':' are ternary operators.

Verilog HDL's categorical operators are – arithmetical, logical, relational, bitwise, shift, concatenation, and equality. Different types of Verilog operators and their symbols are given in the below table.

10.6 Dataflow Modeling: Dataflow modeling provides the means of describing combinational circuits by their function rather than by their gate structure. Dataflow modeling uses several operators that act on operands to produce the desired results. Verilog HDL provides about 30 operator types.

Type of Operator	Symbol	Operation	Operands needed
Arithmetic	*	*Multiplication*	*Two*
	/	*Division*	*Two*

	+	*Addition*	*Two*
	–	*Subtraction*	*Two*
	%	*Modulus*	*Two*
Logical	!	*Negation*	*One*
	&&	*AND*	*Two*
	||	*OR*	*Two*
Relational	>	*Greater than*	*Two*
	<	*Less than*	*Two*
	> =	*Greater than or equal to*	*Two*
	< =	*Less than or equal to*	*Two*
Equality	==	*Equals to*	*Two*
	! =	*Not equals to*	*Two*
	===	*Case equal*	*Two*
	! ==	*Case not equal*	*Two*
Bitwise	~	*Negation*	*One*
	&	*Bitwise AND*	*Two*
	|	*Bitwise OR*	*Two*

	^	*Bitwise XOR*	*Two*
	~^	*Bitwise XNOR*	*Two*
Reduction	&	*Reduction AND*	*One*
	~&	*Reduction NAND*	*One*
	|	*Reduction OR*	*One*
	~|	*Reduction NOR*	*One*
	^	*Reduction XOR*	*One*
	^~	*Reduction XNOR*	*One*
Shift	>>	*Right Shift*	*Two*
	<<	*Left Shift*	*Two*
Concatenation	{ }	*Concatenation*	*Can be of any numbers*
Replication	{ { } }	*Replication*	*Can be of any numbers*
Conditional	? :	*Conditional*	*Three*

Verilog Number Specifications

Verilog numbers are of two types, sized numbers and unsized numbers.

Sized Verilog numbers: The general structure for representing sized numbers in Verilog HDL is given below.

<size>'<base format><numbers>

For example – 8'b3456.

size: Size is the number of digits the main number has. Size is described using decimal values.

base format: Base format suggests which type of number it would be. There are several types – binary (given by – 'b'), decimal (given by – 'd'), octal (given by – 'o'), hexadecimal (given by – 'h'). If there is no specification for base format, then by default it is a decimal number.

numbers: The main number you want to put in.

Unsized Verilog numbers: These numbers do not require any specified size.

The general structure for representing unsized numbers in Verilog HDL is given below.

'<base format><numbers>

For example – 'h3456.

This is an unsized Verilog number which describes that it is a hexadecimal number.

Negative Numbers: If you want to declare a number as a negative number, then put a minus symbol (-) before the number.

For example: – 345; it is a negative, unsized, decimal number.

Starting a New Project

1. To create a new project, open Project Navigator either from the Desktop icon or by selecting

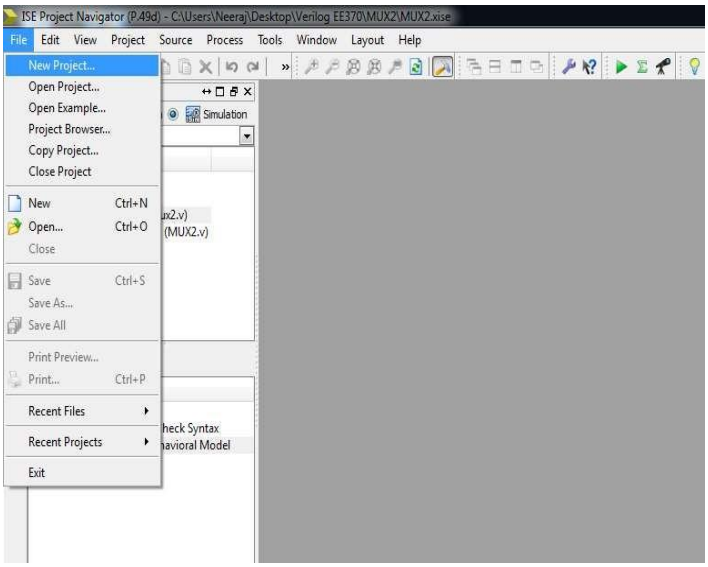
Start > Programs > Xilinx ISE Design Suite 11 > ISE > Project Navigator. In Project Navigator,

Select the New Project option from the Getting Started menu (or by selecting Select File > New Project).

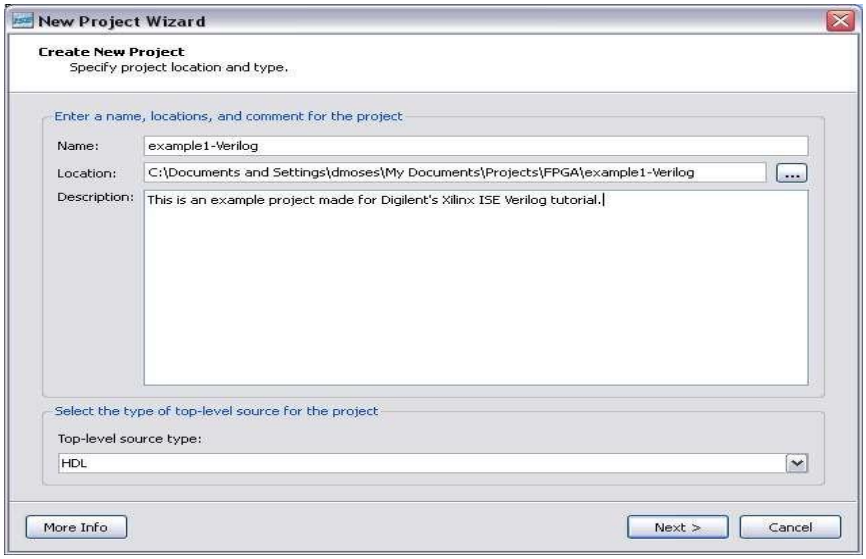
This brings up a Dialog box where you can enter the desired project name and project location.

Choose a meaningful name for easy reference. Here we call this project "example1-Verilog" and save it in a local directory. You can place comments for your project in the Description text box. We use HDL for our top-level source type in this tutorial.

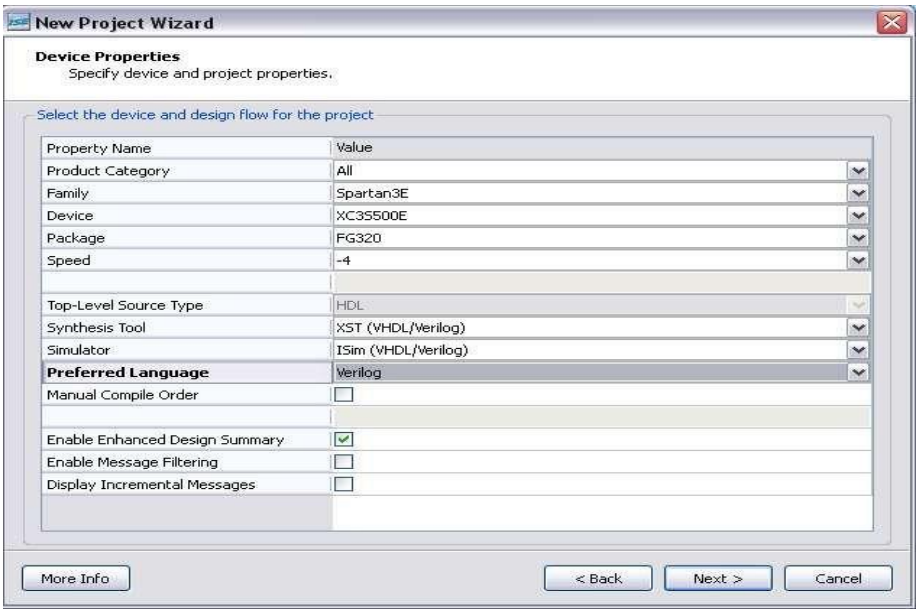
Open New Project



Choose the location to create a New Project. Always choose something other than User folder as it is read only. The best way is to create a folder on desktop, name it and copy its url and paste it in the “location” of the “new project wizard”.

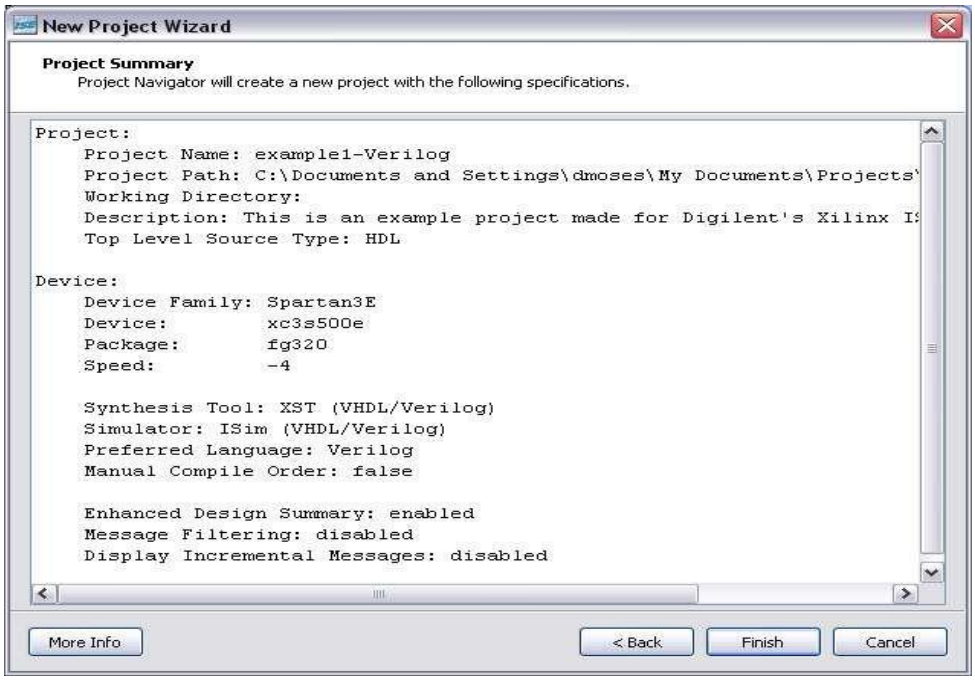


2. The next step is to select the proper Family, Device, and Package for your project. This depends on the chip you are targeting for this project. The appropriate settings for a project suited for the Nexys2 500k board are as follows:



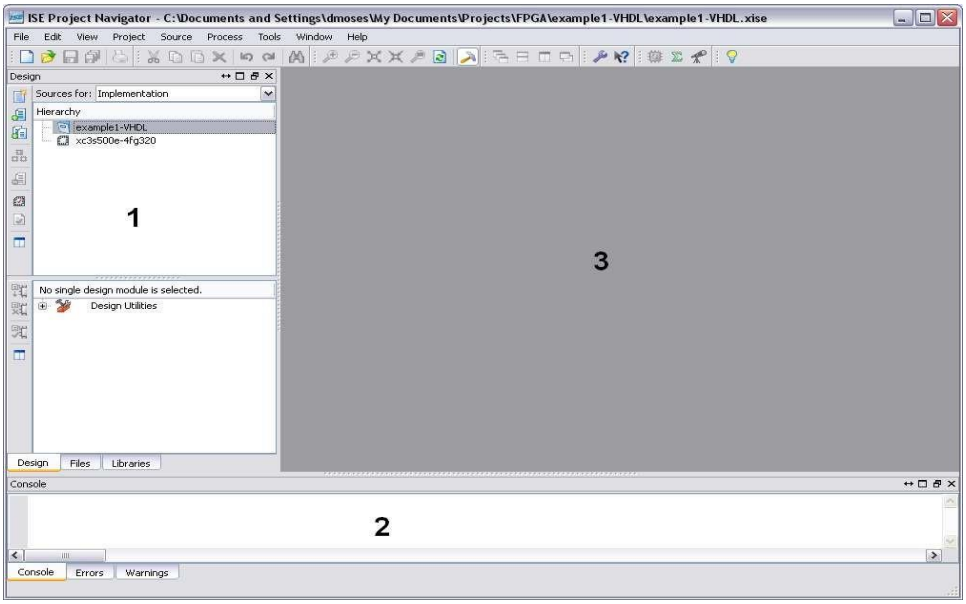
Since this lab does not require physically burn the logic on a chip so we will leave this window as it is. Just click Next.

3. Once the appropriate settings have been entered, click Next. The next two dialog boxes give you the option of adding new or existing source files to your project. Since we will fulfill these steps later, click Next without adding any source files.



Project Navigator Overview

Once the new project has been created, ISE opens the project in Project Navigator. Click the Design tab to show the Design panel and click the Console tab to show the Console panel.



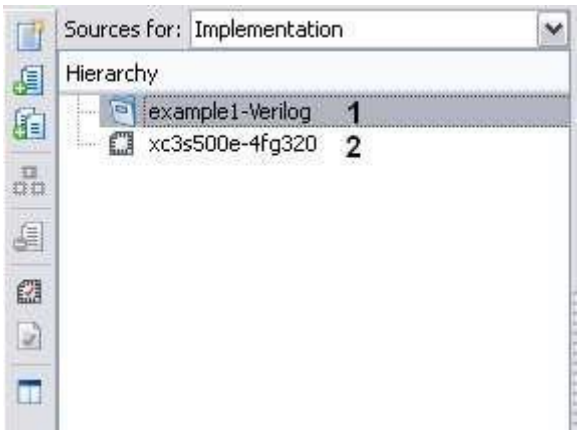
The Design panel (1) contains two windows: a Sources window that displays all source files associated with the current design and a Process window that displays all available processes that can be run on a selected source file.

The Console panel (2) displays status messages including error and warning messages.

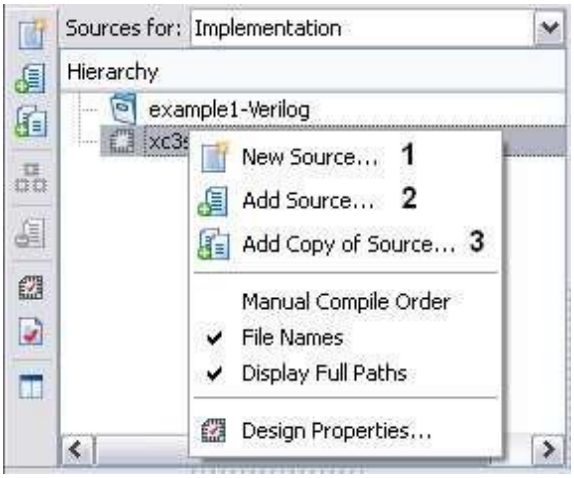
The HDL editor window (3) displays source code from files selected in the Design panel.

Adding New Source Files

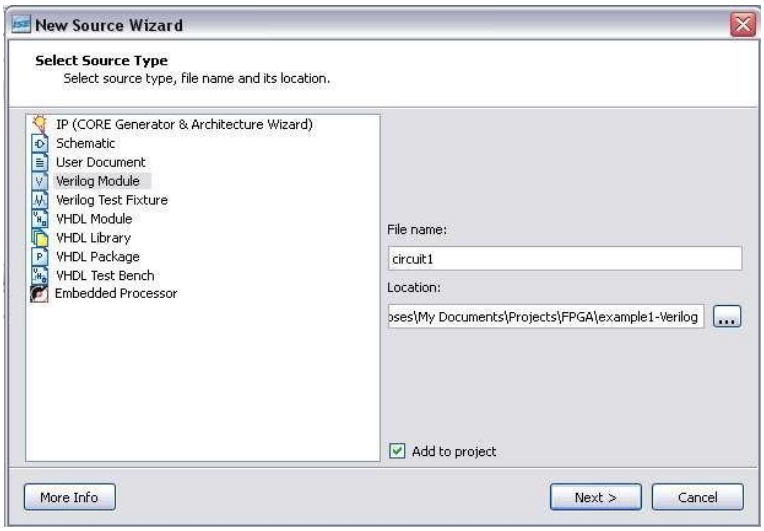
Once the new project is created, two sources are listed under sources in the Design panel: the Project file name and the Device targeted for design.



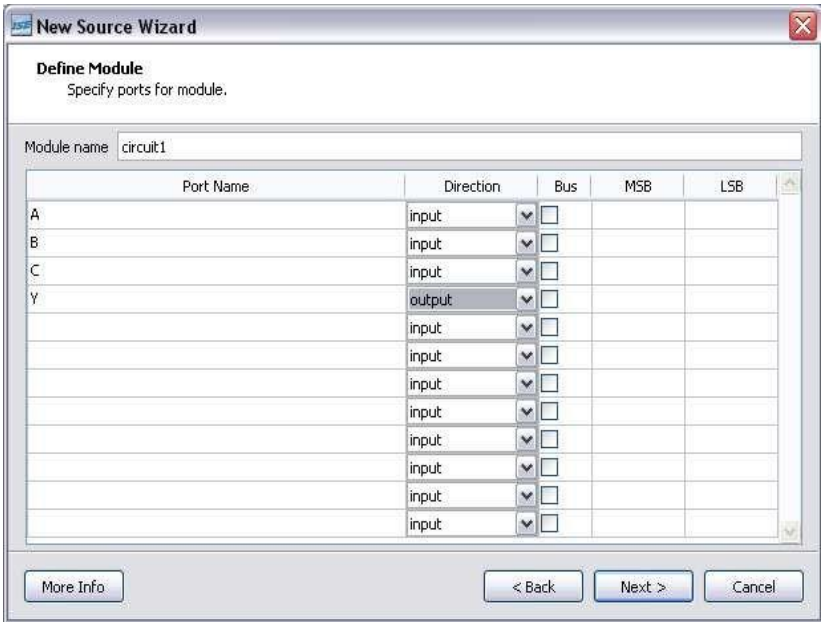
You can add a new or existing source file to the project. To do this, right-click the target device and select one of the three options for adding source files.



We create a new source file, so select New Source from the list. This starts the New Source Wizard, which prompts you for the Source type and file name. Select Verilog Module and give it a meaningful name e.g. circuit1.



When you click Next, you have the option of defining top-level ports for the new Verilog module. We chose A, B, and C as input ports and Y as an output port.



HDL Editor Window

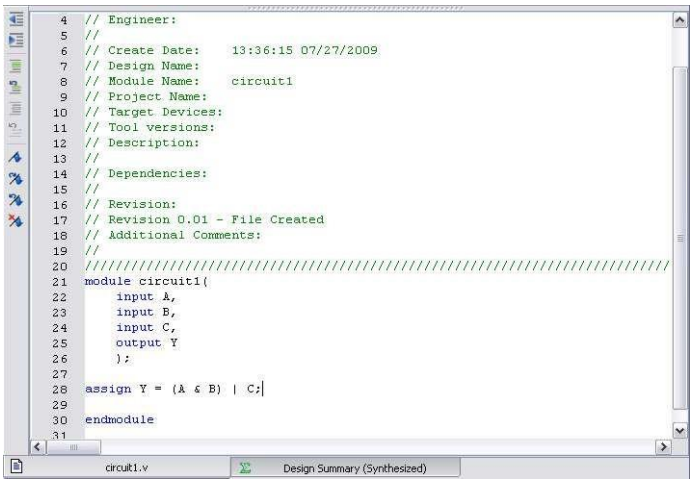
Once you have created the new Verilog file, the HDL Editor Window displays the circuit1.v source code.

The Xilinx tools automatically generate lines of code in the file to get you started with circuit development. This generated code includes:

1. a module statement

2. a comment block template for documentation

The actual behavioral or structural description of the given circuit is to be placed between the "module" and "endmodule" statements in the file. Between these statements, you can define any Verilog circuit you wish. In this tutorial, we use a simple combinational logic example, and then show how it can be used as a structural component in another Verilog module. We start with the basic logic equation: Y= (A·B) + C.



If there are any syntax errors in the source file, ISE can help you find them. For example, the parentheses around the A and B inputs are crucial to this implementation; without them, ISE would return an error during synthesis info.

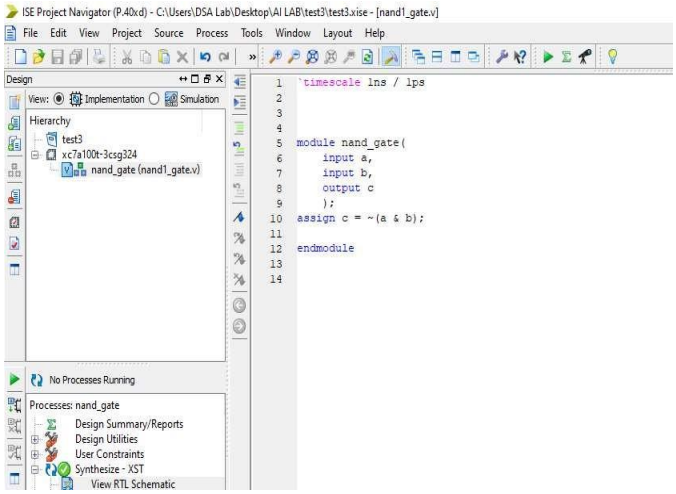
RTL Schematics and Syntax Checking

The RTL schematics can be viewed using the following set of steps, the example used here is that of a Nand gate. Double click on synthesize-XST, on the left-hand side.

Example of NAND gate in Verilog

Code

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



To create a Test bench, create New Source. Select Verilog Test Fixture. Template of Test bench will be created instantiating the Nand gate module.

Stimulus/Testbench

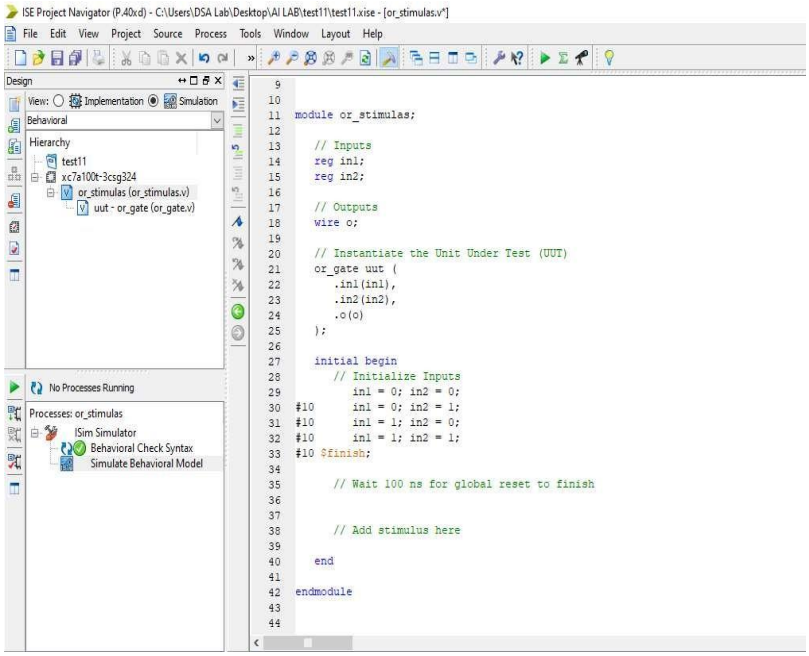
Code

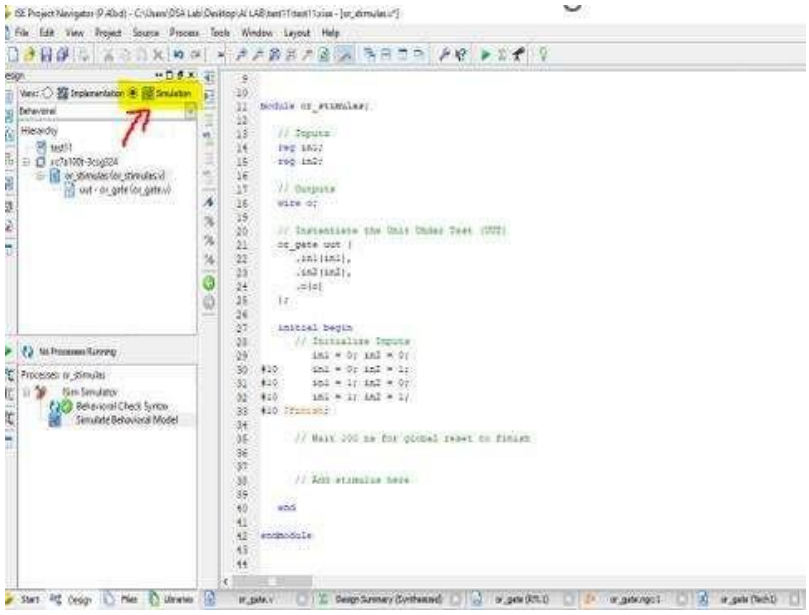
```
module nand1_stimulas;
reg a;
reg b;
wire c;
```

```
        // Instantiate the Unit Under Test (UUT)
nand_gate uut (
                .a(a),
                .b(b),
                .c(c)
        );
initial begin
                // Initialize Inputs
a = 0; b = 0;
#10   a = 0; b = 1;
#10   a = 1; b = 0;
#10   a = 1; b = 1;
#10 $finish;
end
endmodule
```

Initial block: Initial blocks cause particular instructions to be performed at the beginning of the simulation before any other instructions operate. Initial blocks only operate once.

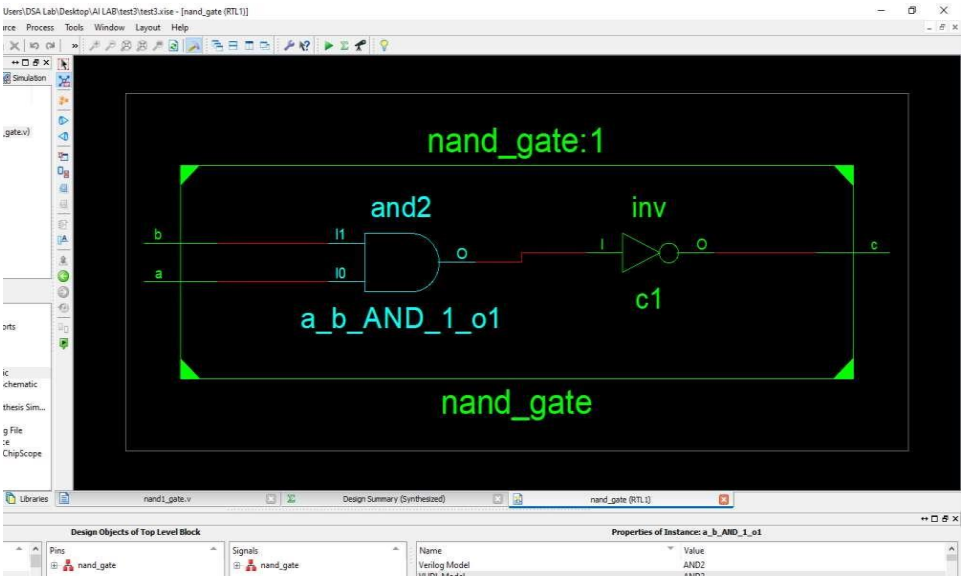
Verilog test benches are used for the verification of the digital hardware design. Verification is required to ensure the design meets the timing and functionality requirements. **Verilog Test benches** are used to simulate and analyze designs without the need for any physical hardware or any hardware device.





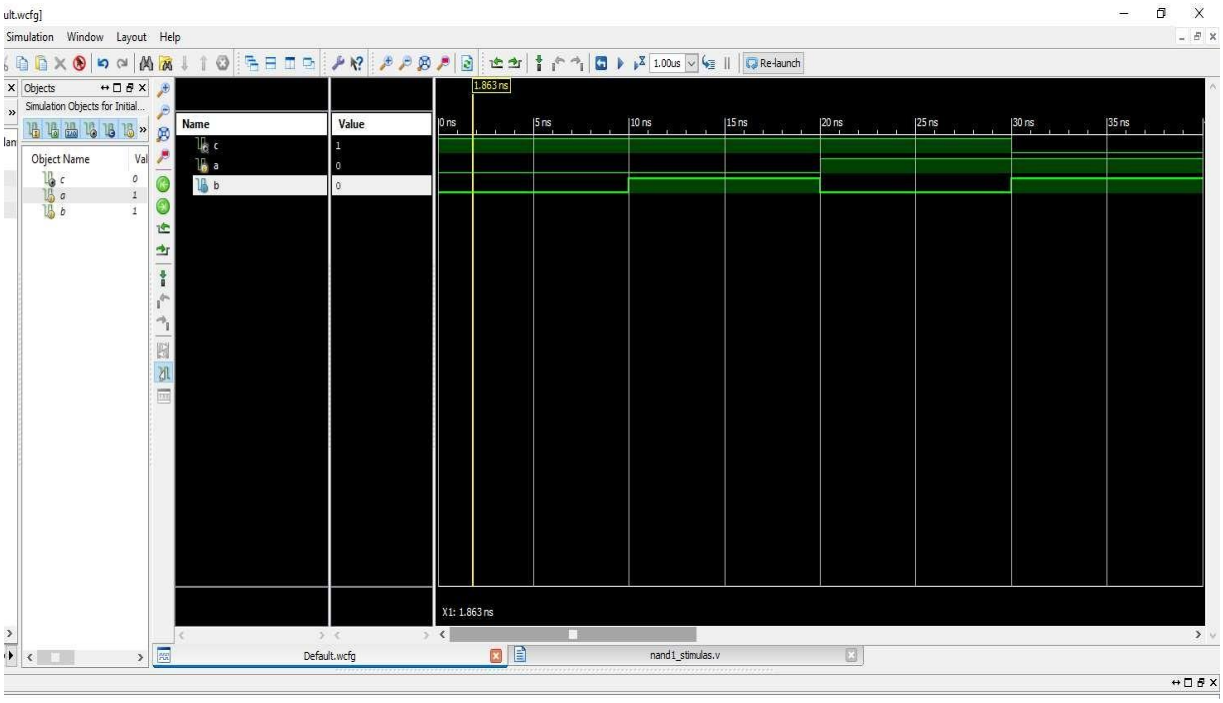
Double click on Simulate Behavioral Model option.

RTL



OUTPUT

This is the simulation window. You can verify the working using waveforms.



10.3 Verilog Gate Level modelling.

In Verilog, most of the digital designs are done at a higher level of abstraction like RTL. However, it becomes natural to build smaller deterministic circuits at a lower level by using combinational elements such as AND and OR. Modeling done at this level is called gate-level modeling as it involves gates and has a one-to-one relationship between a hardware schematic and the Verilog code. Verilog supports a few basic logic gates known as primitives, as they can be instantiated, such as modules, and they are already predefined.

Gate level modeling is virtually the lowest level of abstraction because switch-level abstraction is rarely used. Gate level modeling is used to implement the lowest-level modules in a design, such as multiplexers, full-adder, etc. Verilog has gate primitives for all basic gates.

Gate Types

A logic circuit can be designed using logic gates. Verilog supports basic logic gates as predefined primitives. These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition. All logic circuits can be designed by using basic gates. There are two classes of basic gates: and/or gates and buf /not gates.

And/ Or Gates

And/or gates have one scalar output and multiple scalar inputs. The first terminal in the list of gate terminals is an output and the other terminals are inputs. The output of a gate is evaluated as soon as one of the inputs changes. And /or gates available in Verilog are shown below.

And, or, xor, nand, nor, xnor

Buf /Not Gates

Buf/not gates have one scalar input and one or more scalar outputs. The last terminal in the port list is connected to the input. Other terminals are connected to the outputs. We will discuss gates that have one input and one output.

Two basic buf/not gate primitives are provided in Verilog.

Buf, not

The corresponding logic symbols for these gates are shown in Figure A. We consider gates with two inputs.

Basic Logic Gates

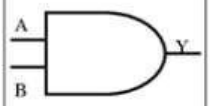
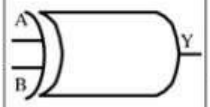
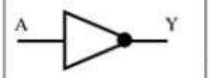
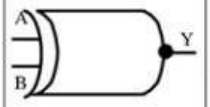
Logic	Schematic	Boolean Expression	Truth Table	English Expression
AND		$A \bullet B = Y$	A B Y 0 0 0 1 1 0 1 1	The only time the output is positive is when all the inputs are positive.
OR		$A + B = Y$	A B Y 0 0 0 1 1 0 1 1	The output will be positive when any one or all inputs are positive.
XOR		$A \oplus B = Y$	A B Y 0 0 0 1 1 0 1 1	The only time the output is positive is when the inputs are not the same.
NOT		$\bar{A} = Y$	A Y 0 1	The output is the opposite of the input.
NAND		$\overline{A \bullet B} = Y$	A B Y 0 0 0 1 1 0 1 1	The output is positive provided all the inputs are not positive.
NOR		$\overline{A + B} = Y$	A B Y 0 0 0 1 1 0 1 1	The only time the output is positive is when all the inputs are negative.
XNOR		$\overline{A \oplus B} = Y$	A B Y 0 0 0 1 1 0 1 1	The only time the output is positive is when all the inputs are the same.

Fig. A. Basic Logic gates

Syntax

```
module and_gate (z,x,y);
input x,y;
output z;
and a1 (z,x,y);
endmodule
```

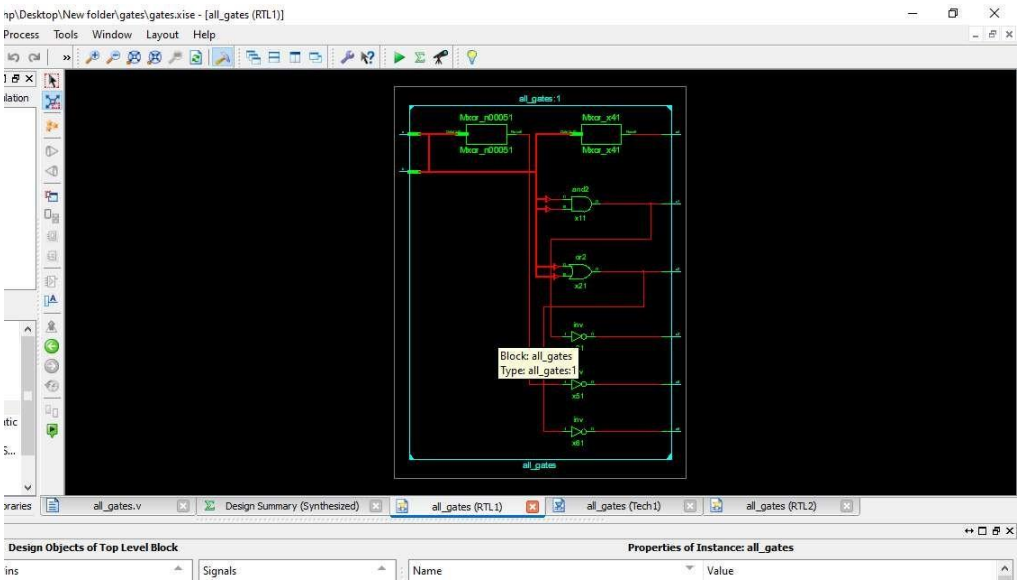
Example # 1 Implementation of all basic gates

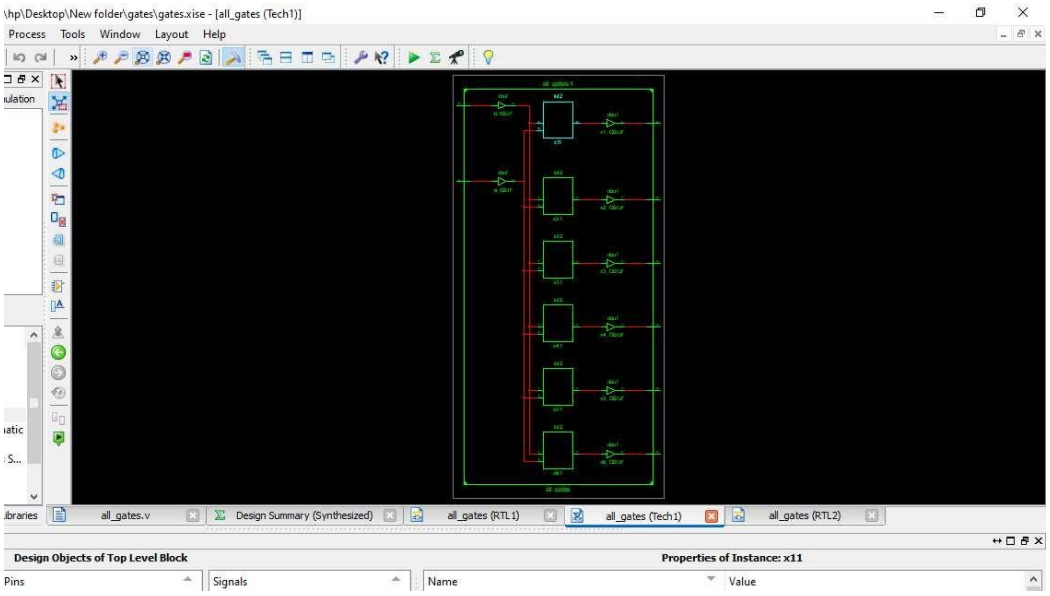
Implement all basic gates (Gate level designing) in Xilinx. implement test benches to verify their working. Visualize their behavior in waveforms.

Code

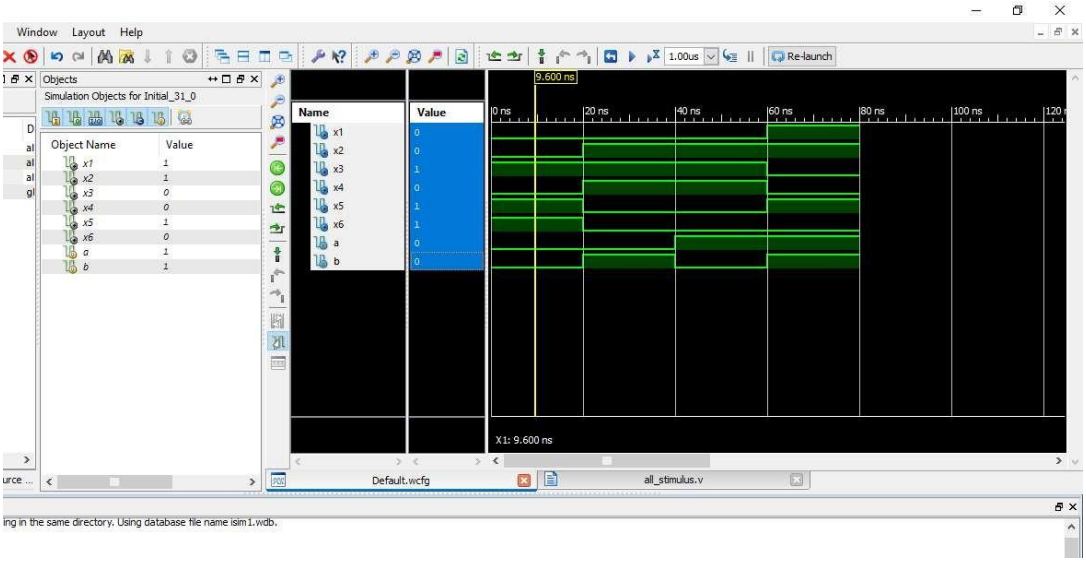
```
module all_gates(
   input a,
   input b,
   output x1,
   output x2,
   output x3,
   output x4,
   output x5,
   output x6
   );
and (x1,a,b);
or  (x2,a,b);
nand (x3,a,b);
xor (x4,a,b);
xnor (x5,a,b);
nor (x6,a,b);
endmodule
```

RTL





Waveform

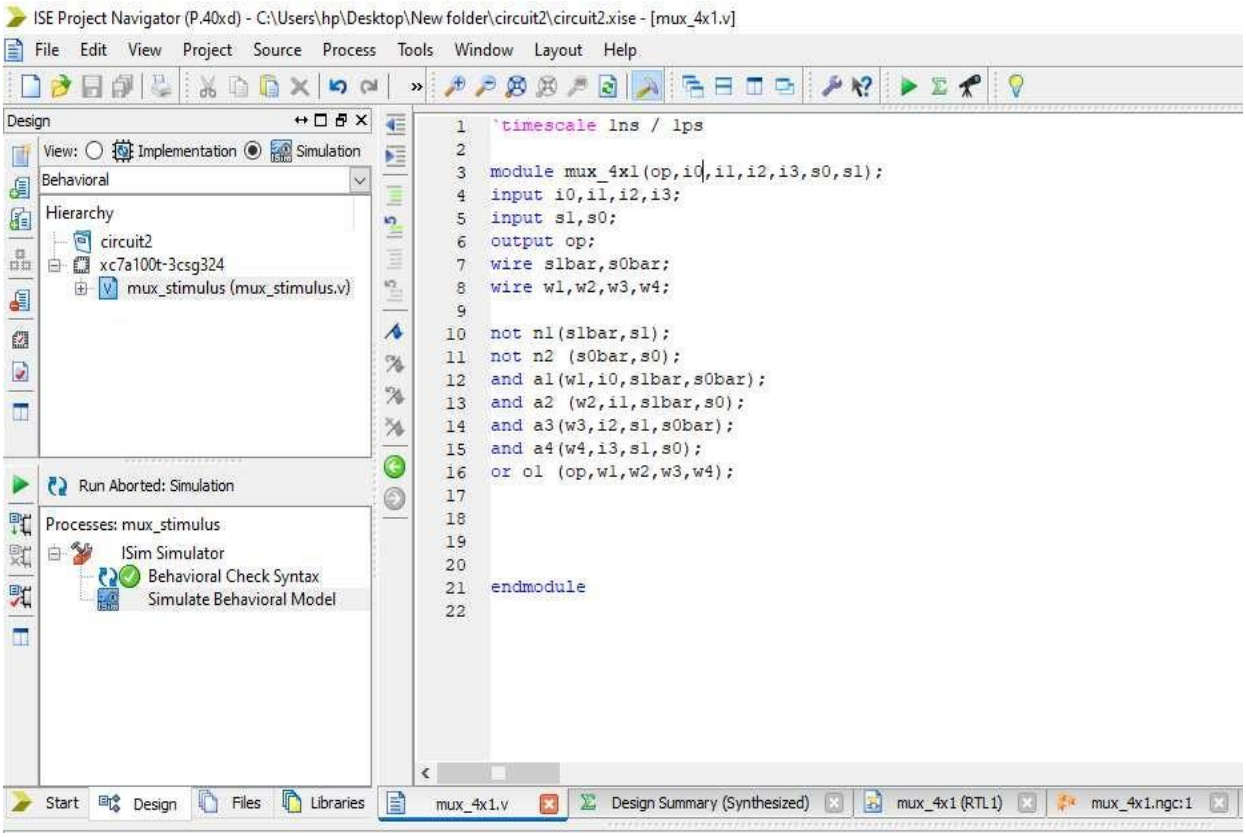


Example # 2

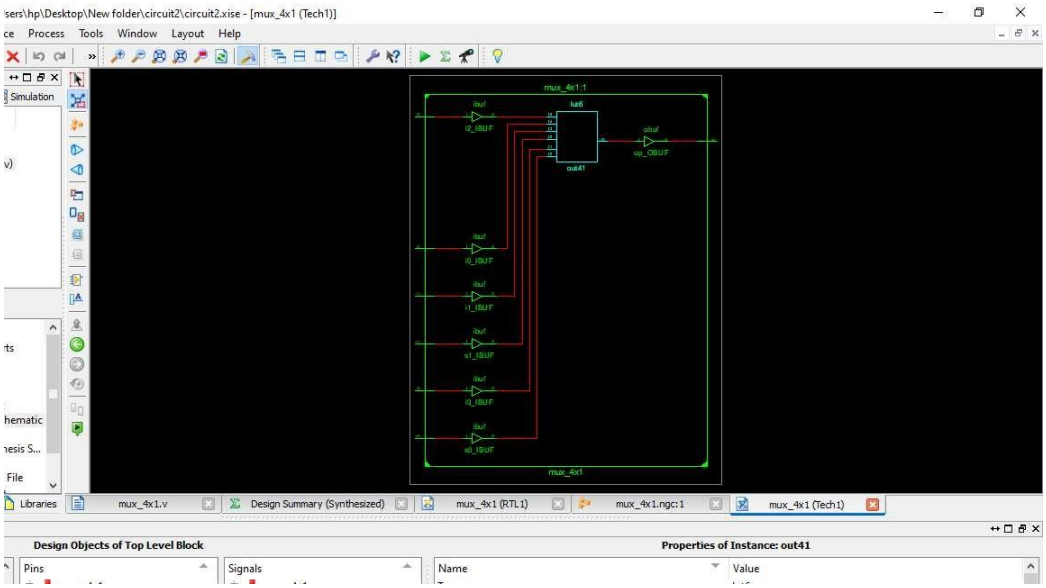
Implement 4x1 MUX (gate level designing) in Xilinx ISE. Implement test benches to verify their working. Visualize their behavior in waveforms.

Code

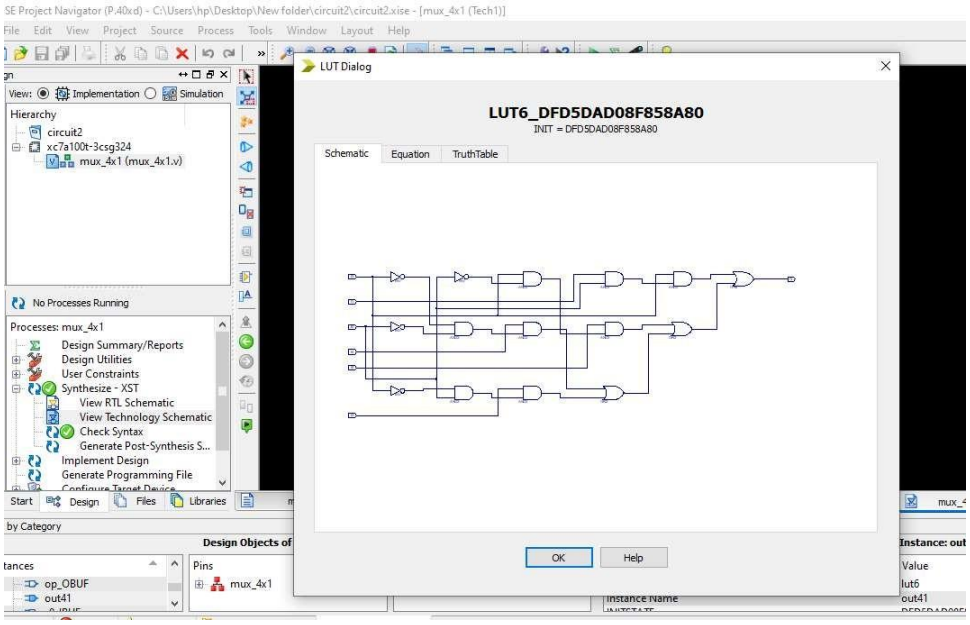
```
module mux_4x1(op,i0,i1,i2,i3,s0,s1);
input i0,i1,i2,i3;
input s1,s0;
output op;
wire s1bar,s0bar;
wire w1,w2,w3,w4;

not n1(s1bar,s1);
not n2 (s0bar,s0);
and a1(w1,i0,s1bar,s0bar);
and a2 (w2,i1,s1bar,s0);
and a3(w3,i2,s1,s0bar);
and a4(w4,i3,s1,s0);
or o1 (op,w1,w2,w3,w4);
endmodule
```



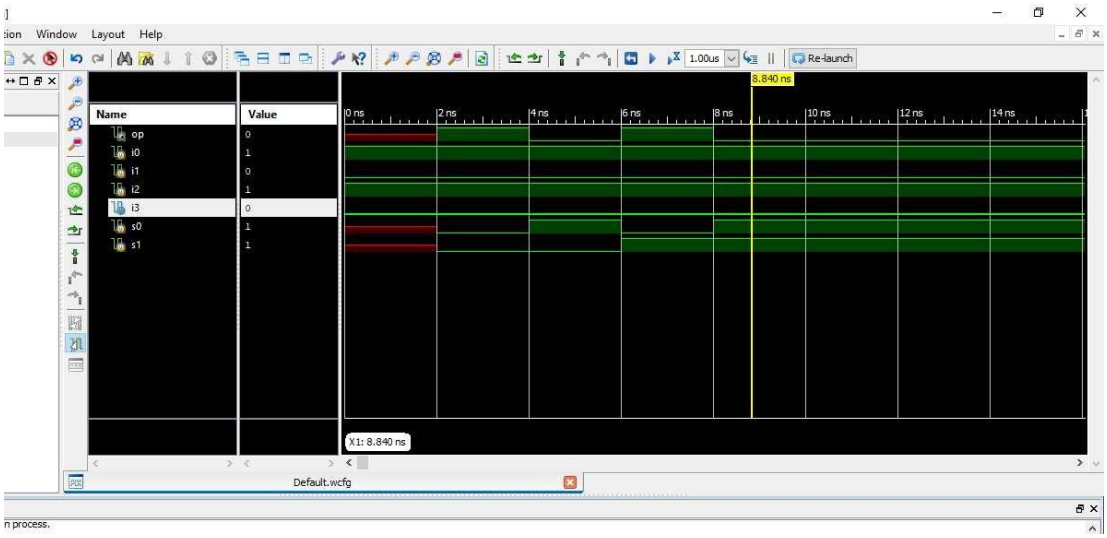
RTL



Schematics



Waveform



Lab Tasks:

A) Implement the following logic gates in Xilinx ISE and implement test benches to verify their working. Visualize their behavior in waveforms.

1. AND
2. OR
3. XOR
4. NOT

B) F = (AB) + (CD)

C) G = (A|B) & (C|D)

D) Implement full adder and implement test benches to verify their working. Visualize their behavior in waveforms.

E) Implement the following in Xilinx (Gate Level Designing). implement test benches to verify their working. Visualize their behavior in waveforms.

I. Full adder
II. Full subtractor
III. 8 X 1 multiplexer
IV. 4 to 2 encoders
V. 4 to 1 demultiplexer

Lab Rubrics

Reg. No. : _____ Date of Lab: _____

	Excellent	Good	Satisfactory	Poor	Score
	3	**2**	**1**	**0**	
Coding Skills	Exceptional coding with minimal errors	Good coding with minor errors	Satisfactory with noticeable errors	Struggles with frequent errors	
Problem Solving	Effectively identifies and solves complex problems	Capably identifies and resolves issues	Struggles with significant assistance	Frequently fails to identify or solve	
Achieving Design Objectives	Design objectives havebeen achieved within 10% of the desired specifications	Design objectives have been achieved within 25% of the desired specifications	Design objectives have been achievedwithin 40% of the desired specifications	Design objectives are not met	
Presentation of Results	Results are presented most accurately	Results are presented accurately	Results are presented much accurately	Results are not presented accurately	
Viva	All the questions are answered correctly with information relevant to topic.	Most of the questions are answered with almost information relevant to topic	Much of the questions are answered but withminimal information relevant to topic	Most of the questions are not answered and information is notrelevant to topic	
Total Score in Lab					**/15**

Instructor Signature: _____

Date: _____