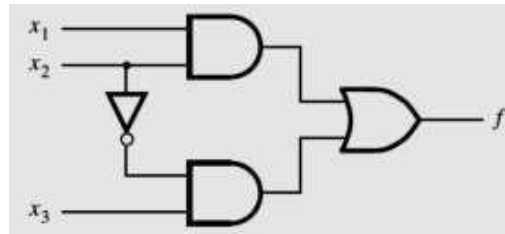


## SAMPLE LAB OBSERVATION NOTE PREPARATION

### Title: Introduction to Verilog

1. Write Verilog code to implement the following circuit using the continuous assignment.

**Aim:** To write Verilog code, Truth table, and waveform for the above circuit.



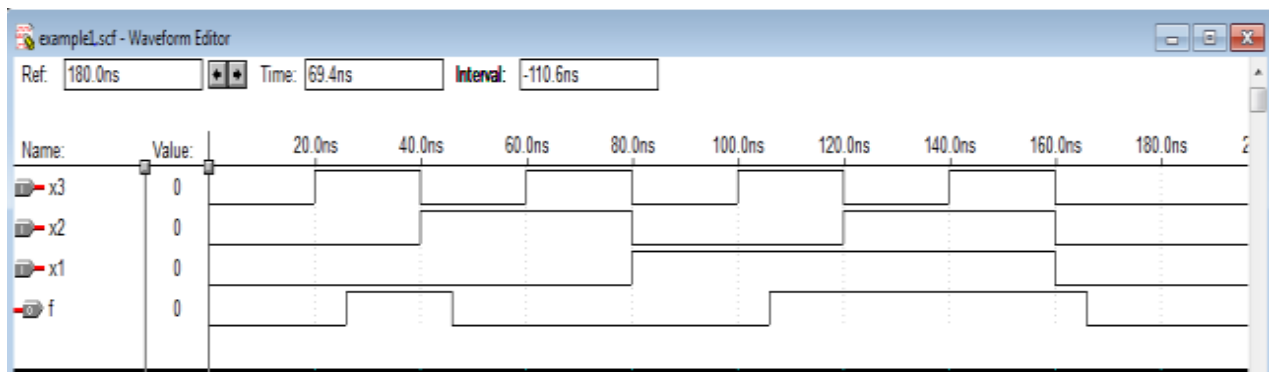
**Verilog code:**

```
module example2(x1,x2,x3,f);  
    input x1,x2,x3;  
    output f;  
    assign f=(x1 & x2)| (~x2 & x3);  
endmodule
```

**Truth table:**

x1	x2	x3	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

**Waveform:**



## Lab No - 1

### INTRODUCTION TO VERILOG

#### I. Basic concepts of Logic Circuits

##### Logic Circuits

- Perform operations on digital signals.
- Signal values are restricted to a few discrete values.
- In binary logic circuits, there are only two values, 0 and 1.

##### Logic Gates and Networks

- Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a logic gate.
- It has one or more inputs and one output that is a function of its inputs.
- It is often convenient to describe a logic circuit by drawing a circuit diagram, or schematic, consisting of graphical symbols representing the logic gates.

The graphical symbols for the AND, NOT and OR gates are shown in Fig. 1.1, 1.2 and 1.3 respectively.

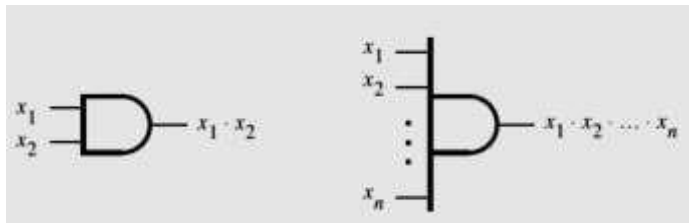


Figure 1.1

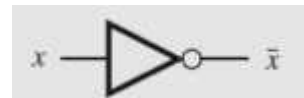


Figure 1.2

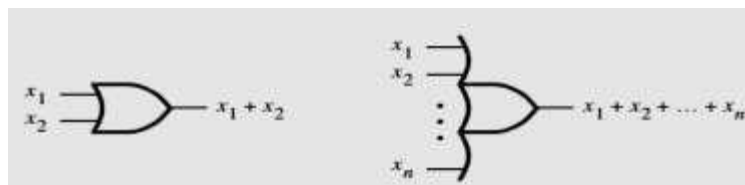


Figure 1.3

- A larger circuit is implemented by a network of gates, as shown in Fig. 1.4

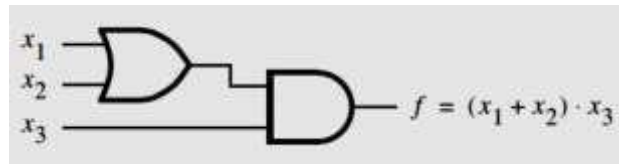


Figure 1.4

$x_1$	$x_2$	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

Figure 1.5 Truth Table

- The operations AND, OR etc can also be defined in the form of a table as shown in Figure 1.5.
- The first two columns (to the left of the heavy vertical line) give all four possible combinations of logic values that the variables  $x_1$  and  $x_2$  can have.
- The next column defines the AND operation for each combination of values of  $x_1$  and  $x_2$ , and the last column defines the OR operation.
- In general, for  $n$  input variables the truth table has  $2^n$  rows.

## II. Analysis of a Logic Network

- Determining the function performed by an existing logic network is referred to as the Analysis process.
- The reverse task of designing a new network that implements a desired functional behavior is referred to as the Synthesis process.
- To determine the functional behavior of the network in Fig. 1.6, we can consider what happens if we apply all possible values to input signals  $x_1$  and  $x_2$ . The analysis of these input values at various intermediate points is shown in Fig. 1.7.

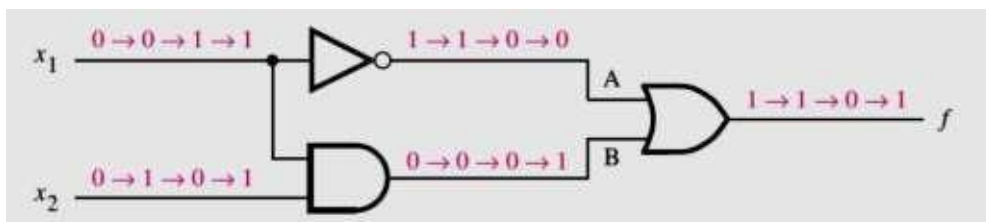


Figure 1.6. functional behavior of the network

$x_1$	$x_2$	<b>A</b>	<b>B</b>	<b>f</b>
0	0	1	0	1
0	1	1	0	1
1	0	0	0	0
1	1	0	1	1

Fig. 1.7. Input values at various intermediate points

## Timing Diagram

- The information in Fig. 1.7 can be presented in graphical form, known as a timing diagram, as shown in Fig. 1.8.
- The figure shows the waveforms for the inputs and output of the network, as well as for the internal signals at the points labeled A and B.

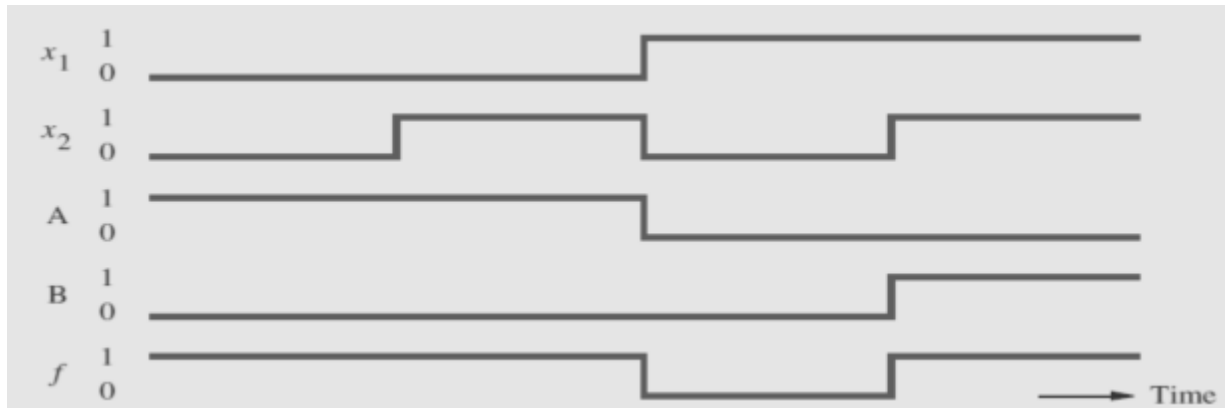


Figure 1.8. Timing diagram of the above circuit

## Functionally Equivalent Networks

- Going through the same analysis procedure, we find that the output 'g' in Fig. 1.9, changes in exactly the same way as f does in Fig. 1.6.
- Therefore,  $g(x_1, x_2) = f(x_1, x_2)$ , which indicates that the two networks are functionally equivalent.

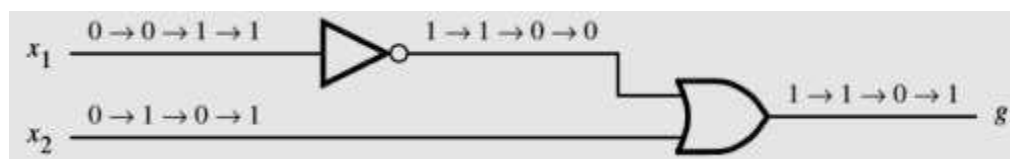


Figure 1.9

## III. Introduction to CAD Tools

- Logic circuits are designed using CAD tools that automatically implement synthesis techniques.
- CAD system includes tools for design entry, synthesis and optimization, simulation and physical design.

### **Design Entry**

- The starting point in the process of designing a logic circuit is the conception of what the circuit is supposed to do and the formulation of its general structure.
- The first stage of this process involves entering into the CAD system a description of the circuit being designed. This stage is called design entry.
- For design entry, we are writing source code in a hardware description language.

### **Hardware Description Languages**

- A hardware description language (HDL) is similar to a typical computer programming language except that an HDL is used to describe hardware rather than a program to be executed on a computer.
- Two HDLs are IEEE standards: Verilog HDL and VHDL.

### **Why uses Verilog**

- Supported by most companies that offer digital hardware technology.
- Verilog provides design portability. A circuit specified in Verilog can be implemented in different types of chips and with CAD tools provided by different companies, without changing the Verilog specification.
- Both small and large logic circuit designs can be efficiently represented in Verilog code.

### **Functional Simulation**

- The functional simulator tool verifies that the designed circuit functions as expected.
- It uses two types of information.
  - First, the user's initial design is represented by the logic equations generated during synthesis.
  - Second, the user specifies the valuations of the circuit's inputs that should be applied to these equations during the simulation.
- For each valuation, the simulator evaluates the outputs produced by the expressions.
- The results of simulations are usually provided in the form of a timing diagram that the user can examine to verify that the circuit operates as required.

### **Timing Simulation**

- When the values of inputs to the circuit change it takes a certain amount of time before a corresponding change occurs at the output. This is called a propagation delay of the circuit.

## **IV. Representation of Digital Circuits in Verilog**

- **Structural representation-** A larger circuit is defined by writing code that connects simple circuit elements together.
- **Behavioral representation-** Describing a circuit by using logical expressions and programming constructs that define the behavior of the circuit but not its actual structure in terms of gates.

## Structural Specification of Logic Circuits

- A gate is represented by indicating its functional name, output, and inputs. Different logic gates are shown in Table 1.1

For example,

- A two-input AND gate, with inputs x1 and x2 and output y, is denoted as **and** (y, x1, x2);
- A four-input OR gate is specified as **or** (y, x1, x2, x3, x4);
- The NOT gate is given by **not** (y, x); implements  $y = x'$ .

Table 1.1. Various Logic gates

Name	Description	Usage
and	$f = (a \cdot b \cdot \dots)$	<b>and</b> (f, a, b, ...)
nand	$f = \overline{(a \cdot b \cdot \dots)}$	<b>nand</b> (f, a, b, ...)
or	$f = (a + b + \dots)$	<b>or</b> (f, a, b, ...)
nor	$f = \overline{(a + b + \dots)}$	<b>nor</b> (f, a, b, ...)
xor	$f = (a \oplus b \oplus \dots)$	<b>xor</b> (f, a, b, ...)
xnor	$f = (a \odot b \odot \dots)$	<b>xnor</b> (f, a, b, ...)
not	$f = \overline{a}$	<b>not</b> (f, a)

## Verilog Module

- It is a circuit or subcircuit described with Verilog code.
- The module has a name, **module\_name**, which can be any valid identifier, followed by a list of ports.
- The term port refers to an input or output connection in an electrical circuit. The ports can be of type **input**, **output**, or **inout** (bidirectional), and can be either scalar or vector.

## The General Form of a Module

```
module module name [(port name{, port name})];  
    [parameter declarations]  
    [input declarations]  
    [output declarations]  
    [inout declarations]  
    [wire or tri declarations]  
    [reg or integer declarations]  
    [function or task declarations]  
    [assign continuous assignments]  
    [initial block]  
    [always blocks]  
    [gate instantiations]  
    [module instantiations]  
endmodule
```

## Documentation in Verilog Code

- Documentation can be included in Verilog code by writing a comment. A short comment begins with the double slash, //, and continues to the end of the line. A long comment can span multiple lines and is contained inside the delimiters /\* and \*/.

## White Space

- White space characters, such as SPACE and TAB, and blank lines are ignored by the Verilog compiler.
- Multiple statements can be written on a single line.
- Placing each statement on a separate line and using indentation within blocks of code, such as an **if-else** statement are good ways to increase the readability of code.

## Signals in Verilog Code

- A signal in a circuit is represented as a net or a variable with a specific type.
- A net or variable declaration has the form  
    **type** [range] signal\_name{, signal\_name};
- The signal\_name is an identifier
- The range is used to specify vectors that correspond to multi-bit signals

## Signal Values and Numbers

- Verilog supports scalar nets and variables that represent individual signals and vectors that correspond to multiple signals.
- Each individual signal can have four possible values:  
0 = logic value 0                      1 = logic value 1  
z = tri-state (high impedance)      x = unknown value
- The value of a vector variable is specified by giving a constant of the form [size] ['radix]constant where size is the number of bits in the constant, and radix is the number base. Supported radices are  
d = decimal b = binary h = hexadecimal o = octal
- Some examples of constants include  
0 the number 0                              10 the decimal number 10  
'b10 the binary number 10 = (2)<sub>10</sub>      'h10 the hex number 10 = (16)<sub>10</sub>  
4'b100 the binary number 0100 = (4)<sub>10</sub>

## Nets

Verilog defines a number of types of nets.

- A net represents a node in a circuit.
- For synthesis purposes, the only important nets are of **wire** type.
- For specifying signals that are neither inputs nor outputs of a module, which are used only for internal connections within the module, Verilog provides the **wire** type.

## Identifier Names

- Identifiers are the names of variables and other elements in Verilog code.
- The rules for specifying identifiers are simple: any letter or digit may be used, as well as the \_ underscore and \$ characters.
- An identifier must not begin with a digit and it should not be a Verilog keyword.
  - Examples of legal identifiers are f, x1, x, y, and Byte.
  - Some examples of illegal names are 1x, +y, x\*y, and 258
- Verilog is case sensitive; hence k is not the same as K, and BYTE is not the same as Byte.



## Verilog Operators

- Verilog operators are useful for synthesizing logic circuits.
- Table 1.2 lists these operators in groups that reflect the type of operation performed.

Table 1.2. Operators and their operations

Operator type	Operator Symbols	Operation Performed	Number of operands
<b>Bitwise</b>	~	1's complement	1
	&	Bitwise AND	2
		Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
<b>Logical</b>	!	NOT	1
	&&	AND	2
		OR	2
<b>Reduction</b>	&	Reduction AND	1
	~&	Reduction NAND	1
		Reduction OR	1
	~	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1
<b>Arithmetic</b>	+	Addition	2
	-	Subtraction	2
	-	2's complement	1
	*	Multiplication	2
	/	Division	2
<b>Relational</b>	>	Greater than	2
	<	Lesser than	2
	>=	Greater than or equal to	2
	<=	Lesser than or equal to	2
<b>Equality</b>	==	Logical equality	2
	!=	Logical inequality	2
<b>Shift</b>	>>	Right shift	2
	<<	Left shift	2
<b>Concatenation</b>	{,}	Concatenation	Any number
<b>Replication</b>	{{}}	Replication	Any number
<b>Conditional</b>	?:	Conditional	3

# Introduction to VIVADO tool

## Aim:

To study the simulation of Verilog code using Vivado tool.

## Introduction:

Xilinx Tool is a suite of software tools used for the implementation of digital circuits using Xilinx Field Programmable Gate Array (FPGA) or Complex Programmable Logic Device (CPLD). The design procedure consists of (a) design entry, (b) synthesis and implementation of the design, (c) functional simulation and (d) testing and verification.

Digital design can be entered in various ways - using *schematic entry* tool, using a Hardware Description Language (HDL) - Verilog or VHDL or a combination of both. Here, we will consider the Verilog HDL based design flow.

A Verilog input file typically consists of the following segments:

- **Header:** module name, list of input and output ports.
- **Declarations:** input and output ports, registers and wires.
- **Logic Descriptions:** equations, state machines and logic functions.
- **End:** endmodule

All the designs must be specified using the above given Verilog input format.

## Getting Started

To start VIVADO, double-click the desktop icon,,



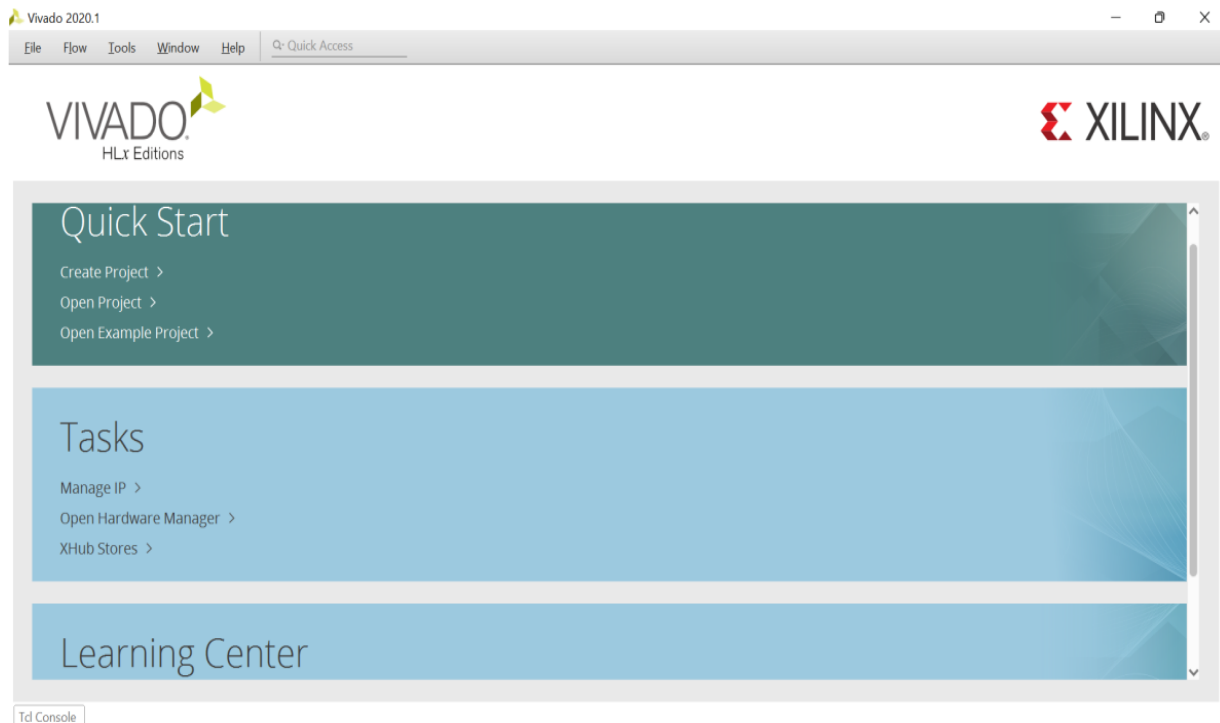
or start VIVADO from the Start menu by selecting:

**Start All Programs VIVADO 2020—• Project Navigator Note:**

Your start-up path is set during the installation process and may differ from the one above.

## Create a New Project

Double click on the Vivado icon on your desktop to open up the welcome window of the development tool (as shown below). Three main sections can be observed in this window: “Quick Start”, “Tasks”, and “Learning Center”.



### To create a new project:

1. In the opening menu, as soon as the VIVADO opens it notifies the Quick start option. **Quick Start>Create Project...** The New “**Create a New VIVADO Project**” Wizard appears.
2. Type **Next**, immediately, you see a below screen. Name your project, 'tst\_1' and it will be in the folder.

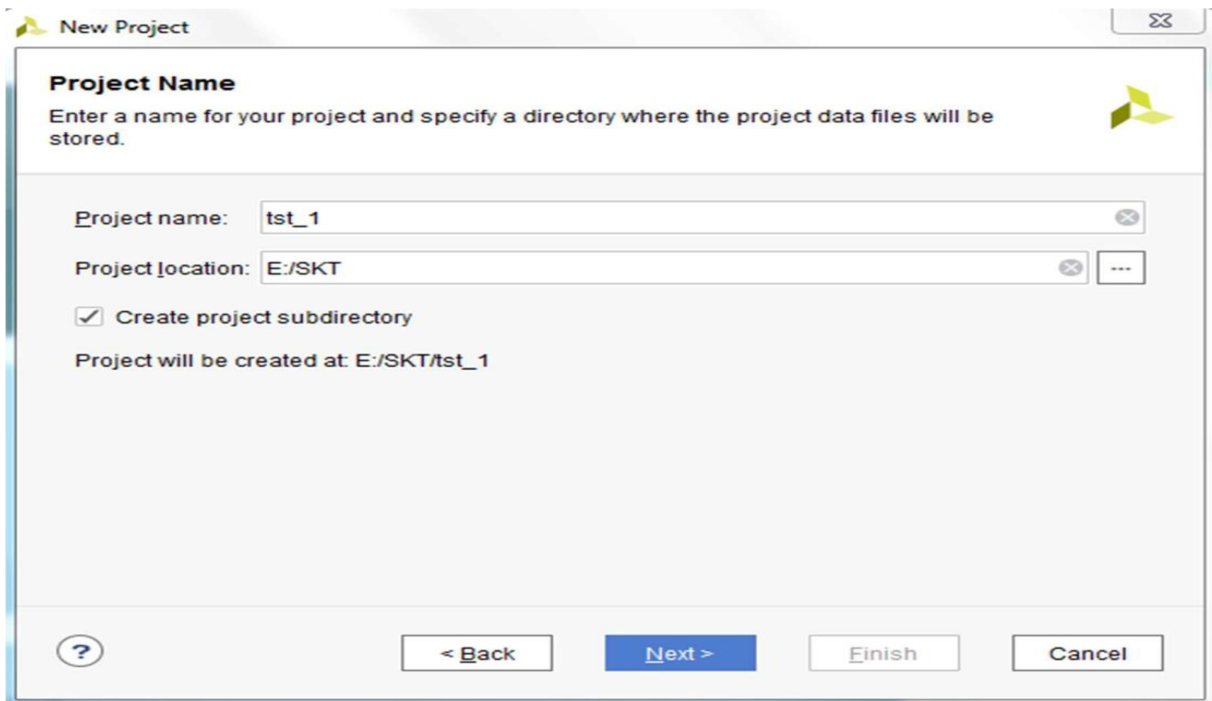


Fig.1.1: The New Project Wizard

3. In the Project Name field enter the desired Project Name. Enter or browse to a project location (directory path) for the new project, check the box project subdirectory, and press next.
4. A new window will open which shows the **project type**. **Click on RTL project**. In the next window, choose “RTL Project” as the project type and click next.

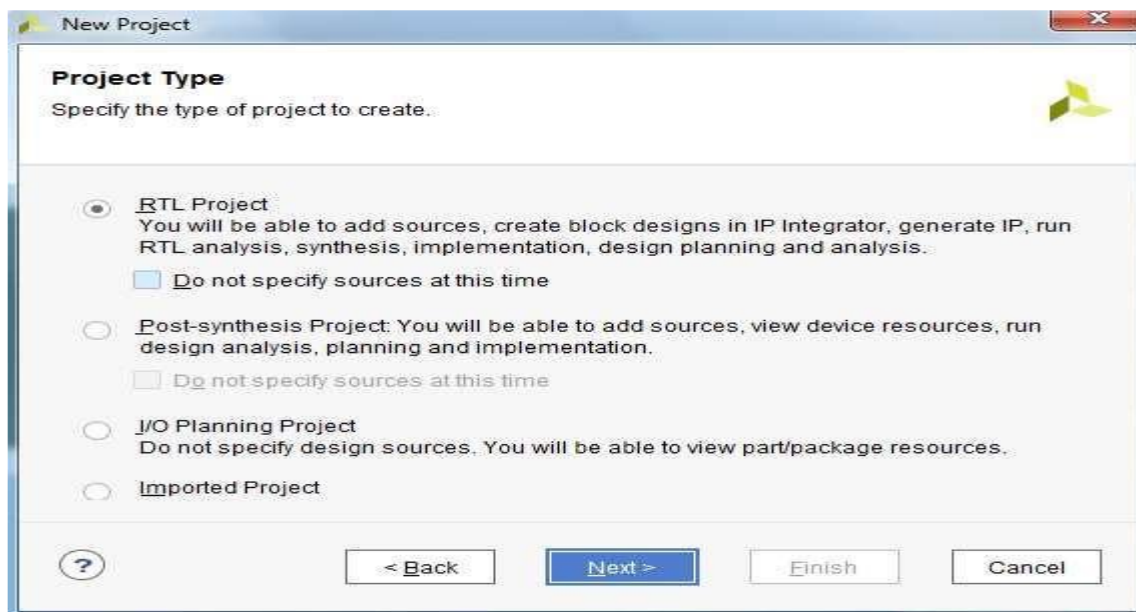


Fig.1.2: Project Type

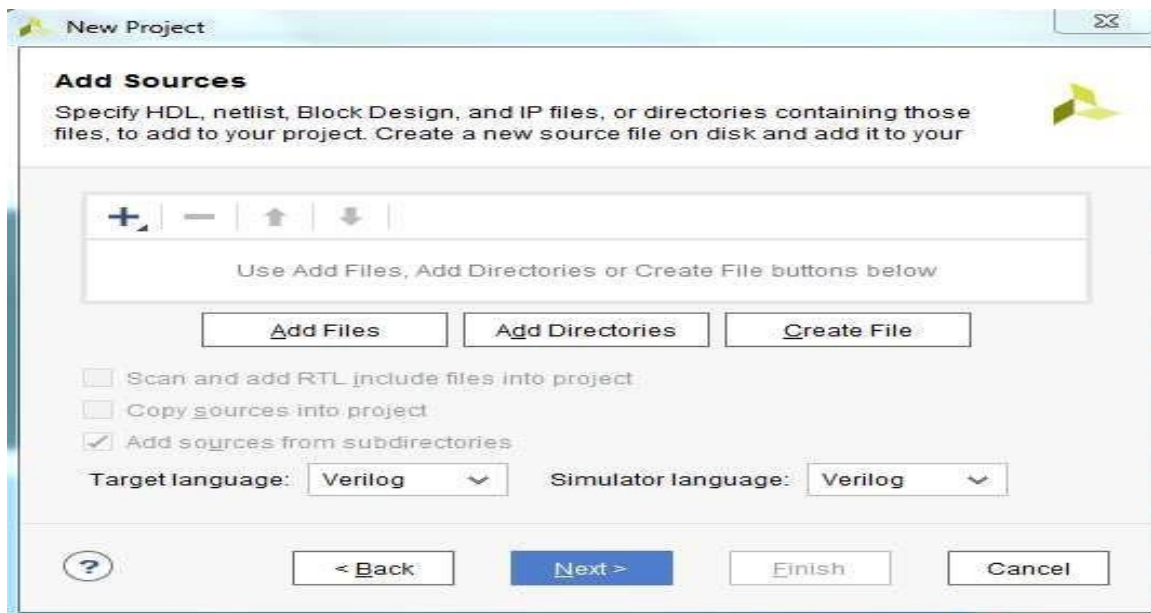


Fig.1.3: Add Sources

5. In the Add sources window click on create source to create a new Verilog file. If the Verilog file already exists, then click on add file and browse folder file.

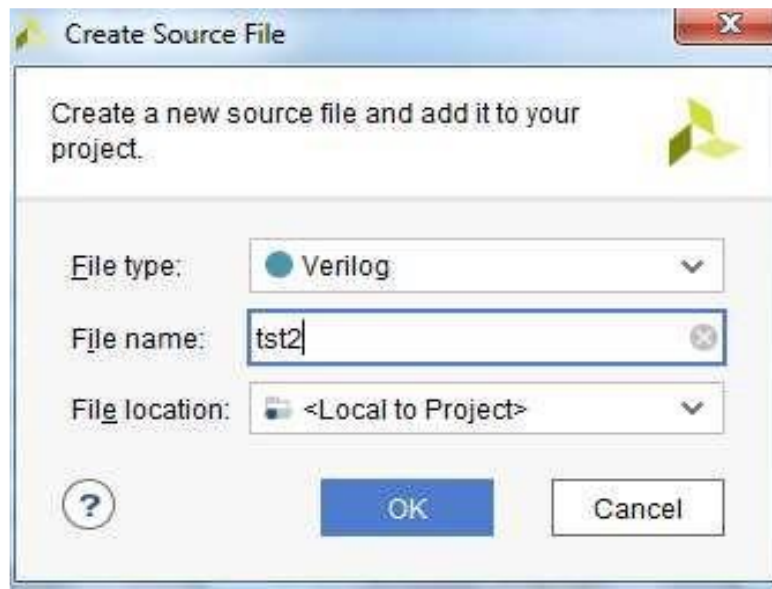


Fig.1.4(a): Create Source File

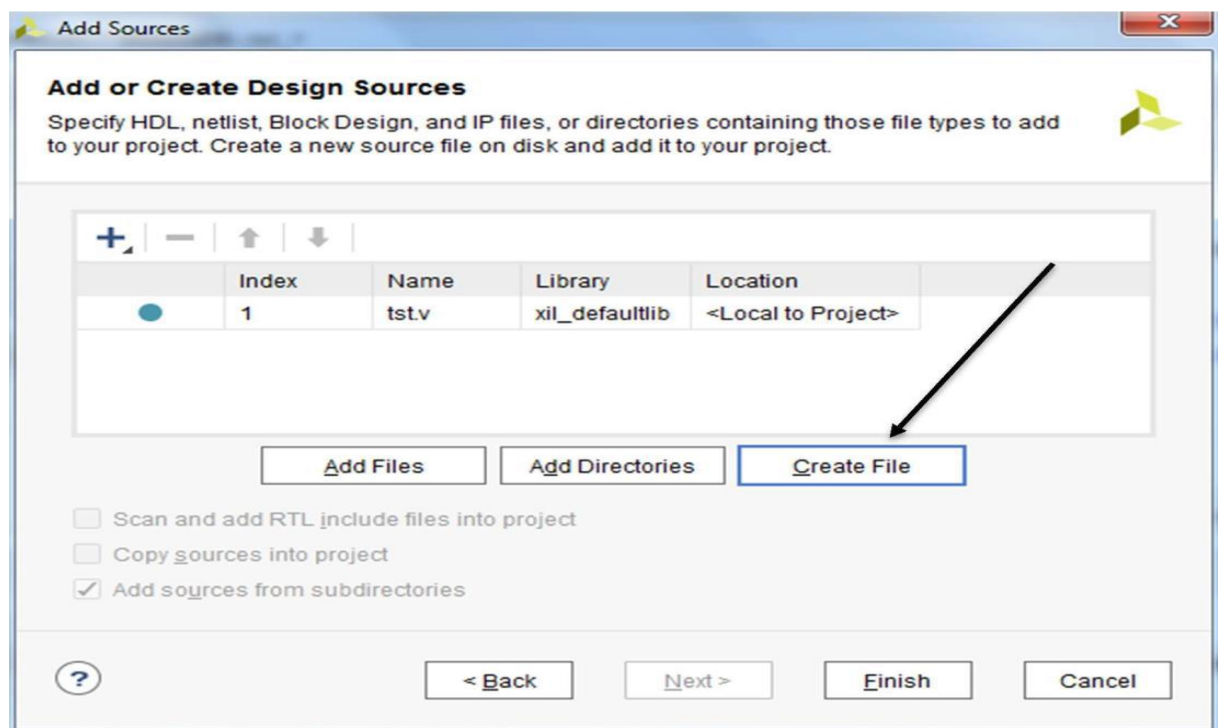


Fig.1.4(b): Create Source File

6. In the create source file window specify the hardware language as Verilog. Specify the **New file name** and click ok.

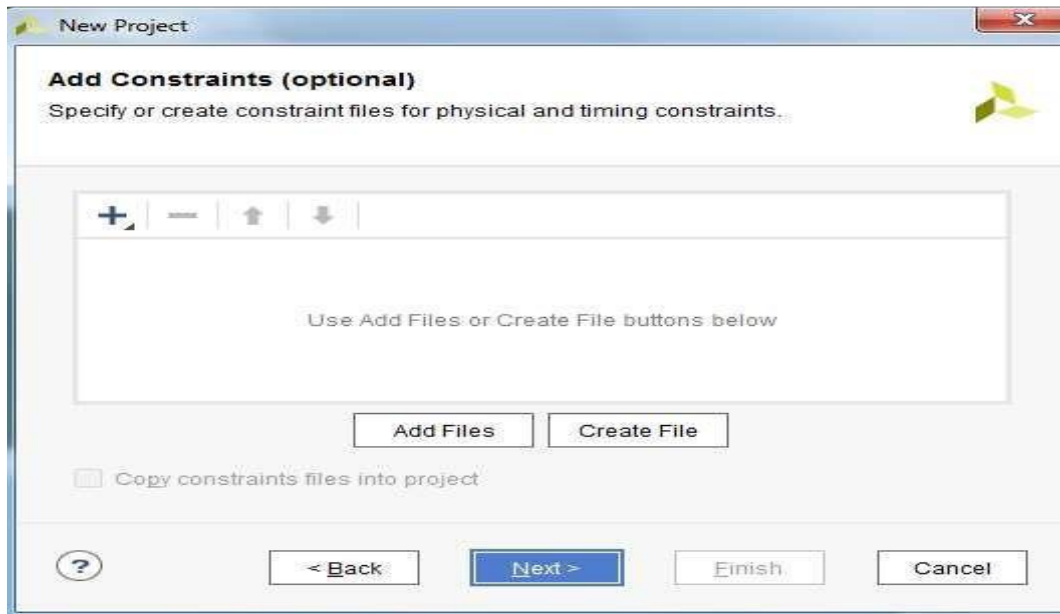


Fig.1.5: Add Constraints

This is an optional window click on next.

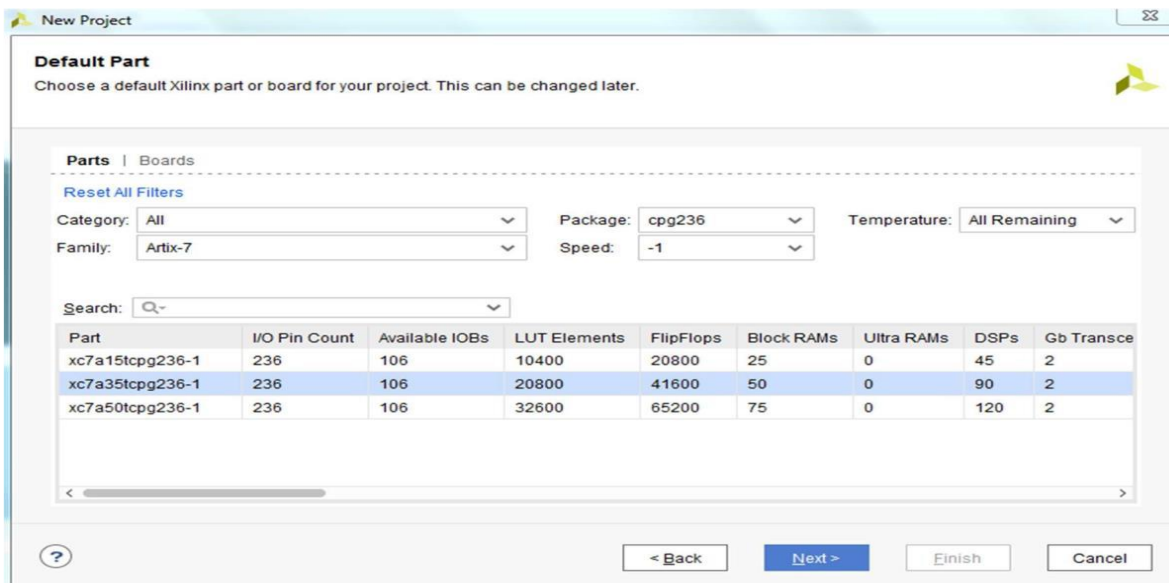


Fig.1.6: Default part

7. Fill in the properties as shown in Fig. 6: to select the board for the project.

- ◆ Product Category: **All**
- ◆ Family: **Artix -7**
- ◆ Speed: **-1**
- ◆ Package: **cpg236**

**Click Next**

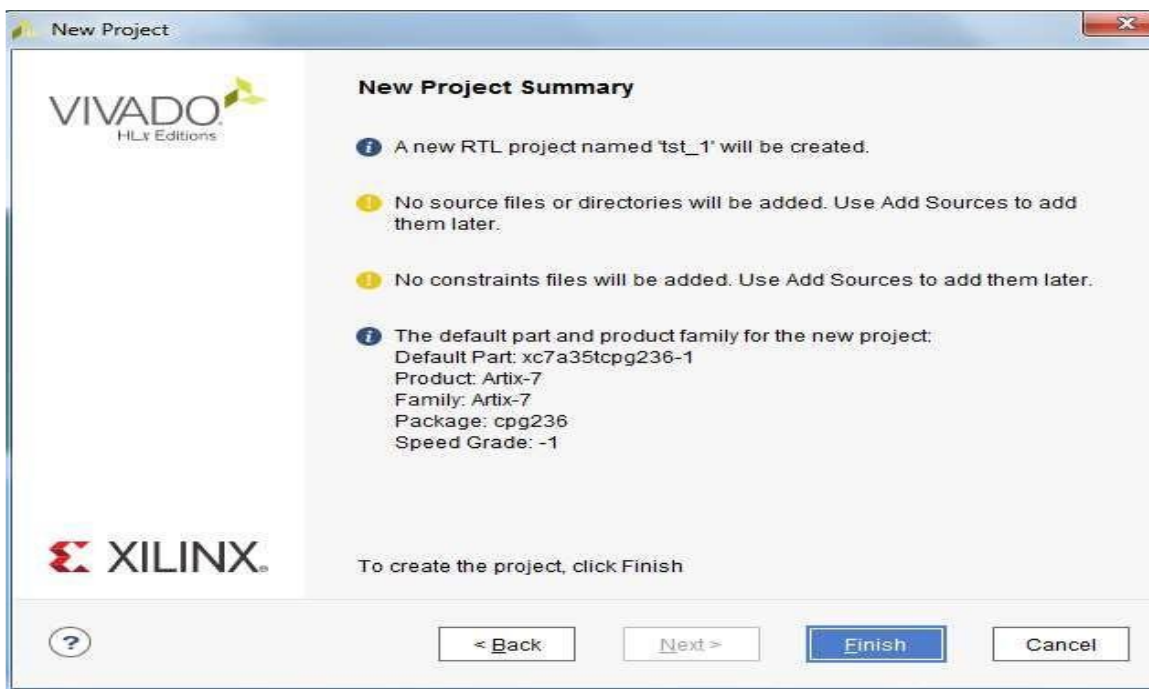


Fig.1.7: New Project Window

**Click Finish** to create the New Project.

8. New Project window will appear. Under Design sources of the Source window your file will appear in .v format. Then double click on the file to start writing your code.

If your file does not appear in the sources window, or if the Sources were not added previously then follow the below figure to add sources. After this step a new window will open. Click on Add Sources.



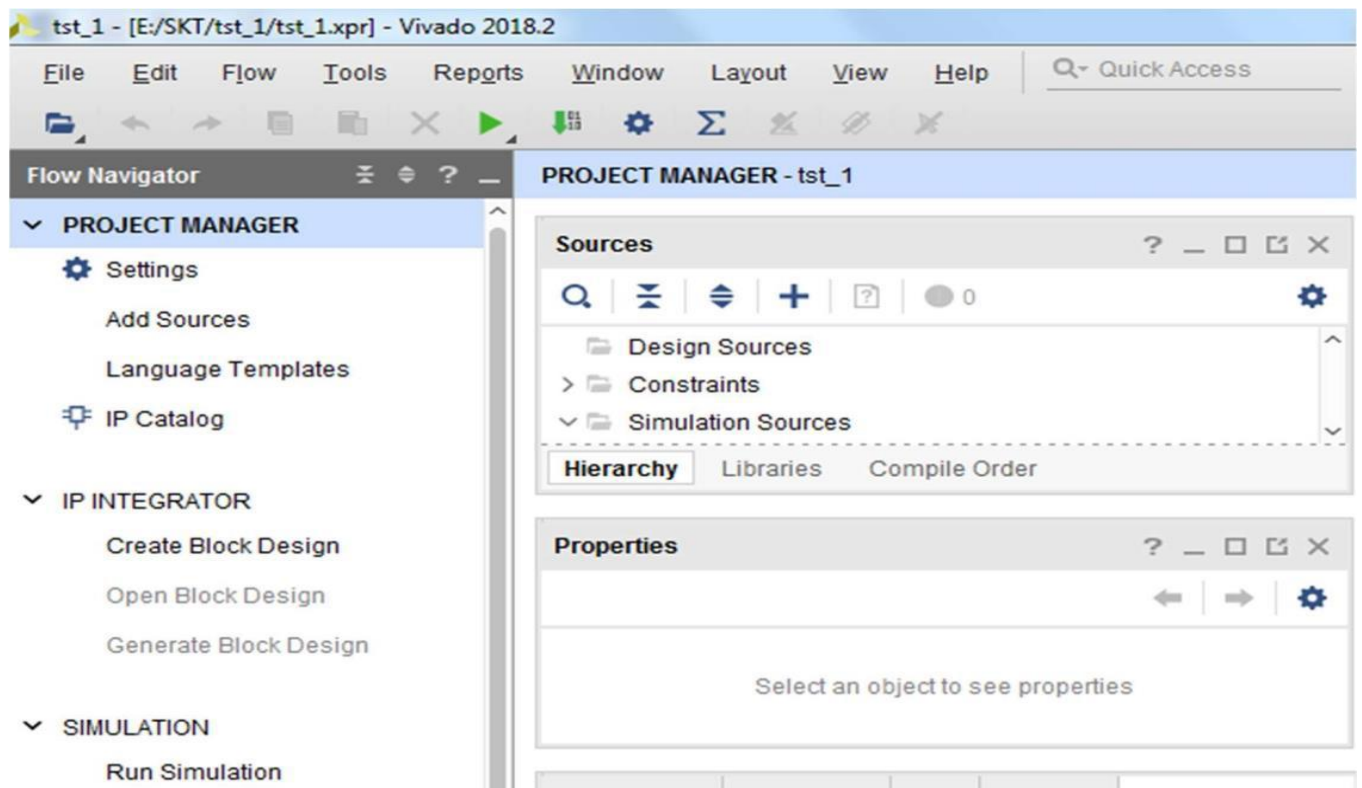


Fig.1.8: Selecting the source file.

9. Adding Sources: Click on add or create design sources and then click next.

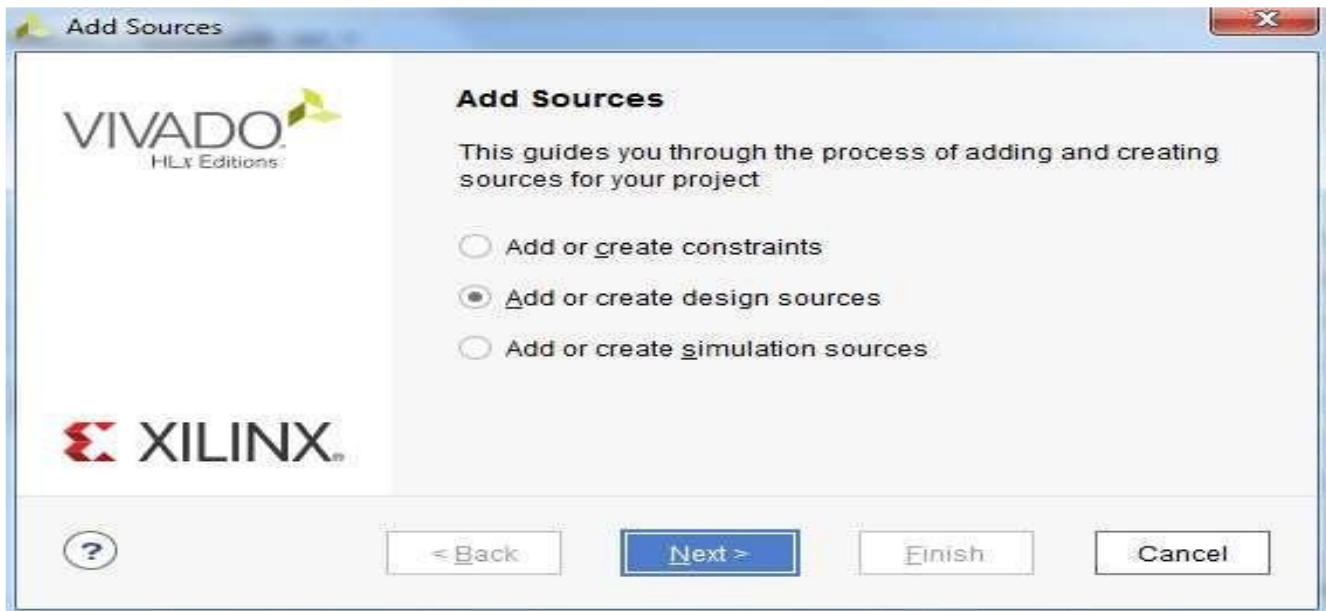
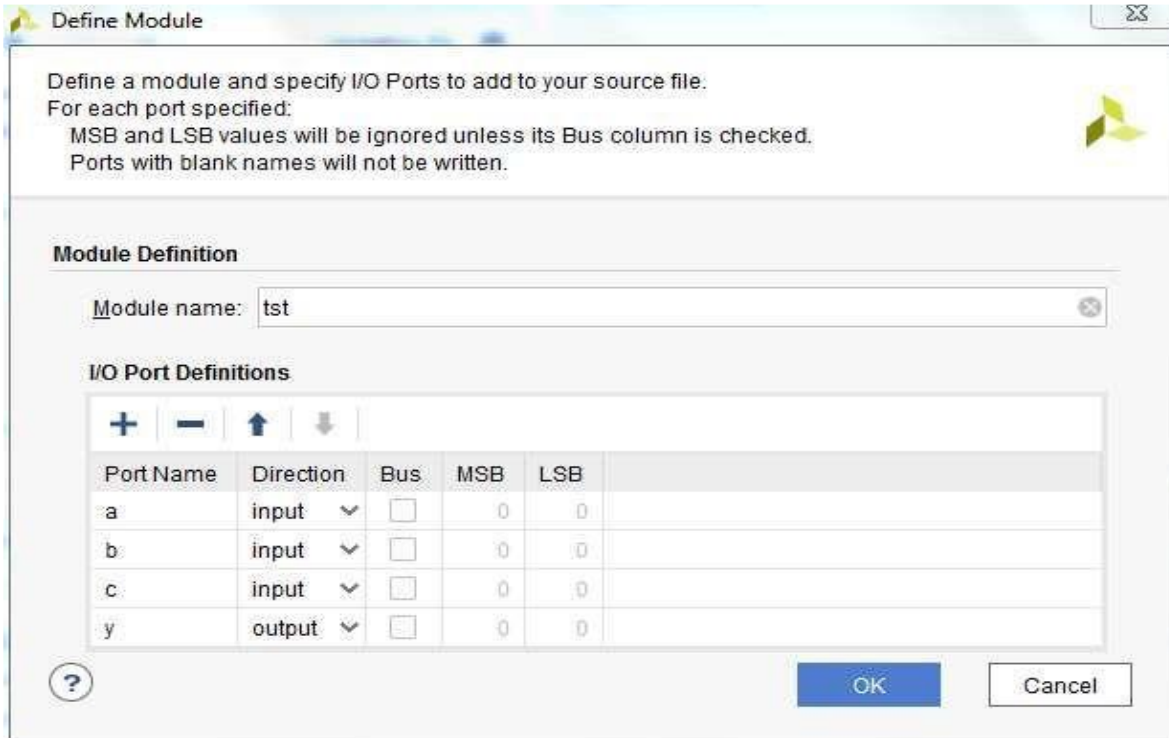


Fig.1.9: Add Sources

10. Add or Create Design source window will open and click on create file.
11. Click Finish.
12. The **Define module** requests input and outputs for the module. Specify your input and output variables and **Click OK**.



Define a module and specify I/O Ports to add to your source file.  
For each port specified:  
MSB and LSB values will be ignored unless its Bus column is checked.  
Ports with blank names will not be written.

**Module Definition**

Module name:

**I/O Port Definitions**

Port Name	Direction	Bus	MSB	LSB
a	input	<input type="checkbox"/>	0	0
b	input	<input type="checkbox"/>	0	0
c	input	<input type="checkbox"/>	0	0
y	output	<input type="checkbox"/>	0	0

Buttons: ? OK Cancel

Fig.1.10: I/O port definitions.

13. Double click the created tst2 to get an editor window, in which we write our program.

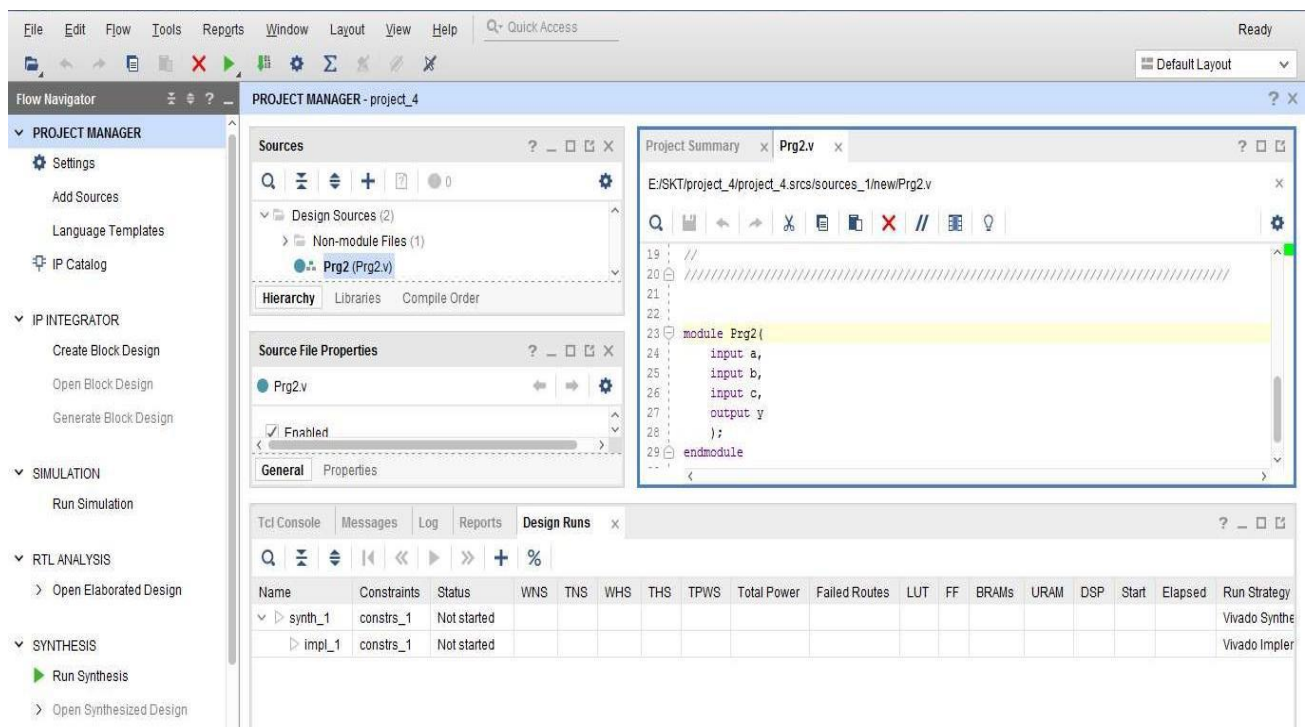


Fig.1.11(a): Writing the source code

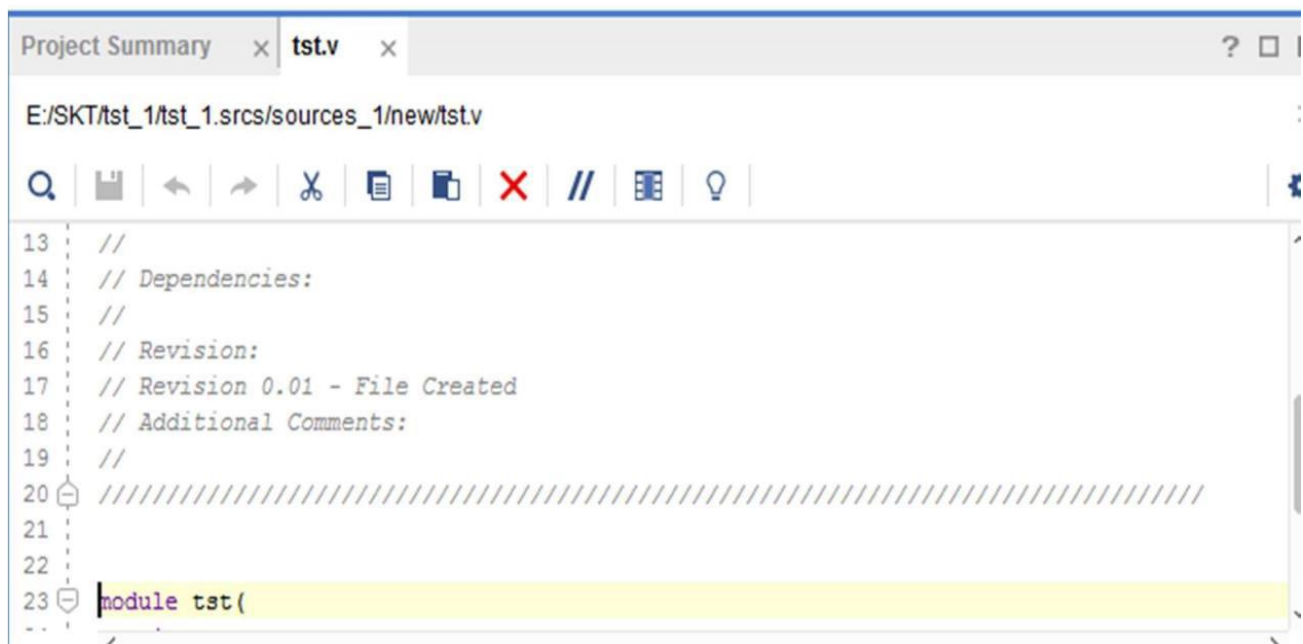


Fig.1.11(b): Writing the source code

14. An example program has been shown here for reference:

After writing the code save the file or press **Ctrl+S**. If there exists a syntax error in your code, it will appear in the source window. Check and remove the syntax errors before moving on to the next step.

After writing the code click on Run simulation icon shown on the left.

After **Run simulation** the top up window will ask **Run Behavioral Simulation**, Click on that.

Example code to get the timing diagrams has been shown below.

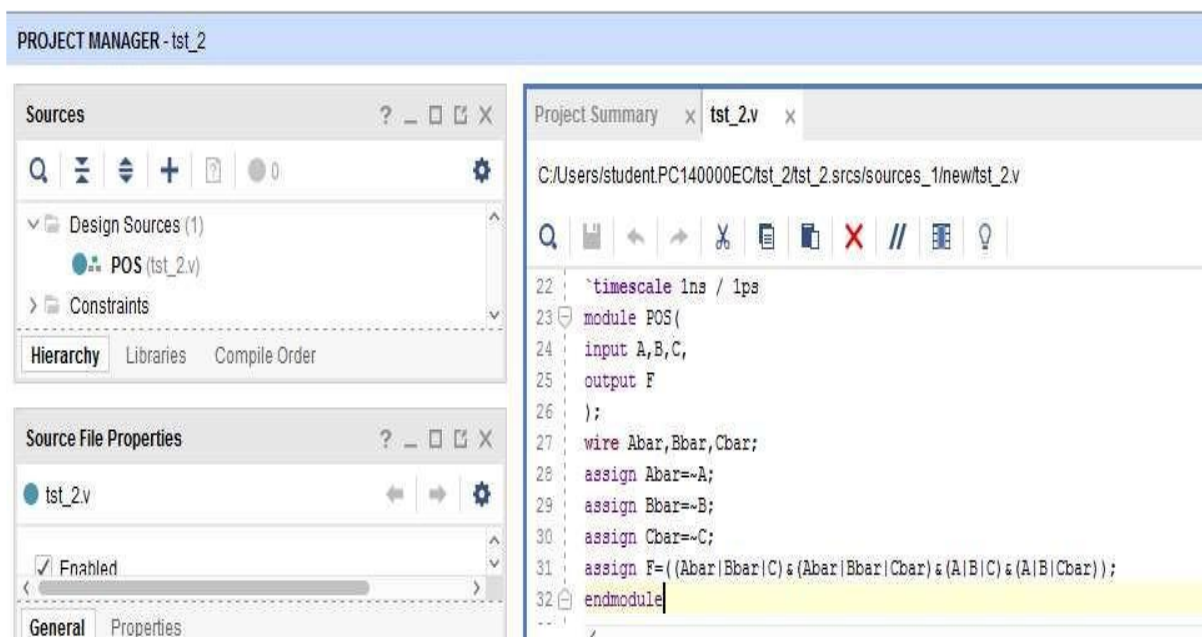


Fig.1.12: Example code

**Example-1.0:** Write a dataflow Verilog code to realize the given logic function in POS form  $y(a, b, c) = \Pi(0, 1, 6, 7)$  and verify the design by simulation.

**Solution:**

$$y(a, b, c) = (a^- + b^- + c)(a^- + b^- + \bar{c})(a + b + c)(a + b + \bar{c})$$

**Verilog Code:**

```
module tst_2( input A,B,C, output F);
```

```
wire
```

```

Abar,Bbar,Cbar;
assign Abar=~A;
assign Bbar=~B;
assign Cbar=~C;
assign F=((Abar|Bbar|C)&(Abar|Bbar|Cbar)&(A|B|C)&(A|B|Cbar));
endmodule

```

Run Simulation: Run Behavioral simulation.

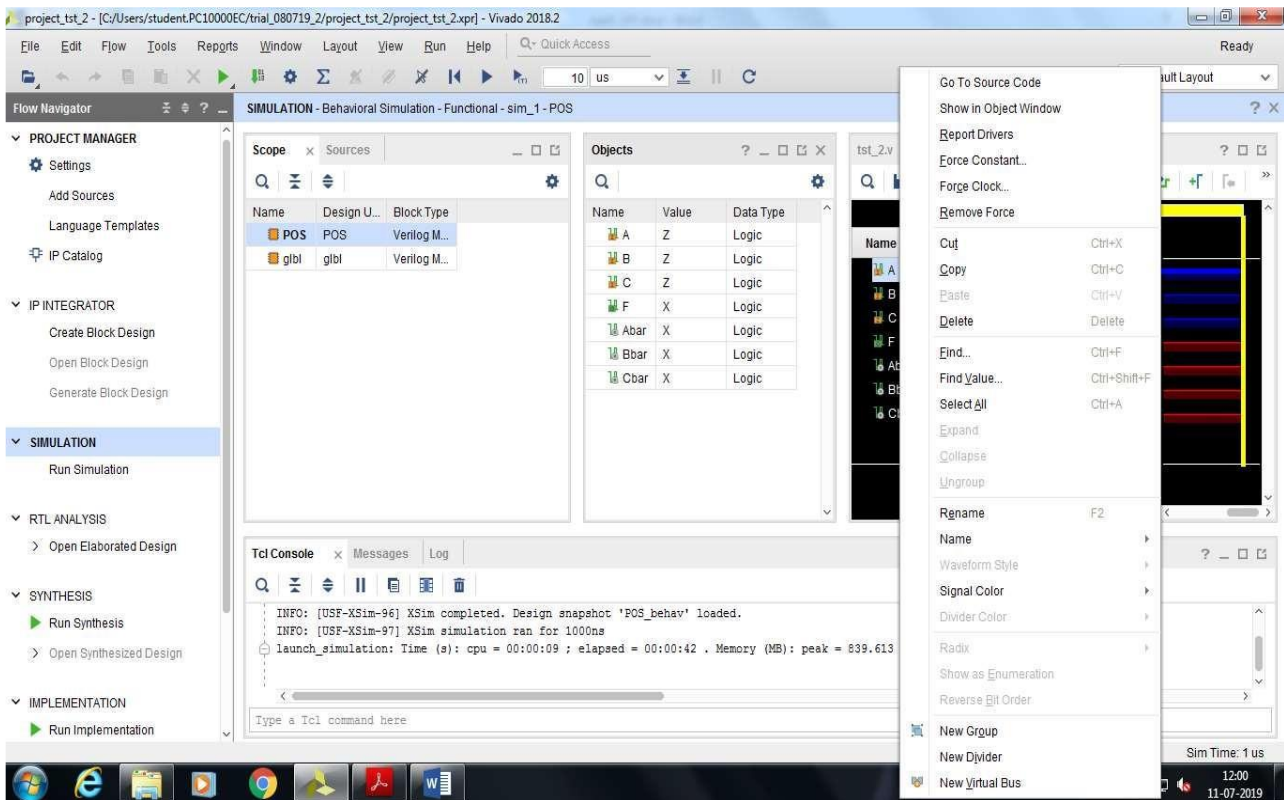


Fig.1.13: Running the simulation

Use force constant, and give the input values, A, B, C in this example. After forcing the values click the icon shown above to run the simulation.

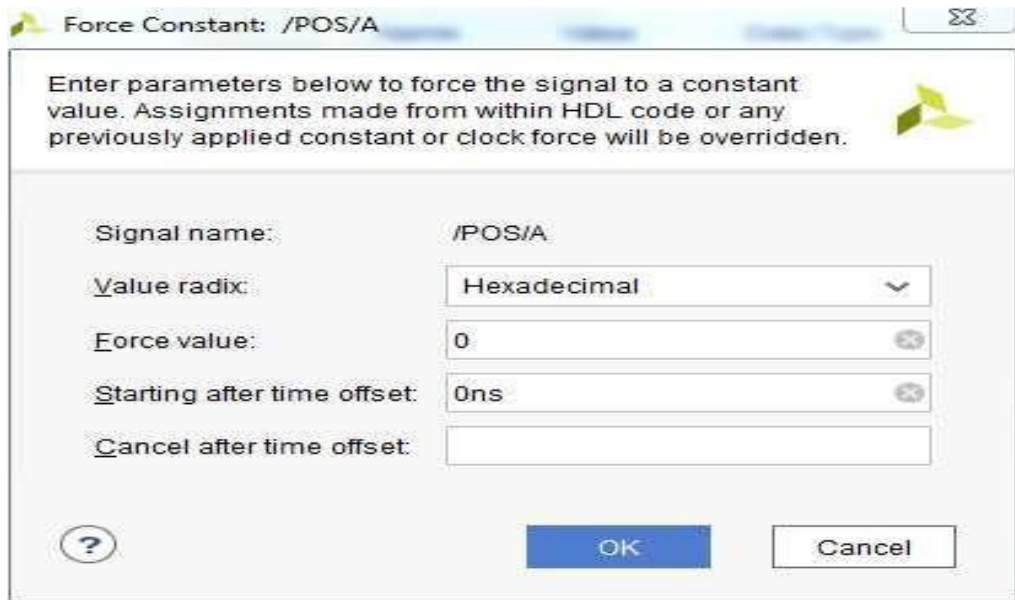


Fig.1.14: Force value as input.

In the figure shown below the input values are A=0; B=1; C=0. F is the output. Corresponding timing diagram is shown.

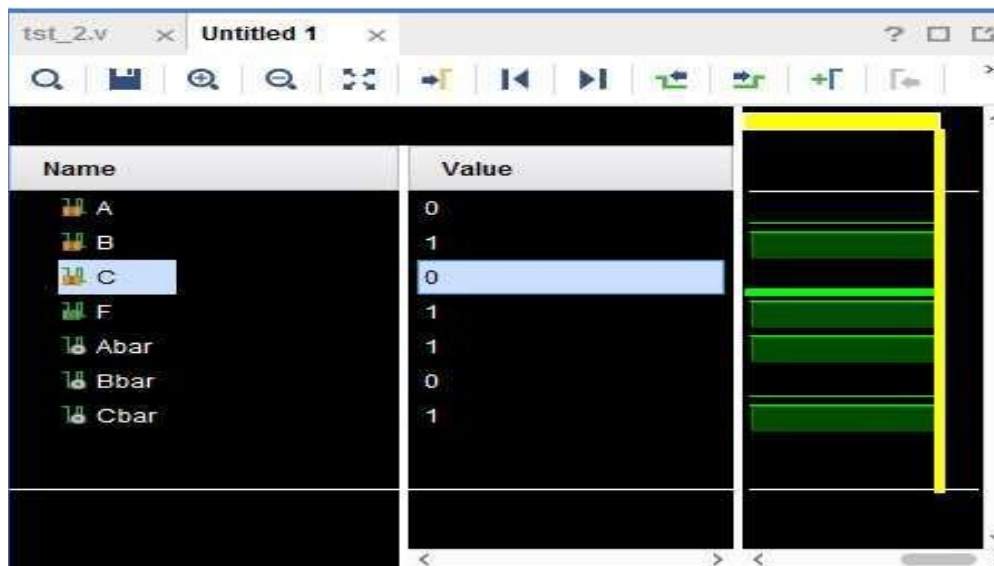


Fig.1.15: Timing diagram



## Lab No 2

### REALIZE ALL LOGIC GATES

#### Block Diagram:

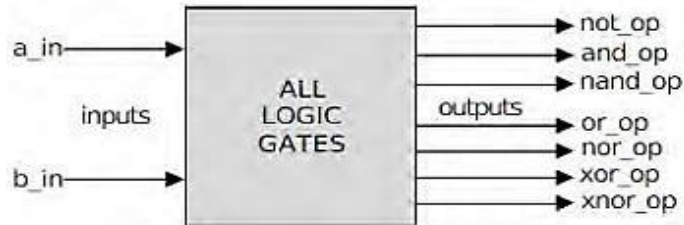


Figure 2.1

#### Logic Diagram:

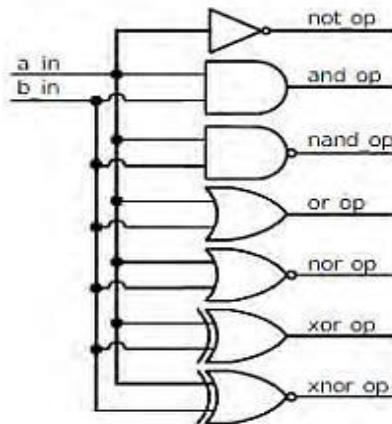


Figure 2.2

#### Truth Table:

Inputs		Outputs						
a_in	b_in	not_op (a_in)	and_op	nand_op	or_op	nor_op	xor_op	xnor_op
0	0	1	0	1	0	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	1	0
1	1	0	1	0	1	0	0	1

### **Verilog Code to Realize all logic gates**

```
module gates(a_in, b_in, not_op, and_op, nand_op, or_op, nor_op, xor_op, xnor_op);  
input a_in, b_in;  
output not_op, and_op, nand_op, or_op, nor_op, xor_op, xnor_op;  
assign not_op= ~a_in;  
assign and_op=a_in&b_in;  
assign nand_op=~(a_in&b_in);  
assign or_op=a_in|b_in;  
assign nor_op=~(a_in|b_in);  
assign xor_op=a_in^b_in;  
assign xnor_op=~(a_in^b_in);  
endmodule
```



## Lab No: 3

### SIMPLIFICATION USING K-MAP

#### I. Steps for Optimization using K-map:

##### K-map

- A systematic way of performing optimization.
- Finds a minimum-cost expression for a given logic function by reducing the number of product (or sum) terms needed in the expression, by applying the combining property.
- **Implicant-** A product term that indicates the input valuation(s) for which a given function is equal to 1.
- **Prime Implicant-** An implicant is called a prime implicant if it cannot be combined into another implicant that has fewer literals.
- **Essential prime implicant-** If a prime implicant includes a minterm for which  $f = 1$  that is not included in any other prime implicant, then it must be included in the cover and is called as an essential prime implicant.
- The process of finding a minimum-cost circuit involves the following steps:
  1. Generate all prime implicants for the given function  $f$ .
  2. Find the set of essential prime implicants.
  3. If the set of essential prime implicants covers all valuations for which  $f = 1$ , then this set is the desired cover of  $f$ . Otherwise, determine the nonessential prime implicants that should be added to form a complete minimum-cost cover.

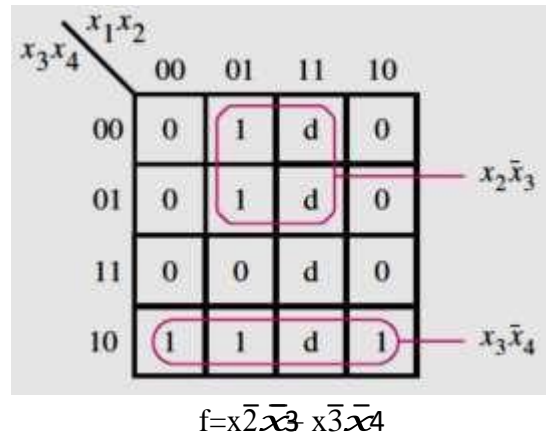
#### II. Incompletely Specified Functions

- A function that has don't-care condition(s).
- Using the shorthand notation, the function  $f$  is specified as
$$f(x_1, \dots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$
where  $D$  is the set of don't cares.

##### Solved Exercise:

Simplify the following function using K-map and write Verilog code to implement this.

$$f(x_1, \dots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$



**Verilog code:**

```

module example4(x2,x3,x4,f);
    input x2,x3,x4;
    output f;
    assign f=(x2 & ~x3) | (x3 & ~x4);
endmodule

```

### Lab Exercises

1. Simplify the following functions using K-map and implement the circuit using logic gates.
  - a)  $f(A,B,C,D) = \sum m(2,3,4,5,6,7,10,11,12,15)$
  - b)  $f(A,B,C,D) = \sum m(1,3,4,9,10,12) + D(0,2,5,11)$
2. Simplify the following functions using K-map and implement the circuit using logic gates.
  - a)  $f(A,B,C,D) = \prod M(0,1,4,6,8,9,12,14)$
  - b)  $f(A,B,C,D) = \prod M(6,9,10,11,12) + D(2,4,7,13)$

## Lab No: 4

### MULTILEVEL SYNTHESIS

- **Multilevel NAND and NOR Circuits**

- Multilevel AND-OR circuits can be realized by a circuit that contains only NAND gates or only NOR gates.
- Each AND gate is converted to a NAND by inverting its output.
- Each OR gate is converted to a NAND by inverting its inputs.
- Each AND gate is converted to a NOR by inverting its inputs.
- Each OR gate is converted to a NOR by inverting its output.
- Inversions that are not a part of any gate can be implemented as two-input NAND/NOR gates, where the inputs are tied together.

- **Functional decomposition-** Complex logic circuit can be reduced by decomposing a two-level circuit into subcircuits, where one or more subcircuits implement functions that may be used in several places to construct the final circuit.

#### Solved Exercise

Apply functional decomposition for the following function to obtain a simplified circuit and simulate using Verilog.

$$f = \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 x_4 + \bar{x}_1 \bar{x}_2 x_4$$

Factoring  $x_3$  from the first two terms and  $x_4$  from the last two terms, this expression becomes

$$f = (\bar{x}_1 x_2 + x_1 \bar{x}_2) x_3 + (x_1 x_2 + \bar{x}_1 \bar{x}_2) x_4$$

Now let  $g(x_1, x_2) = \bar{x}_1 x_2 + x_1 \bar{x}_2$  and observe that

$$\begin{aligned}\bar{g} &= \overline{\bar{x}_1 x_2 + x_1 \bar{x}_2} \\ &= \overline{\bar{x}_1 x_2} \cdot \overline{x_1 \bar{x}_2} \\ &= (x_1 + \bar{x}_2)(\bar{x}_1 + x_2) \\ &= x_1 \bar{x}_1 + x_1 x_2 + \bar{x}_2 \bar{x}_1 + \bar{x}_2 x_2 \\ &= 0 + x_1 x_2 + \bar{x}_1 \bar{x}_2 + 0 \\ &= x_1 x_2 + \bar{x}_1 \bar{x}_2\end{aligned}$$

Then  $f$  can be written as

$$f = g x_3 + \bar{g} x_4$$

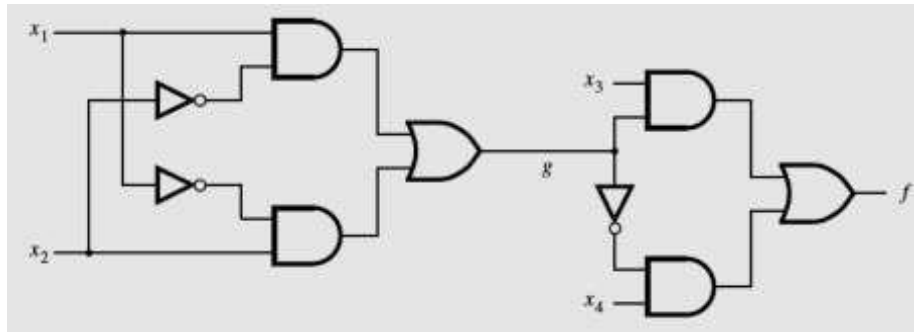


Figure 4.1

**Verilog code:**

```

module example5(x1,x2,x3,x4,f);
    input x1,x2,x3,x4;
    output f;
    assign g=(x1 & ~x2) | (~x1 & x2);
    assign f=(g & x3) | (~g & x4);
endmodule

```

### Lab Exercises

1. Minimize the following expression using K-map and simulate using only NAND gates.  
 $f(A,B,C,D) = \pi M(2,6,8,9,10,11,14)$
2. Minimize the following expressions using K-map and simulate using only NOR gates.  
 $f(A,B,C,D) = \sum m(0,1,2,5,8,9,10)$
3. Minimize the following expressions using K-map and simulate using NOR gates only.  
 $f(A,B,C,D) = \sum m(1,3,5,7,9) + D(6,12,13)$

## Lab No: 5

### ARITHMETIC CIRCUITS

#### I. Adder circuit:

- **Half adder-** a circuit that implements the addition of only two single bit inputs.
- **Full adder-** a circuit that implements the addition of two single bit inputs and one carry bit.
- **Ripple-carry adder**
  - For each bit position, we can use a full-adder circuit, connected as shown in Fig. 4.1.
  - Carries that are produced by the full-adders propagate to the left.

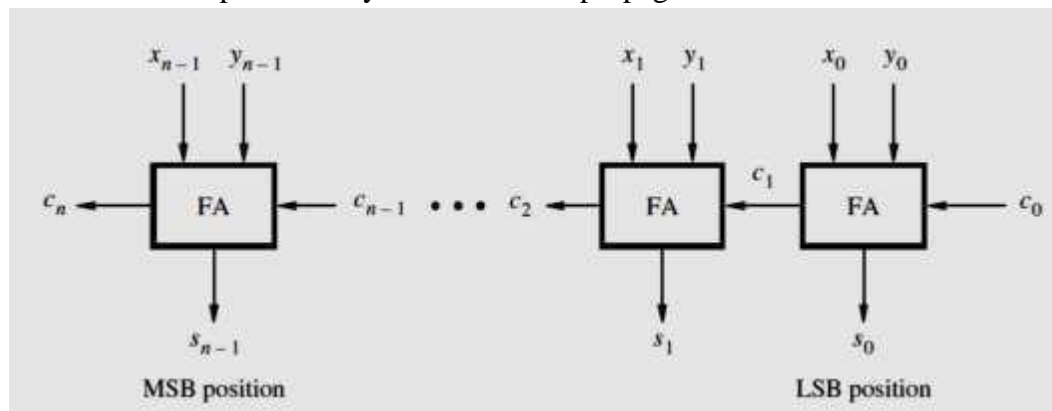


Figure 5.1 An n-bit ripple carry adder

- **Adder/Subtractor unit-**
  - The only difference between performing addition and subtraction is that for subtraction it is necessary to use the 2's complement of one operand.
  - Add/Sub control signal chooses whether addition or subtraction is to be performed.
  - Outputs of the XOR gates represent  $Y$  if Add/Sub = 0, and they represent the 1's complement of  $Y$  if Add/Sub = 1.
  - Add/Sub is also connected to the carry-in  $c_0$ . This makes  $c_0 = 1$  when subtraction is to be performed, thus adding the 1 that is needed to form the 2's complement of  $Y$ .
  - When the addition operation is performed, we will have  $c_0 = 0$ .
  - The circuit is shown in Fig. 5.1

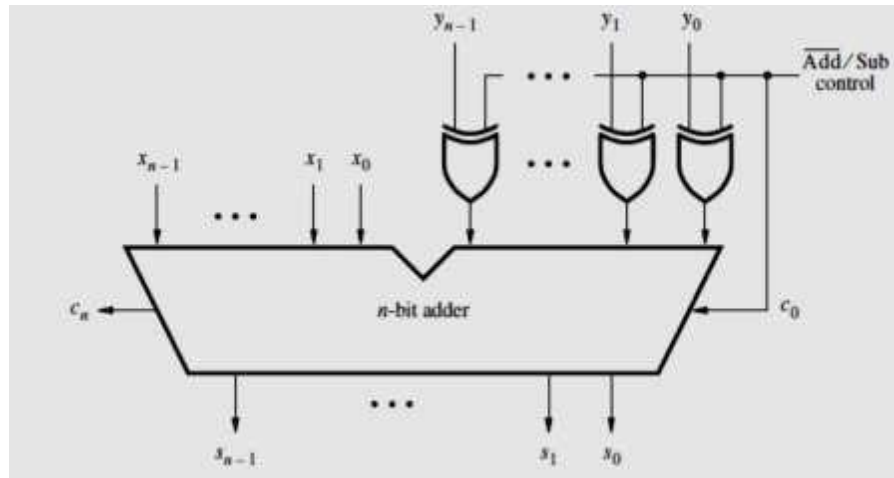


Figure 5.2 Adder/subtractor unit

- **Binary multiplier**

- Multiplication of binary numbers is performed in the same way as in decimal numbers. The multiplicand is multiplied by each bit of the multiplier starting from the least significant bit.
- Each such multiplication forms a partial product. Successive partial products are shifted one position to the left. The final product is obtained from the sum of the partial products.

- **BCD Addition**

In Binary Coded Decimal (BCD) representation each digit of a decimal number is represented by 4-bit binary. When 2 BCD numbers are added,

- If  $X + Y \leq 9$ , then the addition is the same as the addition of 2 four-bit unsigned binary numbers.
- A correct decimal digit can be generated by adding 6 to the result of a four-bit addition whenever the result exceeds 9 or when the carry is generated.

## II. Designing subcircuits in Verilog

- A Verilog module can be included as a subcircuit in another module.
- Both modules must be defined in the same file.
- The general form of a module instantiation statement is given below.

```
module_name [#(parameter overrides)] instance_name (
    .port_name ( [expression] ) { , .port_name ( [expression] ) } );
```

- The **instance\_name** can be any legal Verilog identifier and the port connections specify how the module is connected to the rest of the circuit.
- The same module can be instantiated multiple times in a given design provided that each instance name is unique.

- The `#(parameter overrides)` can be used to set the values of parameters defined inside the *module\_name* module.
- Each *port\_name* is the name of a port in the subcircuit, and each expression specifies a connection to that port.
- **Named port connections** -The syntax *.port\_name* is provided so that the order of signals listed in the instantiation statement does not have to be the same as the order of the ports given in the **module** statement of the subcircuit.
- **Ordered port connections**-If the port connections are given in the same order as in the subcircuit, then *.port\_name* is not needed.

### Using Vektored Signals

- Multibit signals are called *vectors*.
- An example of an input vector is  

```
input [3:0] W;
```
- This statement defines *W* to be a four-bit vector. Its individual bits can be referred to using an index value in square brackets.
- The most-significant bit (MSB) is referred to as *W[3]* and the least-significant bit (LSB) is *W[0]*.

### Solved Exercise

Write the Verilog code to implement a 4-bit adder.

#### Verilog code:

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;
    fulladd stage0 (carryin, x0, y0, s0, c1);
    fulladd stage1 (c1, x1, y1, s1, c2);
    fulladd stage2 (c2, x2, y2, s2, c3);
    fulladd stage3 (c3, x3, y3, s3, carryout);
endmodule

module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;
    assign s = x ^ y ^ Cin;
    assign Cout = (x & y) | (x & Cin) | (y & Cin);
endmodule
```

- Separate Verilog module for the ripple carry adder instantiate the fulladd module as a subcircuit.

### **Lab Exercises**

Write Verilog code to implement the following and simulate

1. Half adder, full adder
2. Decomposed full adder.
3. Four-bit adder using full adders.
4. Four-bit adder/ subtractor using a four-bit adder.
5. BCD adder using a four-bit adder(s).



## Lab No: 6

### MULTIPLEXERS

#### I. Multiplexers

- The multiplexer has a number of data inputs, one or more select inputs, and one output.
- It passes the signal value on one of the data inputs to the output.
- A multiplexer that has  $N$  data inputs,  $w_0, \dots, w_{N-1}$ , requires  $\log_2 N$  select inputs.
- Fig. 6.1a shows the graphical symbol for a 2-to-1 multiplexer.
- The functionality of a multiplexer can be described in the form of a truth table. Fig. 6.1b shows the functionality of a 2-to-1 multiplexer.

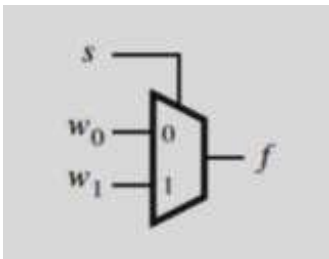


Figure 6.1a Graphical symbol

$s$	$f$
0	$w_0$
1	$w_1$

Figure 6.1b Truth table

#### The Conditional Operator

- In a logic circuit, it is often necessary to choose between several possible signals or values based on the state of some condition.
- Verilog provides a conditional operator ( $?:$ ) which assigns one of the two values depending on a conditional expression.

##### Syntax of conditional operator

```
conditional_expression ? true_expression : false_expression
```

- If the conditional expression evaluates to 1 (true), then the value of true\_expression is chosen; otherwise, the value of false\_expression is chosen.

#### The case Statement

- The general form of a **case** statement is given below.

```

case (expression)
    alternative1: begin
        statement;
    end
    alternative2: begin
        statement;
    end
    [default: begin
        statement;
    end]
endcase

```

- The bits in expression, called as controlling expression, are checked for a match with each alternative.
- The first successful match causes the associated statements to be evaluated.
- Each digit in each alternative is compared for an exact match of the four values 0, 1, x, and z.
- A special case is the **default** clause, which takes effect if no other alternative matches.
- The **casex** statement reads all z and x values as don't cares.

## Functions and Tasks

- The purpose of a **function** is to allow the code to be written in a modular fashion without defining separate modules.
- A **function** is defined within a module, and it is called either in a continuous assignment statement or in a procedural assignment statement inside that module.
- A **function** can have more than one input, but it does not have an output, because the **function** name itself serves as the output variable.
- Function general form

```

function [range | integer] function_name;
    [input declarations]
    [parameter, reg, integer declarations]
    begin
        statement;
    end
endfunction

```

## Verilog task

- A task is declared by the keyword **task** and it comprises a block of statements that ends with the keyword **endtask**.
- The task must be included in the module that calls it.
- It may have input and output ports.
- The task ports are used only to pass values between the module and the task.

## Solved Exercise

Write behavioral Verilog code for 2 to 1 multiplexer using **always** and **conditional** operators.

**Verilog code:**

```
module mux2to1 (w0, w1, s, f);  
    input w0, w1, s;  
    output f;  
    reg f;  
    always @(w0 or w1 or s)  
        f = s ? w1 : w0;  
endmodule
```

## Lab Exercises

1. Write behavioral Verilog code for a 2 to 1 multiplexer using the **if-else** statement.
2. Write behavioral Verilog code for a 4 to 1 multiplexer using **conditional** operator.
3. Write behavioral Verilog code for an 8 to 1 multiplexer using **case** statement.
4. Write behavioral Verilog code for a 2 to 1 multiplexer using **function**.

