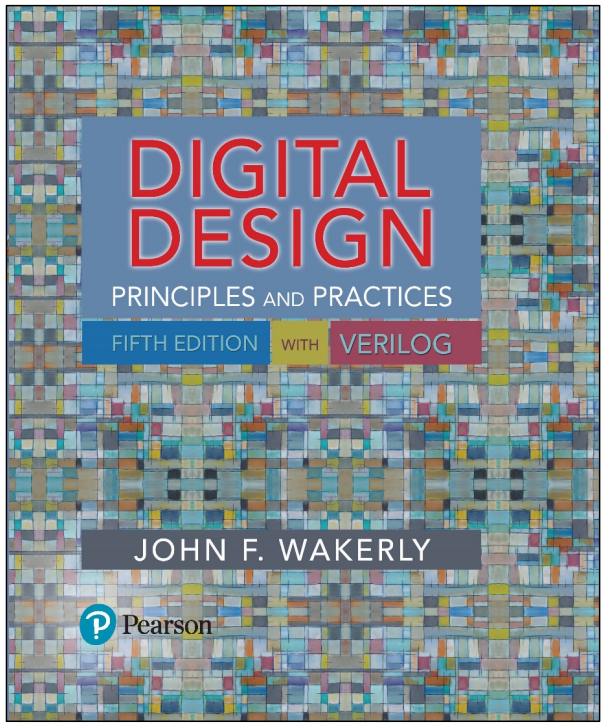
Digital Design: Principles and Practices

Chapter 5

Verilog HDL



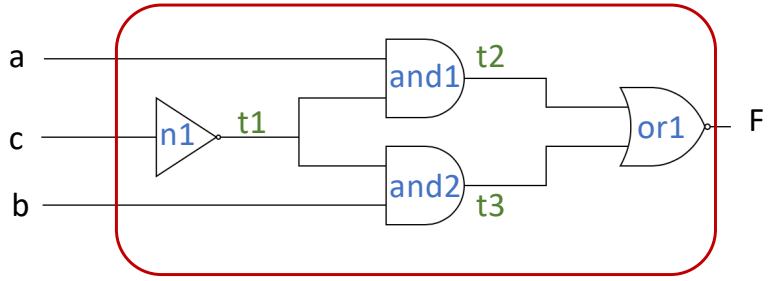
Vivado Tutorial

Creating a New Project

- Before writing your code draw your project using logic diagram or black boxes.
- To become familiar with Verilog coding in Vivado, we will implement a simple logic function:

$$F(a, b, c) = a.c' + b.c'$$

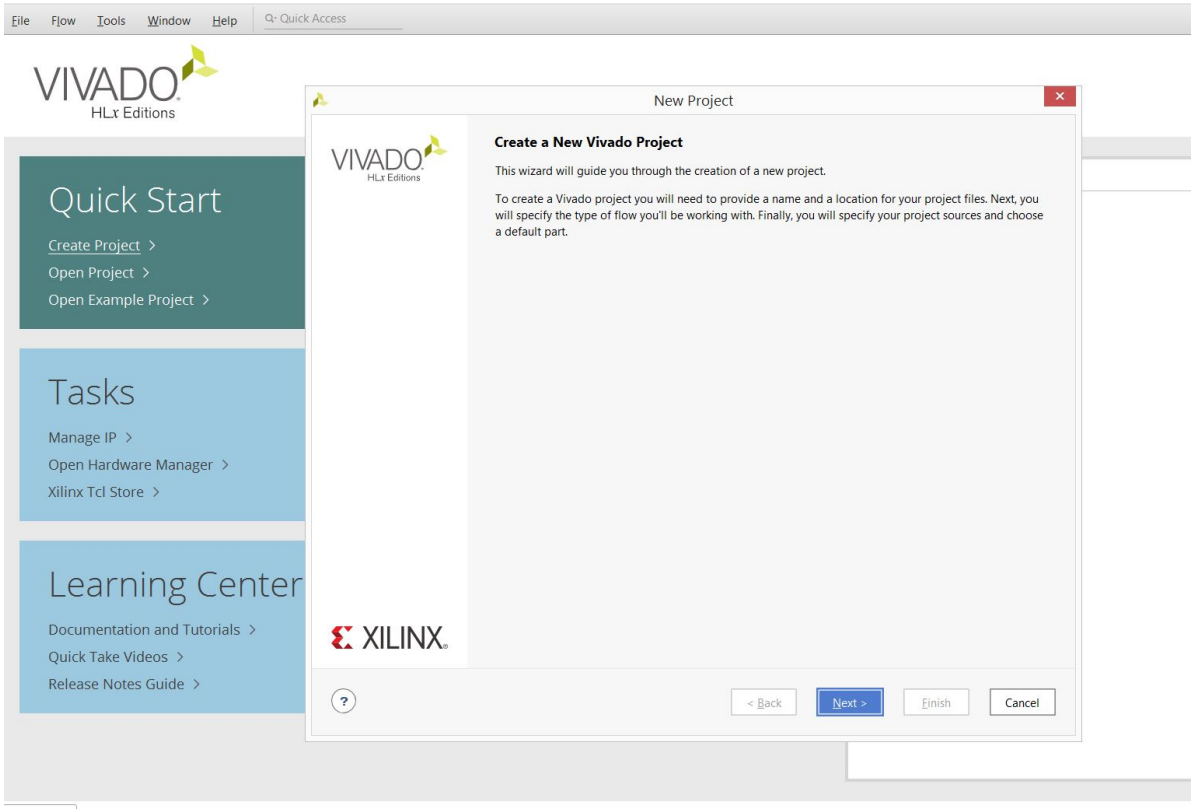
- Inputs: one bit a,b,c
- Output: one bit F
- Can be implemented as figure shows.



Vivado Tutorial

Creating a New Project

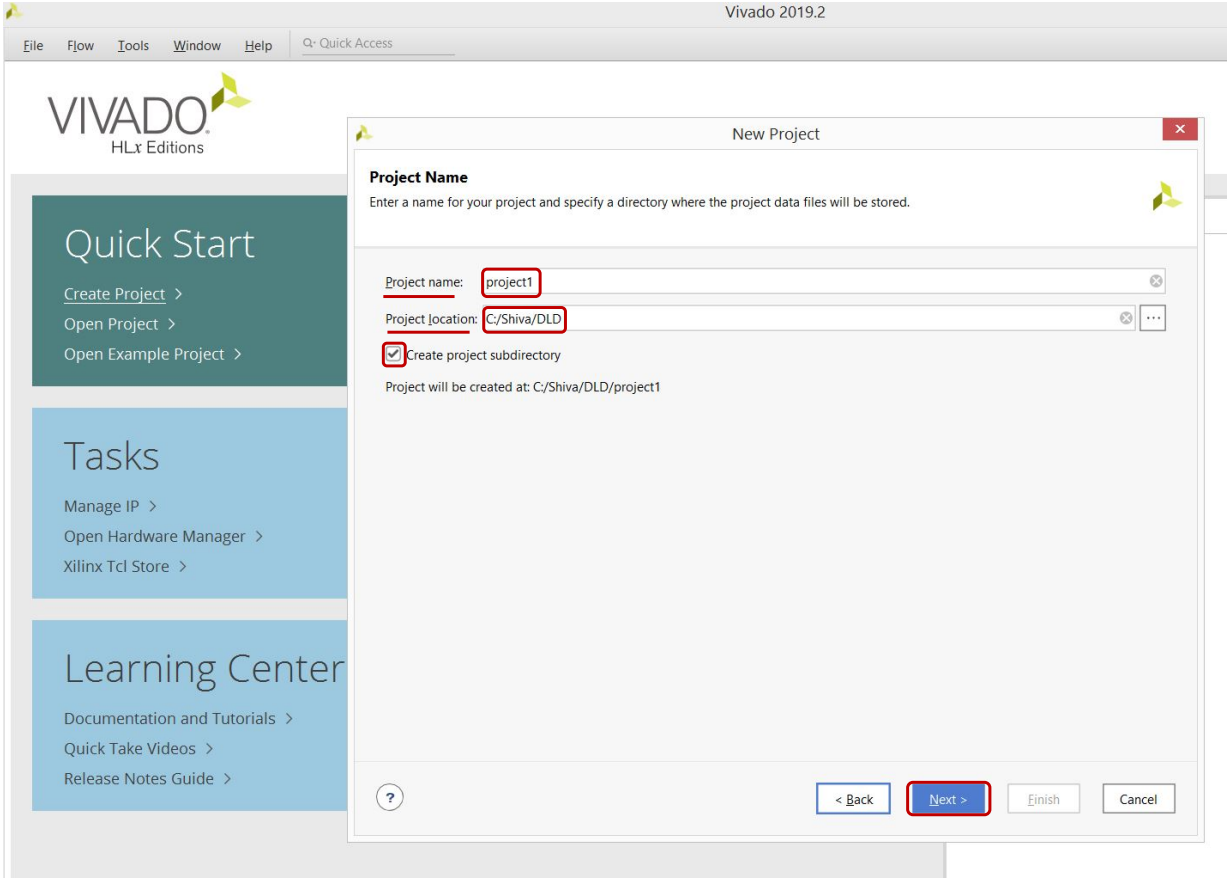
- After launching Vivado,
- click the "**Create New Project**" icon. Alternatively, you can select **File -> New Project**.



Vivado Tutorial

Creating a New Project

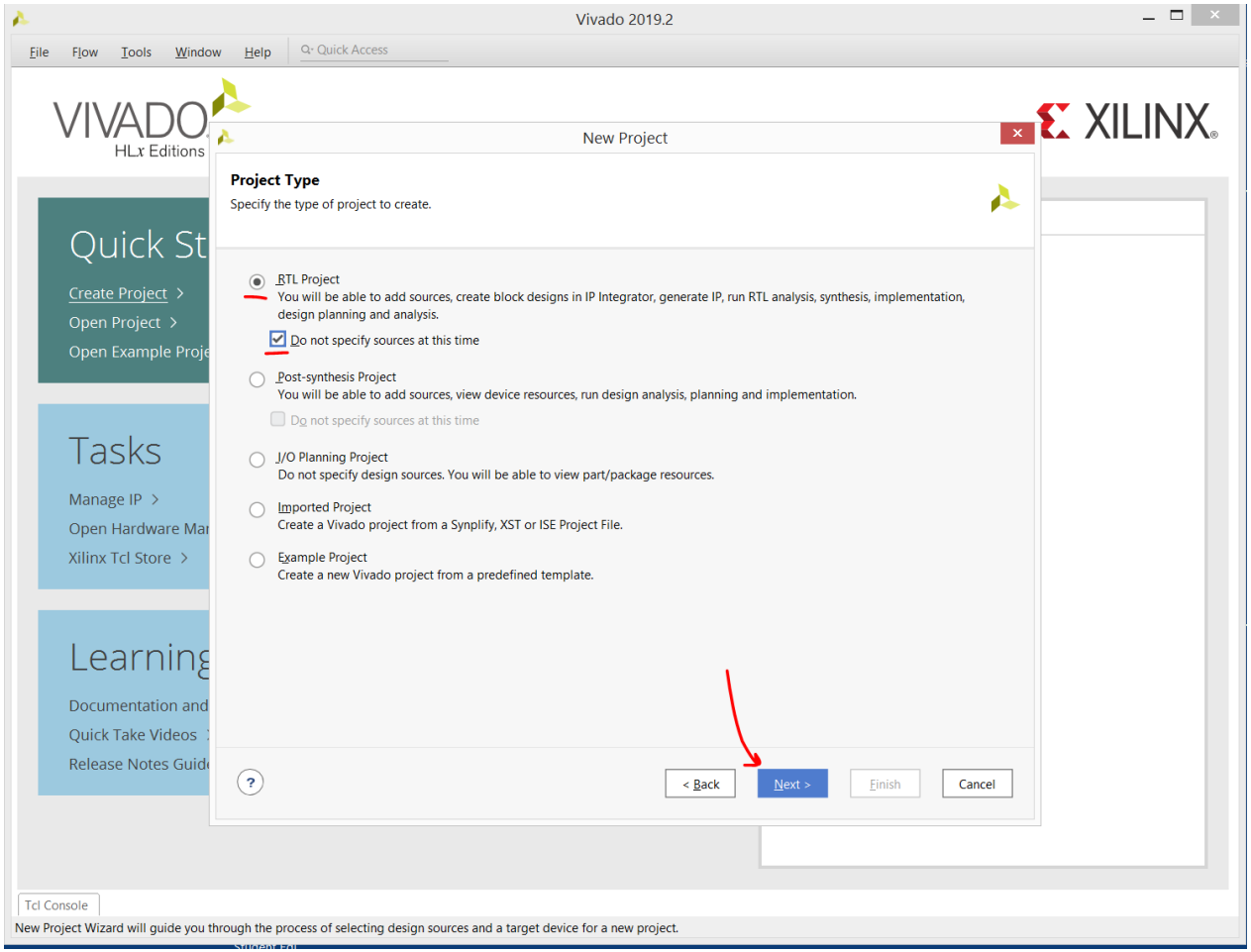
- You will see the New Project dialog box. Enter a **project name** and
- Select a project **location**. Select the location where you want it to be saved.
- Make certain there are **NO SPACES** in the names**!**
- You can use letters, numbers, and underscores.
- Select the "Create project sub-directory". This keeps things neatly organized with a directory for each project and helps avoid problems.
- Click the "**Next** >" button to proceed.



Vivado Tutorial

Specify the Project Type

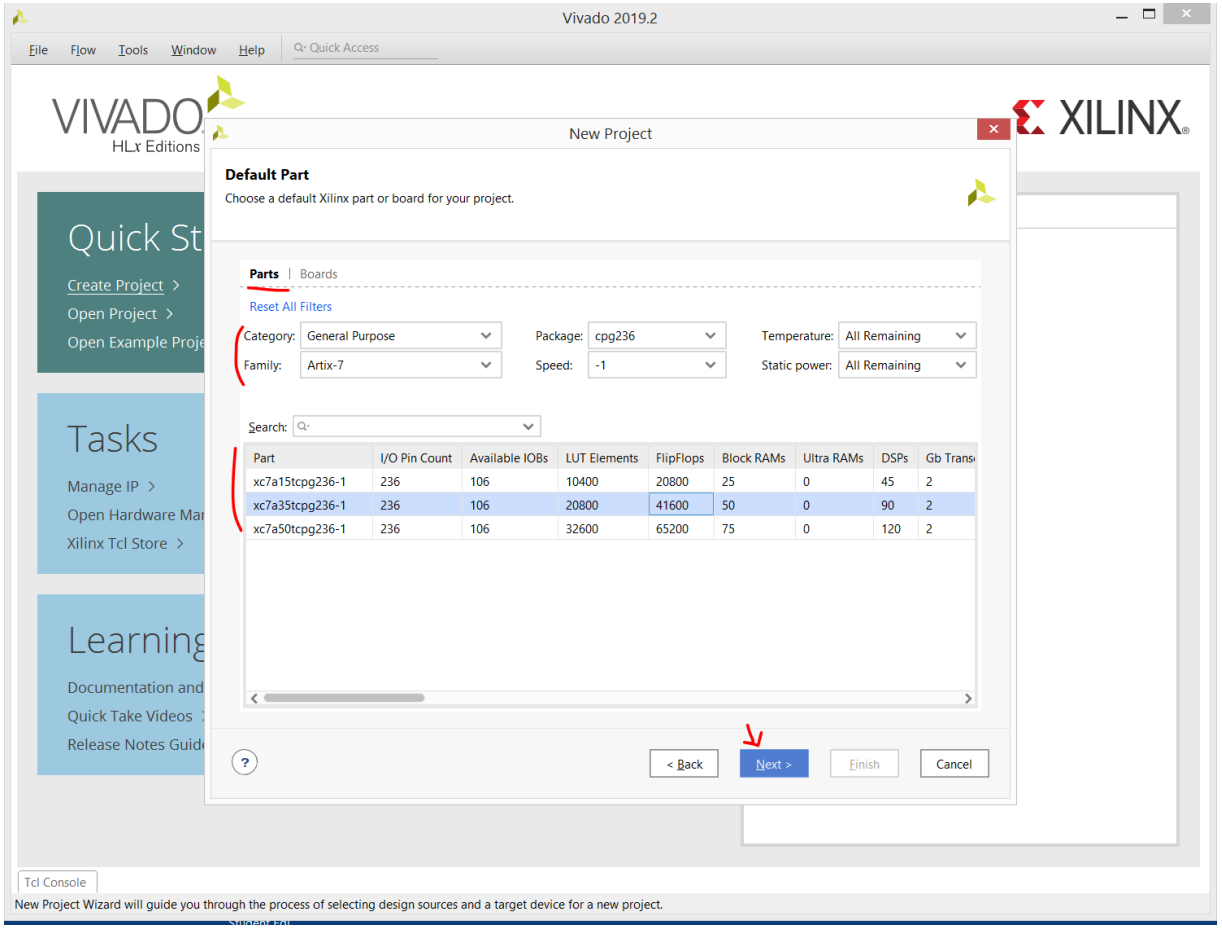
- Select the "**RTL Project**"
- Select the "**Do not specify sources at this time**" check-box.
- If you don't select the check-box the wizard will take you through some additional steps to optionally add preexisting items such as VHDL or Verilog source files, Vivado IP blocks, and. XDC constraint files for device pin and timing configuration. For this first project you will add the necessary items later.
- Click the "**Next** >" button to proceed.



Vivado Tutorial

Specify the part family to be Used

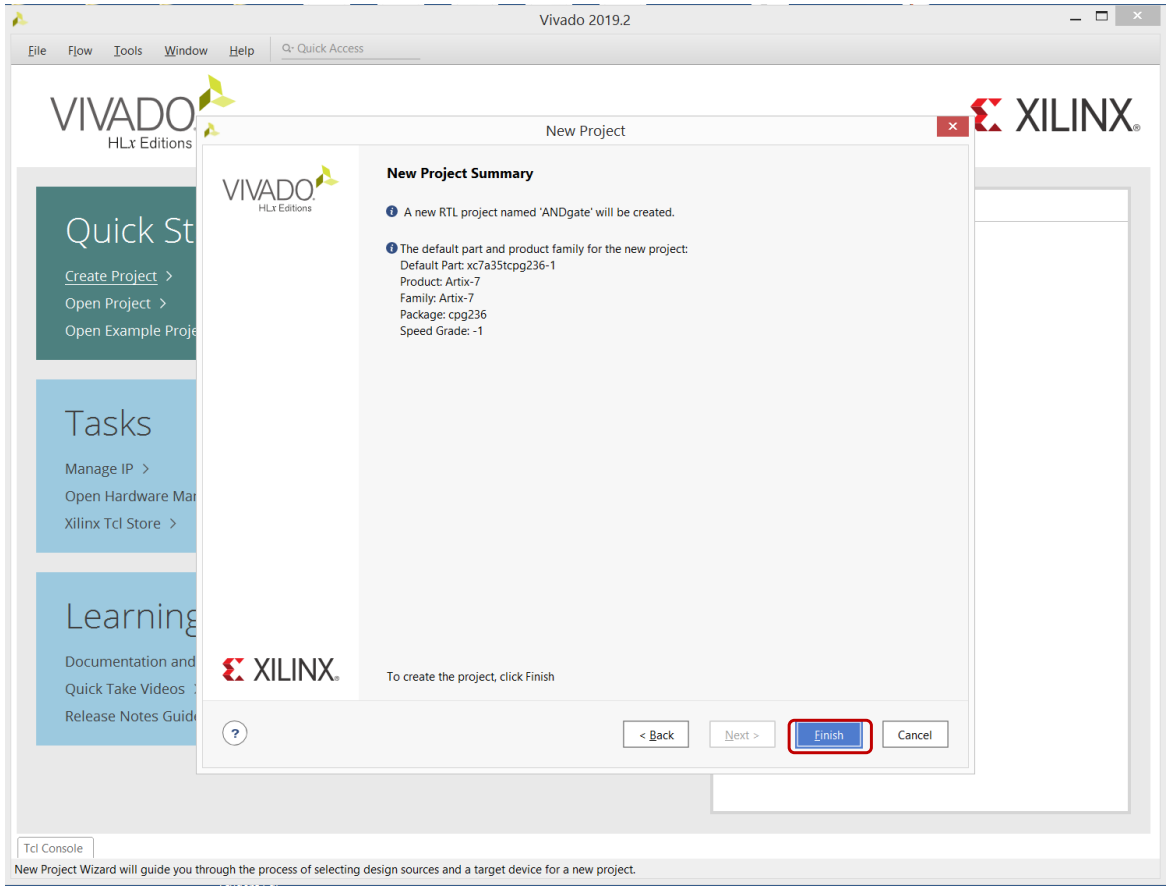
- You need to filter down to and select the specific part number for your project.
- Once you select the correct device click the "**Next**" button to proceed.



Vivado Tutorial

Creating a New Project

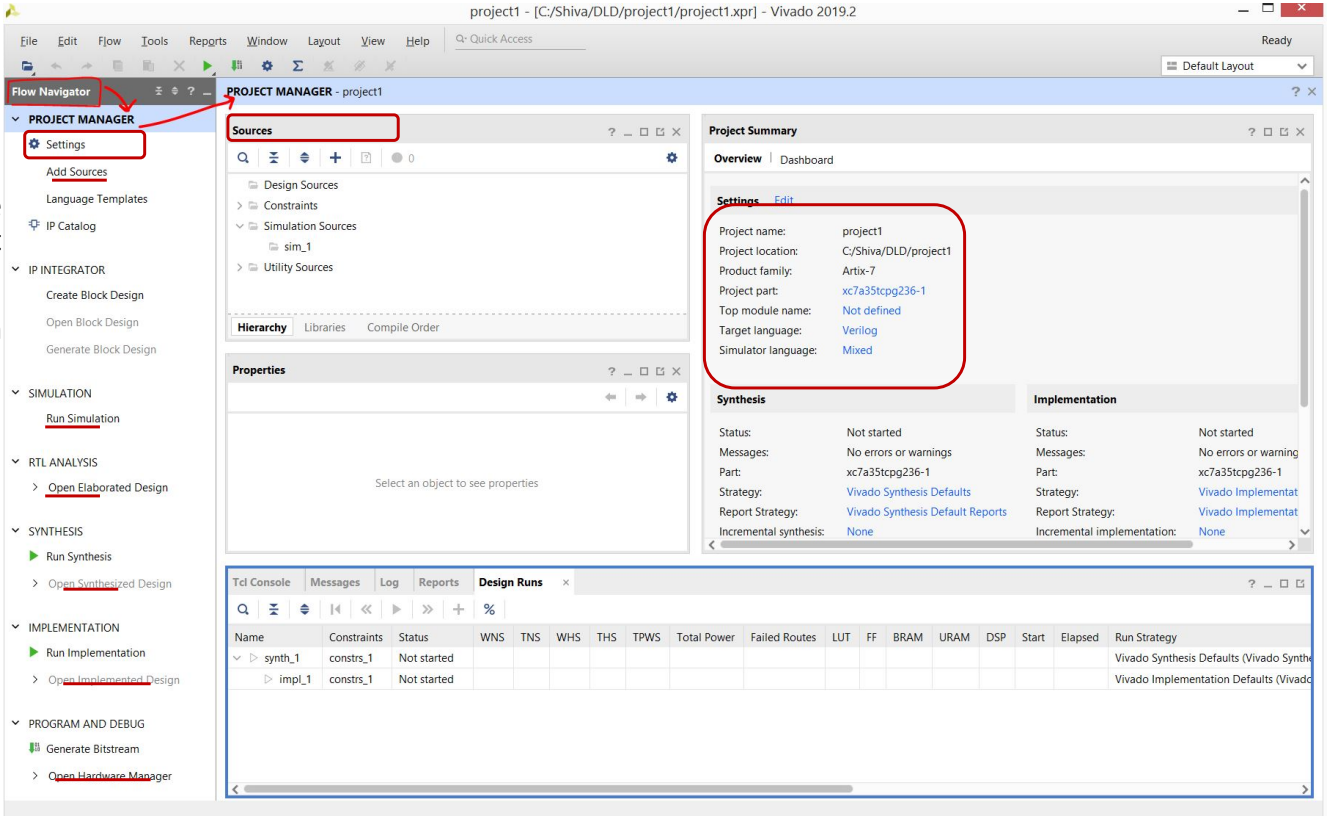
- Click **Finish** to create the project.



Vivado Tutorial

working on the project

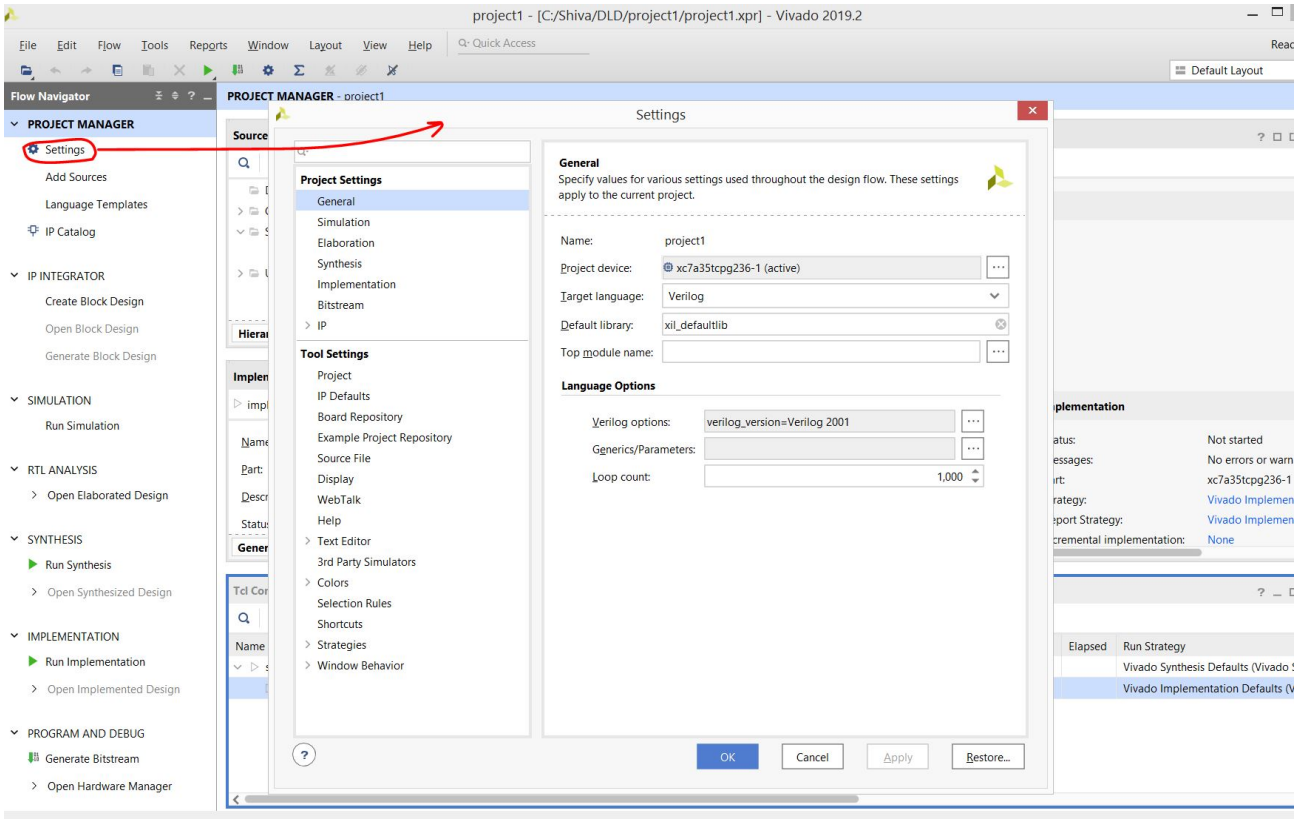
- The Vivado project window contains a lot of information..
- The "**Flow Navigator**" on the left side of the screen has all the major project phases organized from top to bottom.
- "**Project Manager**" will tell you which phase of the design you have open.



Vivado Tutorial

working on the project

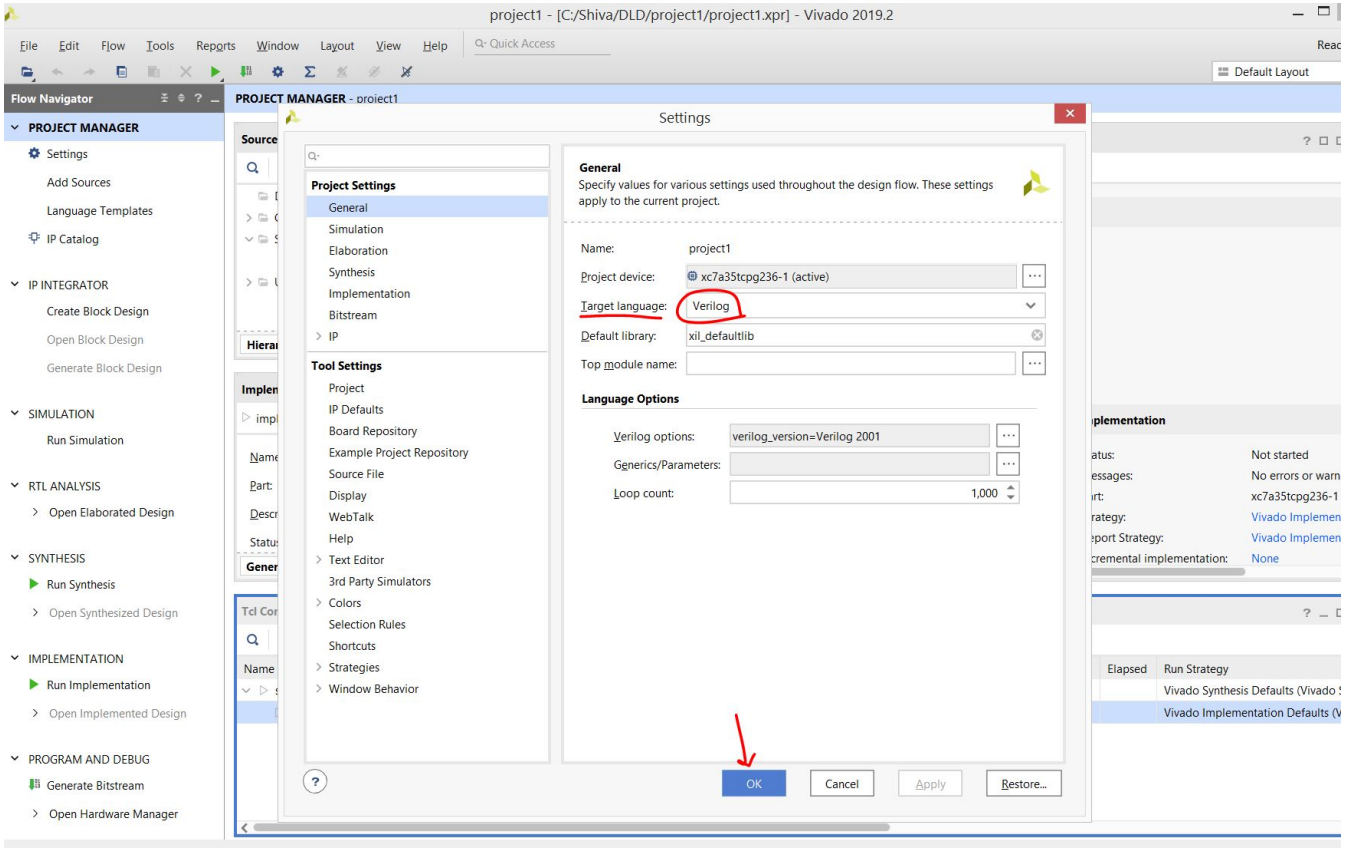
- By clicking on “**Project Settings**” under the Project Manager phase of the Flow Navigator, the setting dialog box will open.



Vivado Tutorial

working on the project

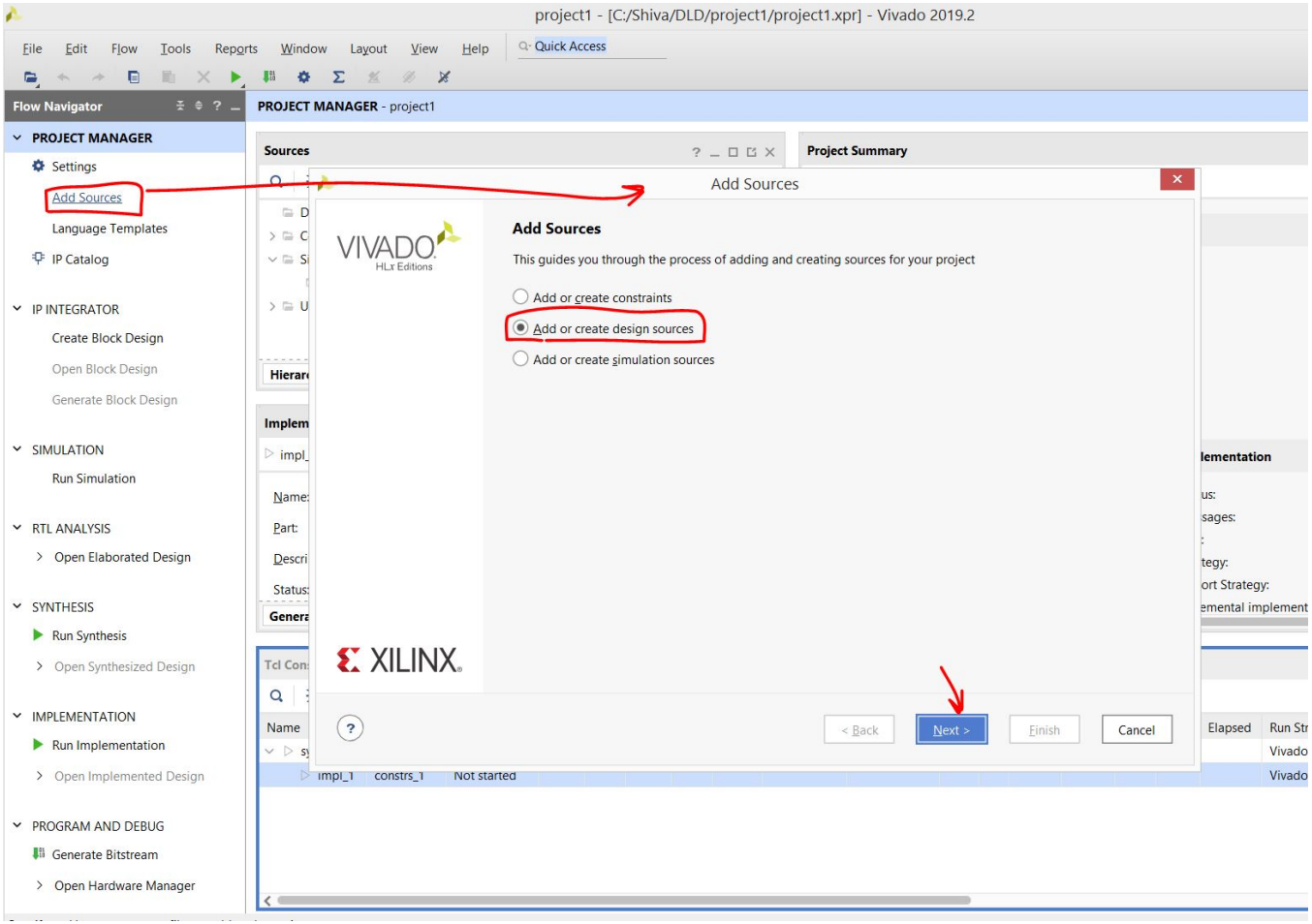
- There are a lot of settings available here for all phases of the project flow, but for now just select “**Vrilog**” from the drop-down for the “Target language” in the “**General**” project settings and click the “**OK**” button.



Vivado Tutorial

working on the project

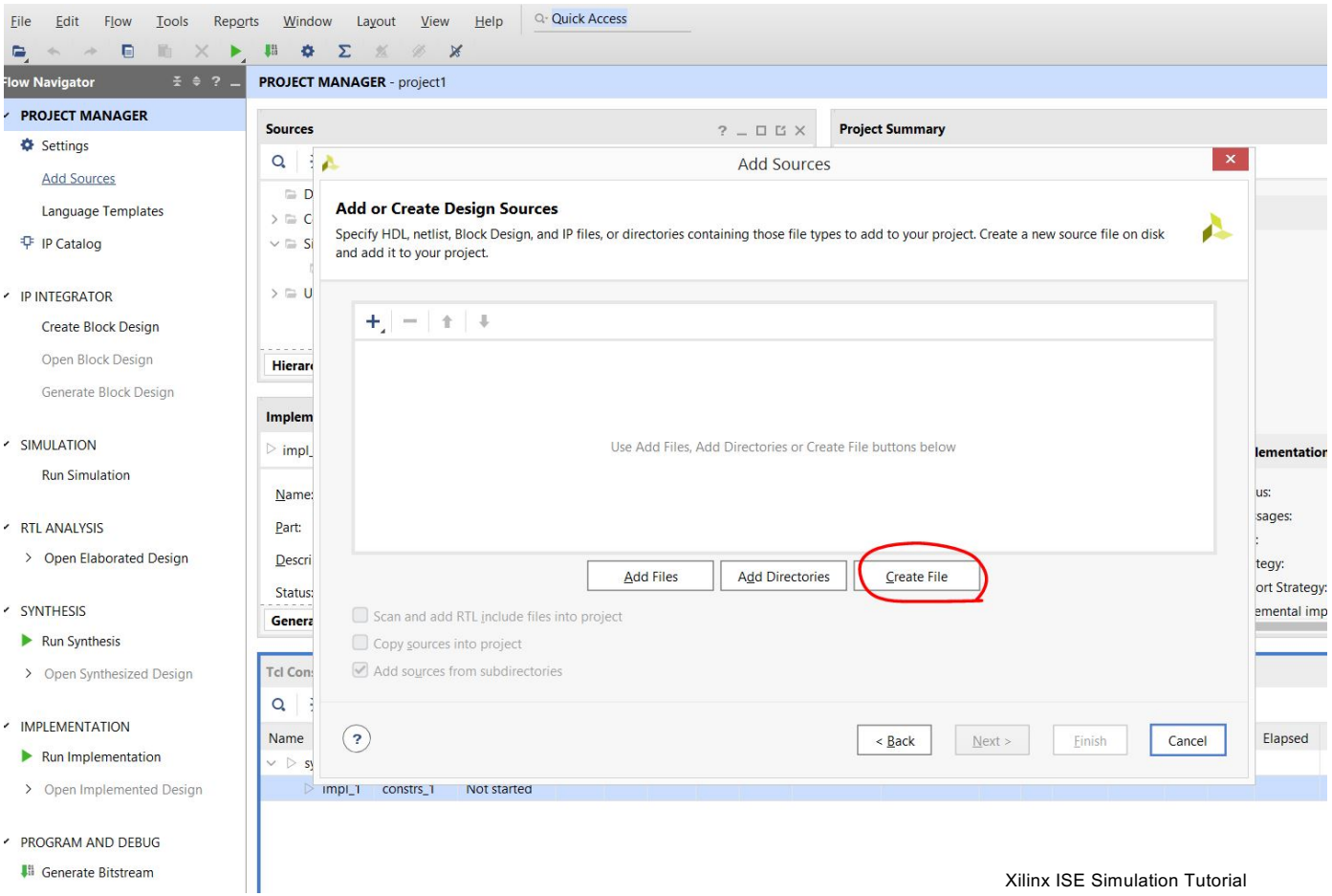
- Now click on “**Add Sources**” under the Project Manager phase of the Flow Navigator.
- Select the “**Add or create design sources**” radial and then click the “**Next** >” button.
-



Vivado Tutorial

working on the project

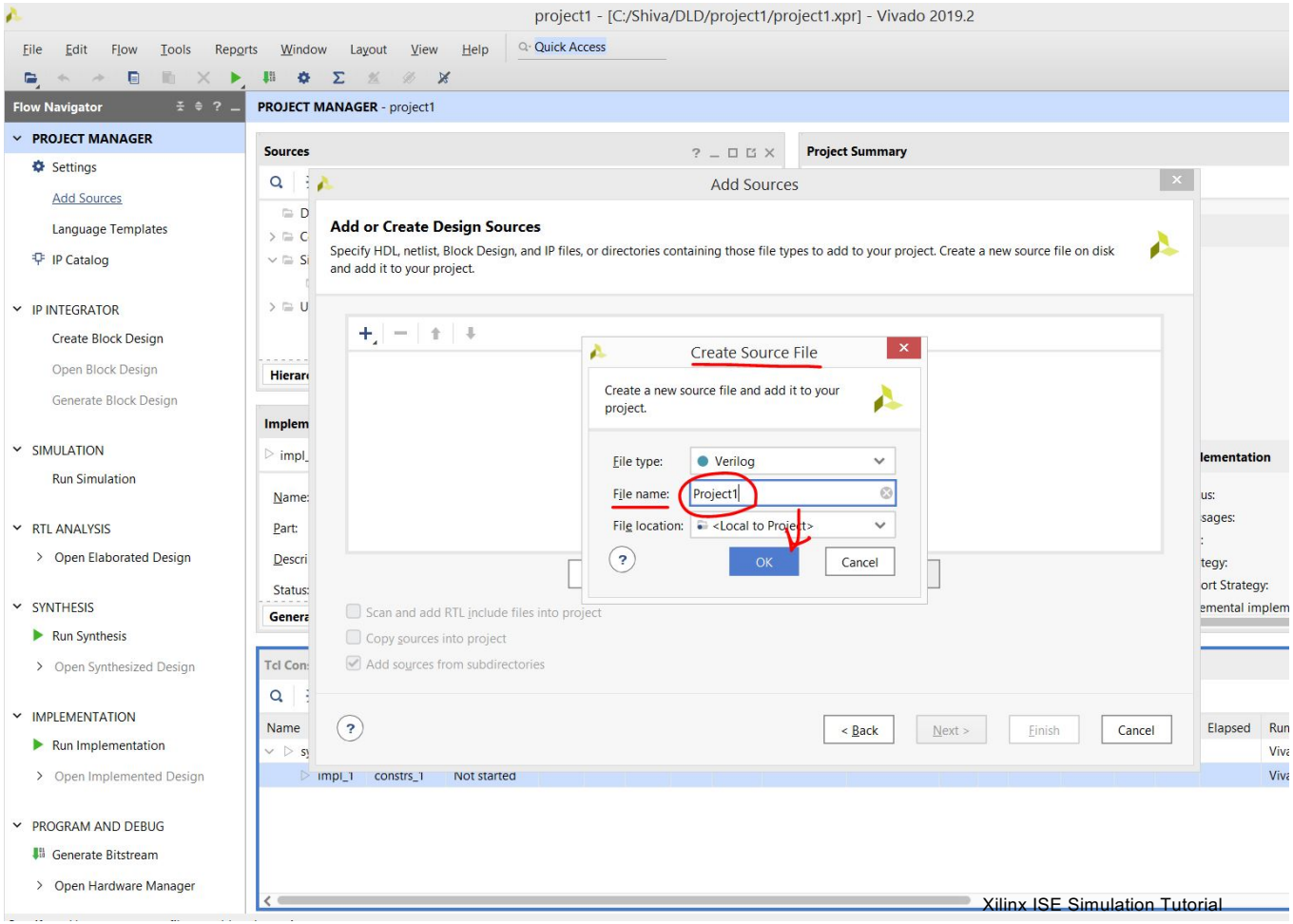
- Click the “**Create File**” button or click the green “+” symbol in the upper left corner and select the “Create File…” option.



Vivado Tutorial

working on the project

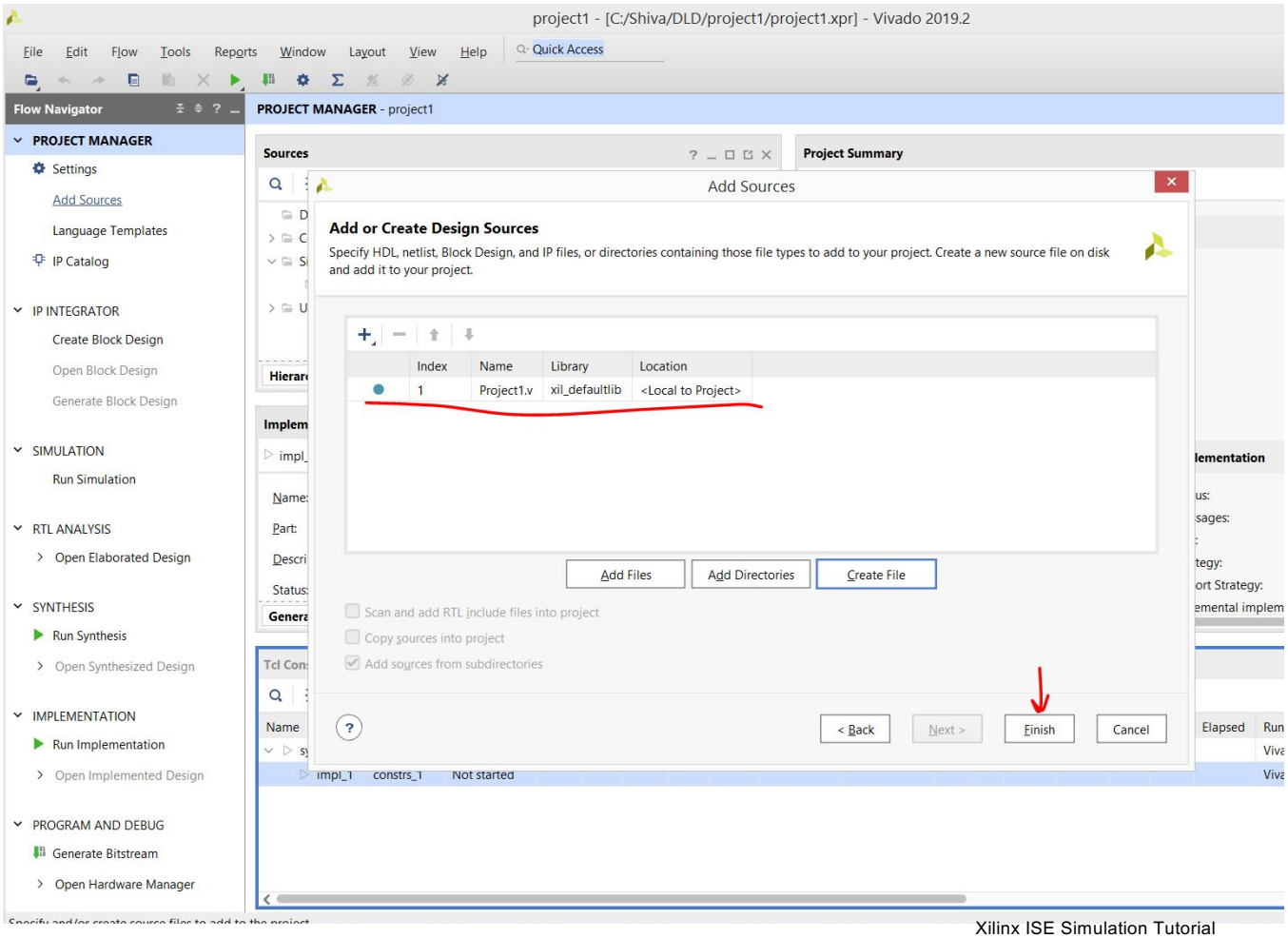
- Make sure the options shown are selected in the "**Create Source File**" popup,
- Enter proper name for the "**File name**". Click the "**OK**" button when finished.
- You can enter anything you like for the 'File name' as long as it's valid, but always make certain there are **NO SPACES!**



Vivado Tutorial

working on the project

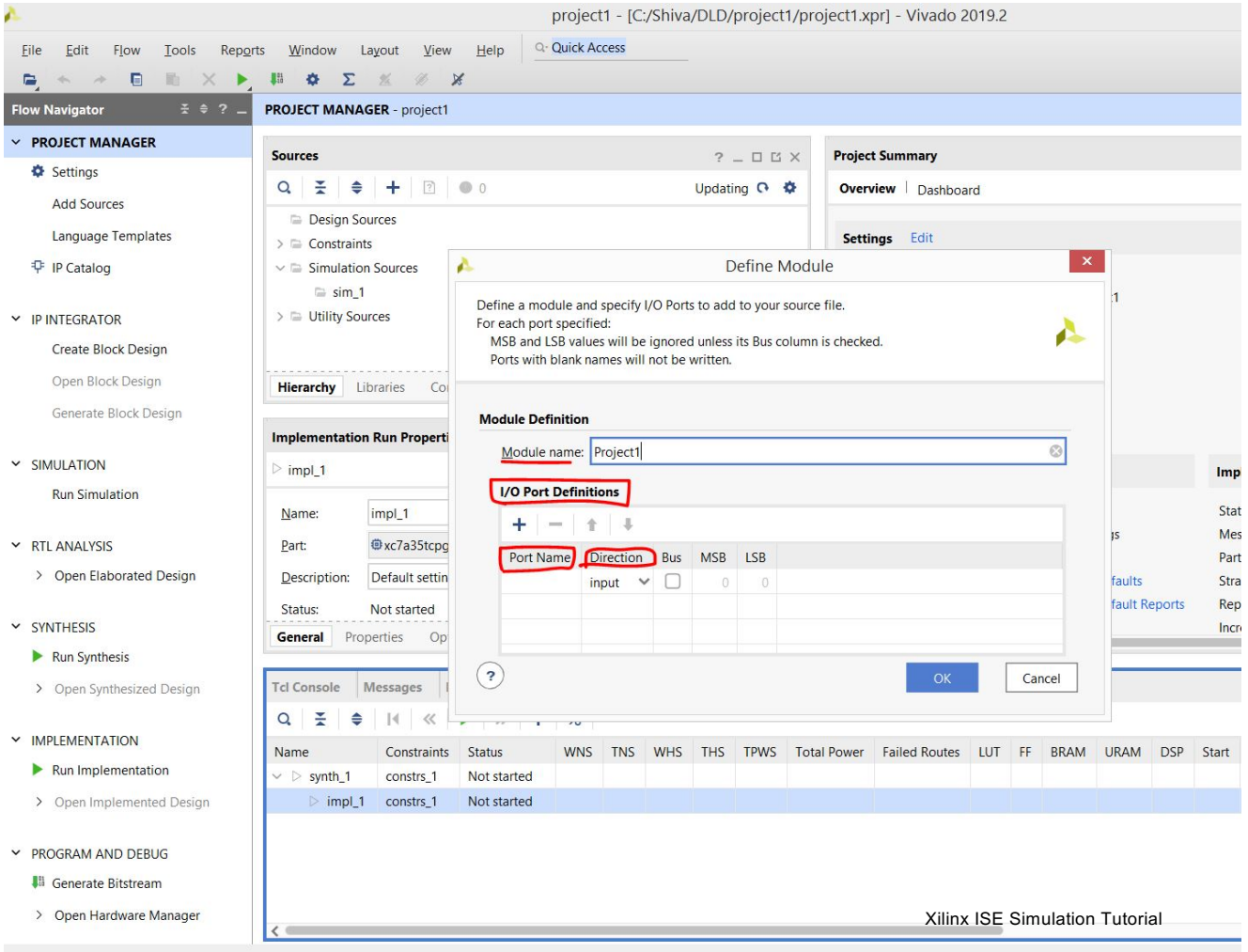
- After file is created, click on **finish**.



Vivado Tutorial

working on the project

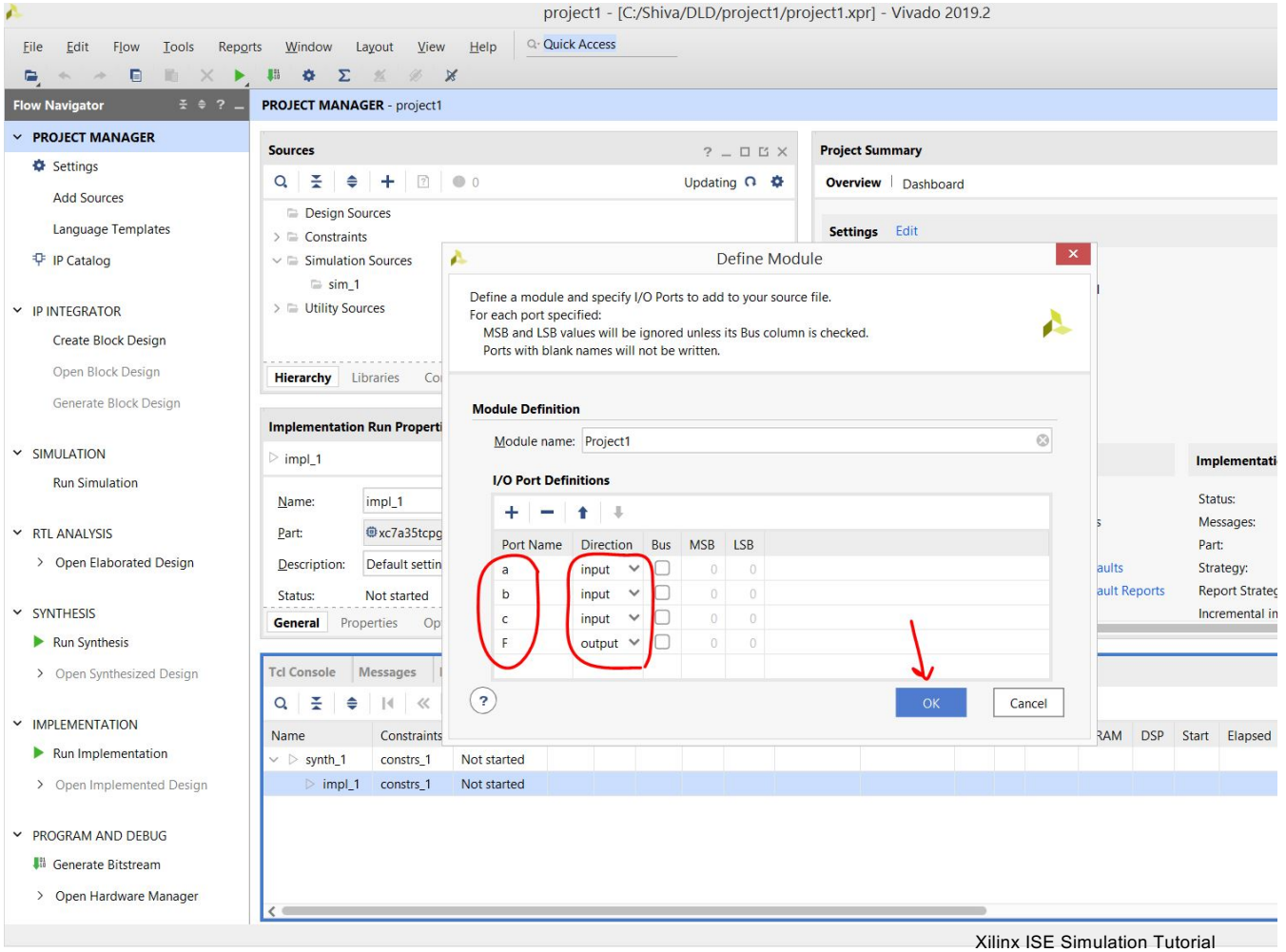
- Additional "**I/O Port Definitions**" can be added by either clicking the "+" symbol in the upper left or by simply clicking on the next empty line.



Vivado Tutorial

Defining input/output

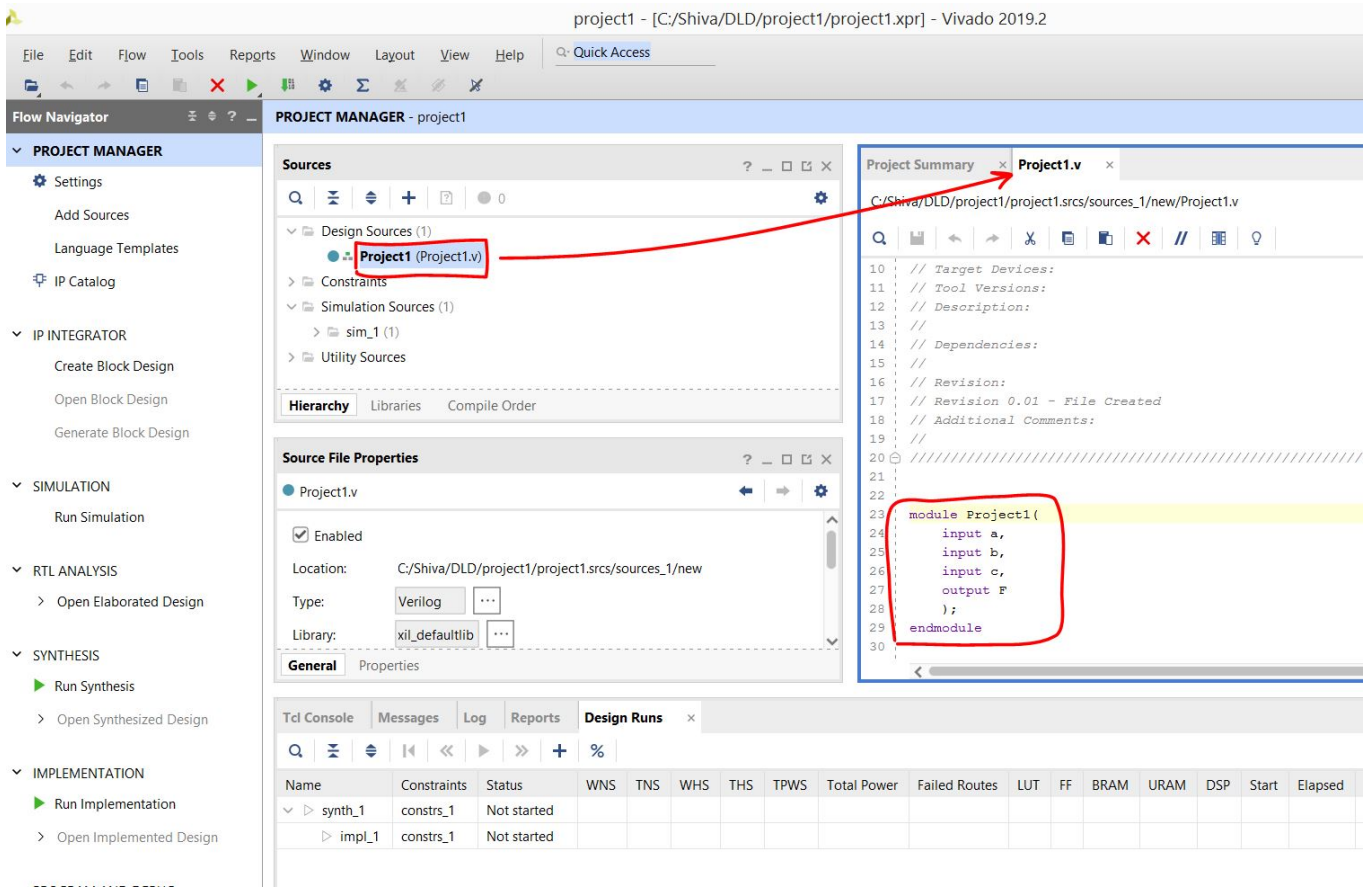
- The “Port Name” used in the code.
- Make sure the proper “**Direction**” is set for each. Click the “**OK**” button when finished.
- Port Types:
  - Input
  - Output
  - Inout : bidirectional port



Vivado Tutorial

Defining input/output

- Click on the project name.
- You can use the “Define module” window to automatically write some of the code for you.
- Note that if you would rather write your own code from scratch you can simply click the “Cancel” button and Vivado will create a completely blank source file inside your project.
- If you click the “OK” button without defining any “I/O Port Definitions” Vivado will still write the basic code structure but the port definition will be empty and commented out for you to uncomment and fill later.



Syntax of a Verilog module Declaration

- A basic syntax for a Verilog module declaration is shown. In the figure.
- Basic unit of design and programming in Verilog is a ***module***.
- It starts with keyword **module**.
- ends with keyword **endmodule**.

```
module module-name (port-name, port-name, ..., port-name);
  input declarations
  output declarations
  inout declarations
  net declarations
  variable declarations
  parameter declarations
  function declarations
  task declarations

  concurrent statements
endmodule
```

```
module ButGate
  (A, B, C, D, Y, Z);
input A, B, C, D;
output reg Y, Z;

always @ (A, B, C, D)
  begin
    if ((A==B)&&(C!=D))
      Y = 1;
    else Y = 0;
    if ((C==D)&&(A!=B))
      Z = 1;
    else Z = 0;
  end
endmodule
```

Verilog Data Types

- Net Data:
  - Represent the physical interconnection between structures. It represent the activity flow between the functional blocks.
- Variable Data:
  - Represent elements to store the data. They are store temporarily and just internally within the module.

```
input  identifier, identifier, ..., identifier;
output identifier, identifier, ..., identifier;
inout  identifier, identifier, ..., identifier;

input  [msb:lsb] identifier, identifier, ..., identifier;
output [msb:lsb] identifier, identifier, ..., identifier;
inout  [msb:lsb] identifier, identifier, ..., identifier;
```

```
wire identifier, identifier, ..., identifier;
wire [msb:lsb] identifier, identifier, ..., identifier;

tri identifier, identifier, ..., identifier;
tri [msb:lsb] identifier, identifier, ..., identifier;
```

```
reg [7:0] byte1, byte2, byte3;
reg [15:0] word1, word2;
reg [1:16] Zbus;
```

Net Data Type:

- wire: represent a node or connection
- tri: represent a tri-state node
- supply 0: logic 0
  supply 1: logic 1

Variable Data Type:

- reg: unsigned variable
- reg signed: signed variable
- integer: signed 32-bit variable
- time, real, realtime

Bus Declaration:

- <data type> [MSB: LSB] <signal name>

Or

- <data type> [LSB:MSB] <signal name>

Example:
reg [7:0] out; //a 8-bit register
wire [7:0] out;

Verilog Data Types

- An input/output signal declared as described above is one bit wide. Multiple bit or “vector” signals can be declared by including a range specification, [msb:lsb], in the declaration.
- Here msb and lsb are integers that indicate the starting and ending indexes of the individual bits within a vector of signals.

Bus Declaration:

- <data type> [MSB: LSB] <signal name>

Or

- <data type> [LSB:MSB] <signal name>

Example:

reg [7:0] out; //a 8-bit register

wire [7:0] out;

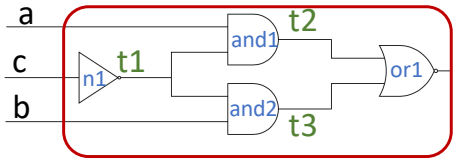
```
wire identifier, identifier, ..., identifier;
wire [msb:lsb] identifier, identifier, ..., identifier;

tri identifier, identifier, ..., identifier;
tri [msb:lsb] identifier, identifier, ..., identifier;
```

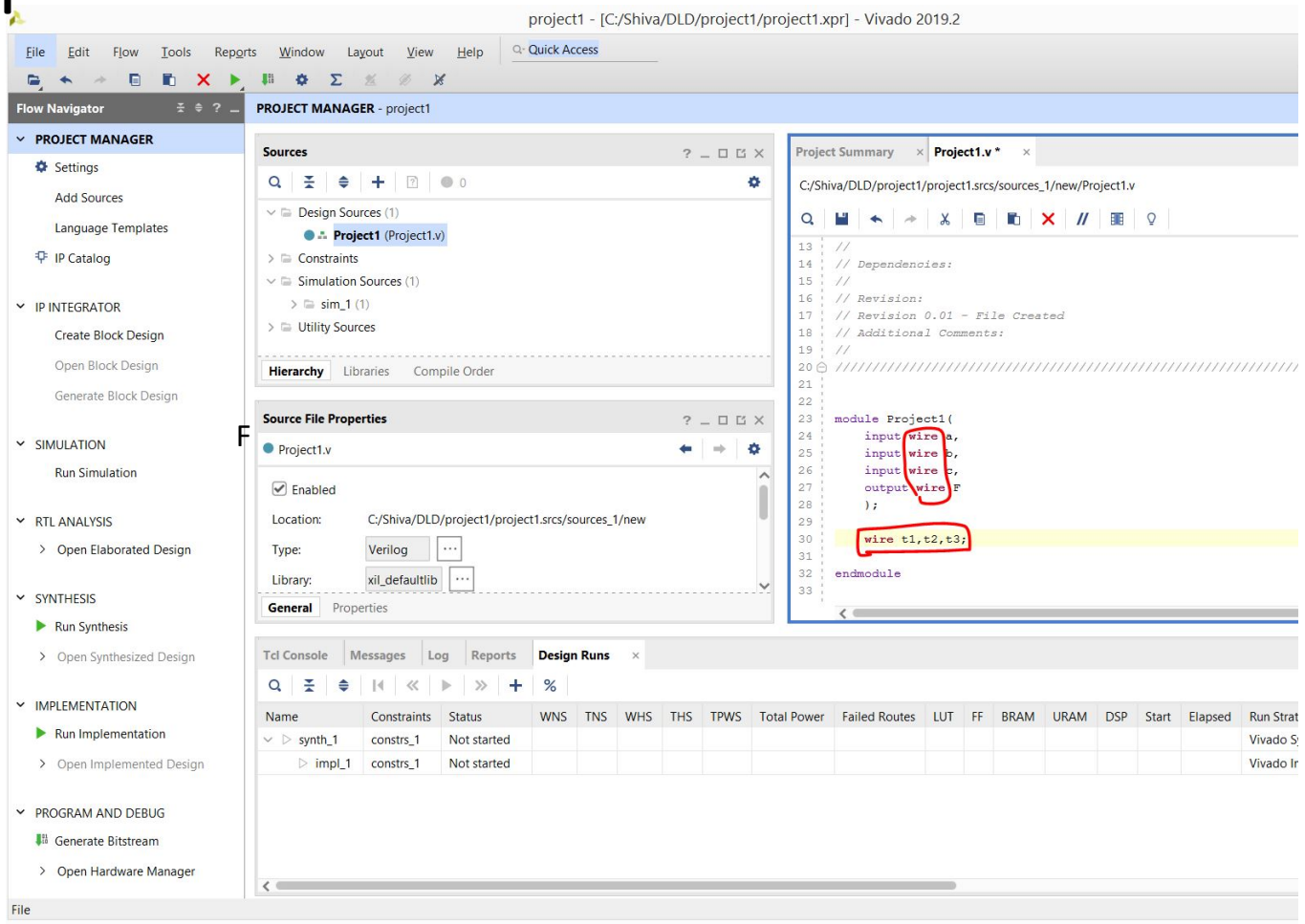
```
reg [7:0] byte1, byte2, byte3;
reg [15:0] word1, word2;
reg [1:16] Zbus;
```

Vivado Tutorial

- Defining input/output

- Define your input/outputs as wire.
- Based on our logic diagram we need 3 more connection.
- We name them t1,t2,t3.

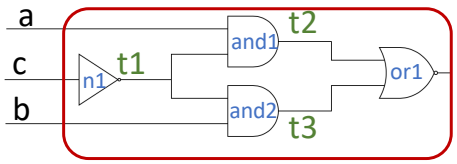


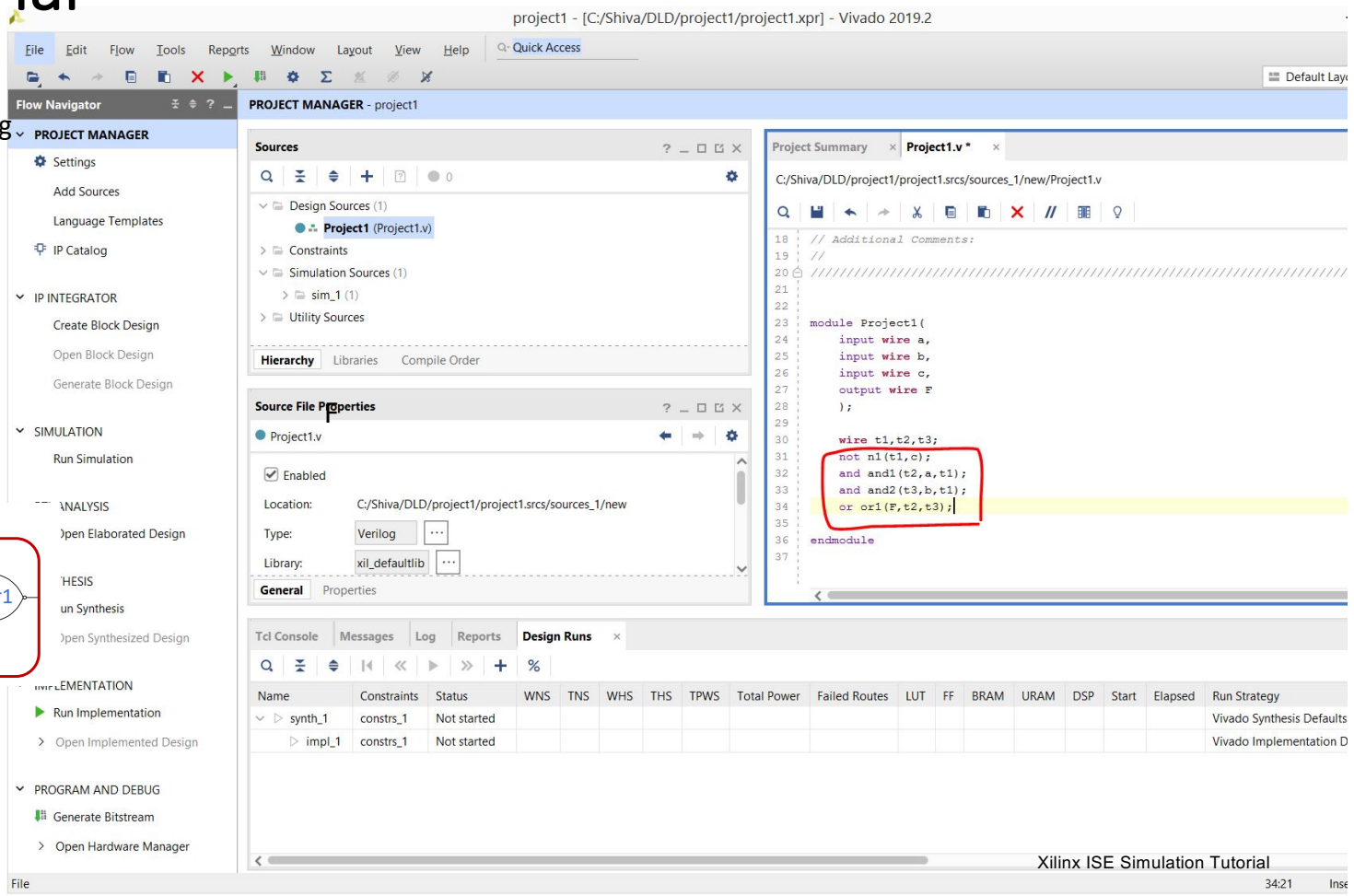


Vivado Tutorial

Statements

- Write the code using Verilog predefined gates.





Verilog Built-In Gates

and	xor	bufif0
nand	xnor	bufif1
or	buf	notif0
nor	not	notif1

Example:

```
out = and(a,b);
```

```
and and1(o1,b1,a1);
and and2(o2,b2,a2);
```

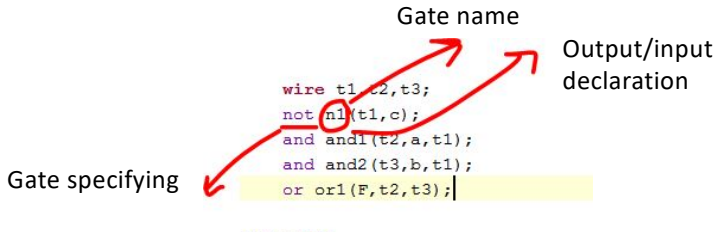
Verilog "Case Equality" Operators

- Used to compare values
- Returns a one bit scalar value of Boolean true or false ('0' or '1')

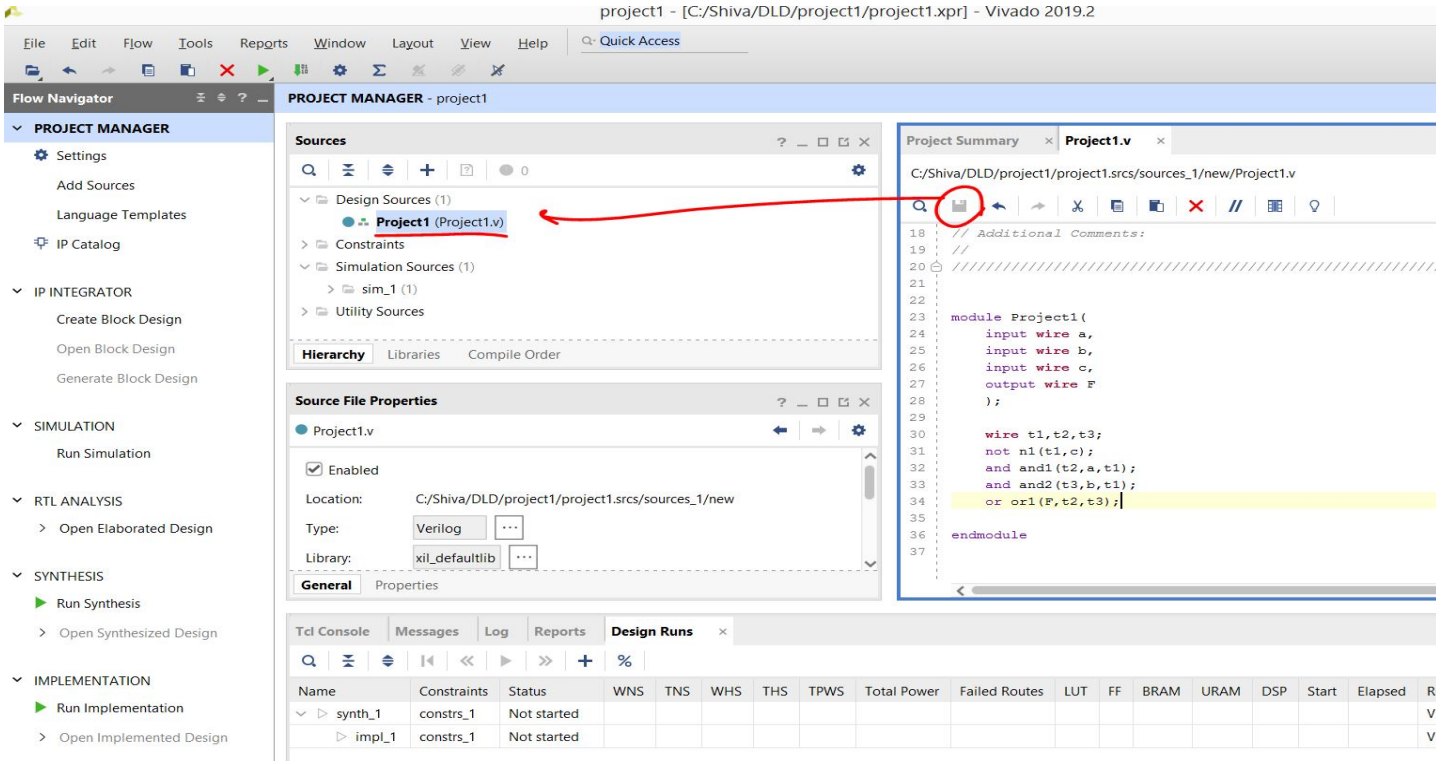
Operator	*Operation*
===	case equality
!==	case inequality

Vivado Tutorial

- Statements



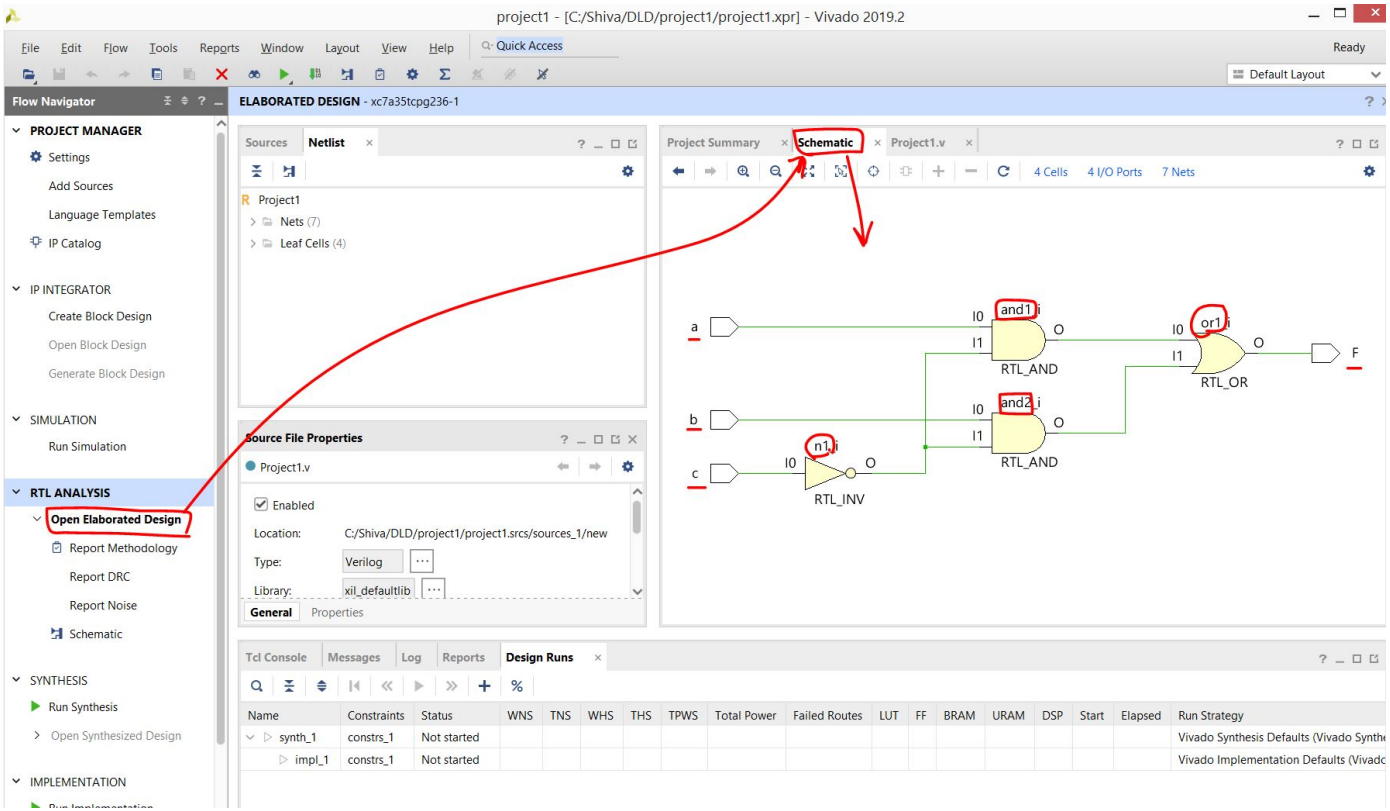
- Save your code.



Vivado Tutorial

RTL Analysis

- On the RTL analysis, click on the open elaborated design to the optimized design.



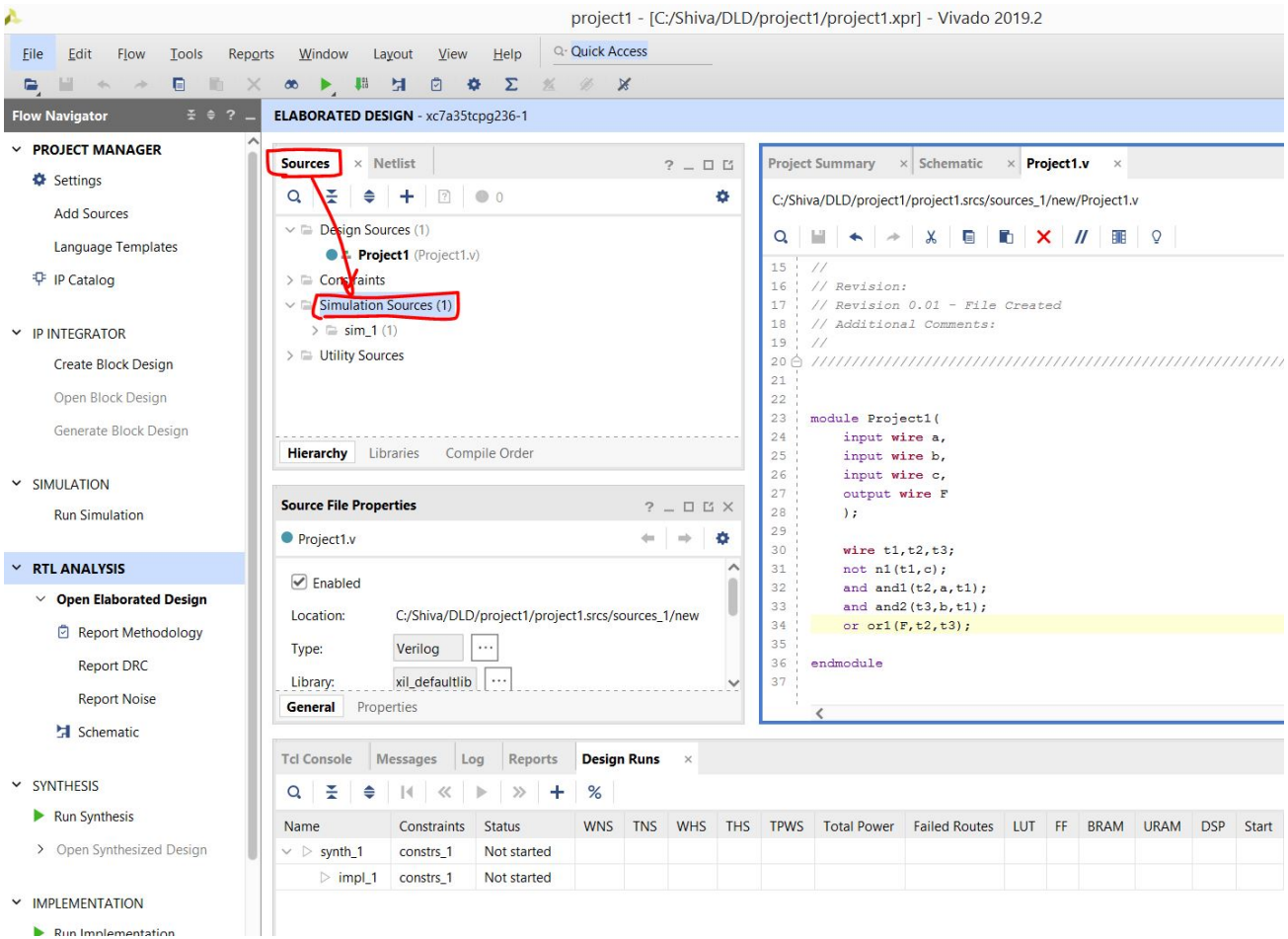
Vivado Tutorial

Simulation

We finished our Verilog module, now we need to simulate it !

To create a simulation file right click on the simulator sources.

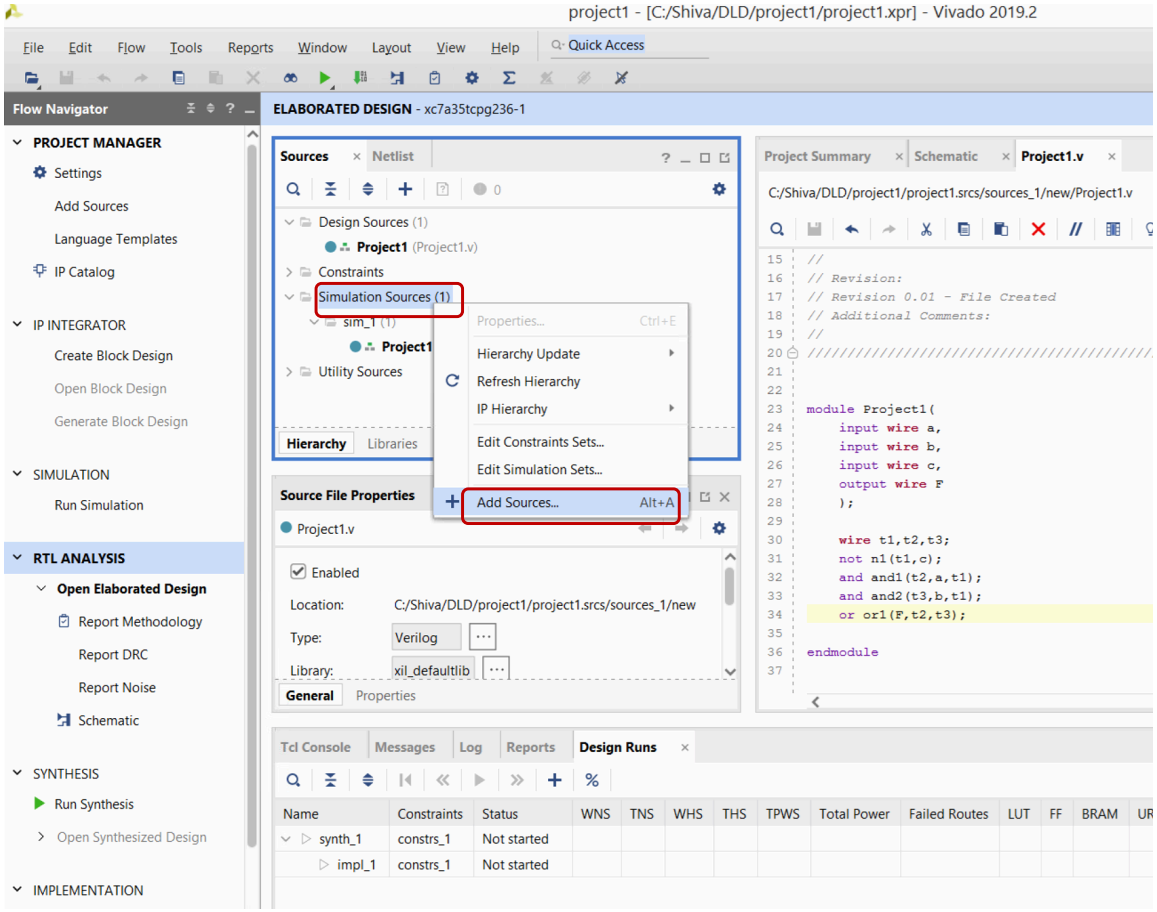
Add source> Create simulation source.



Vivado Tutorial

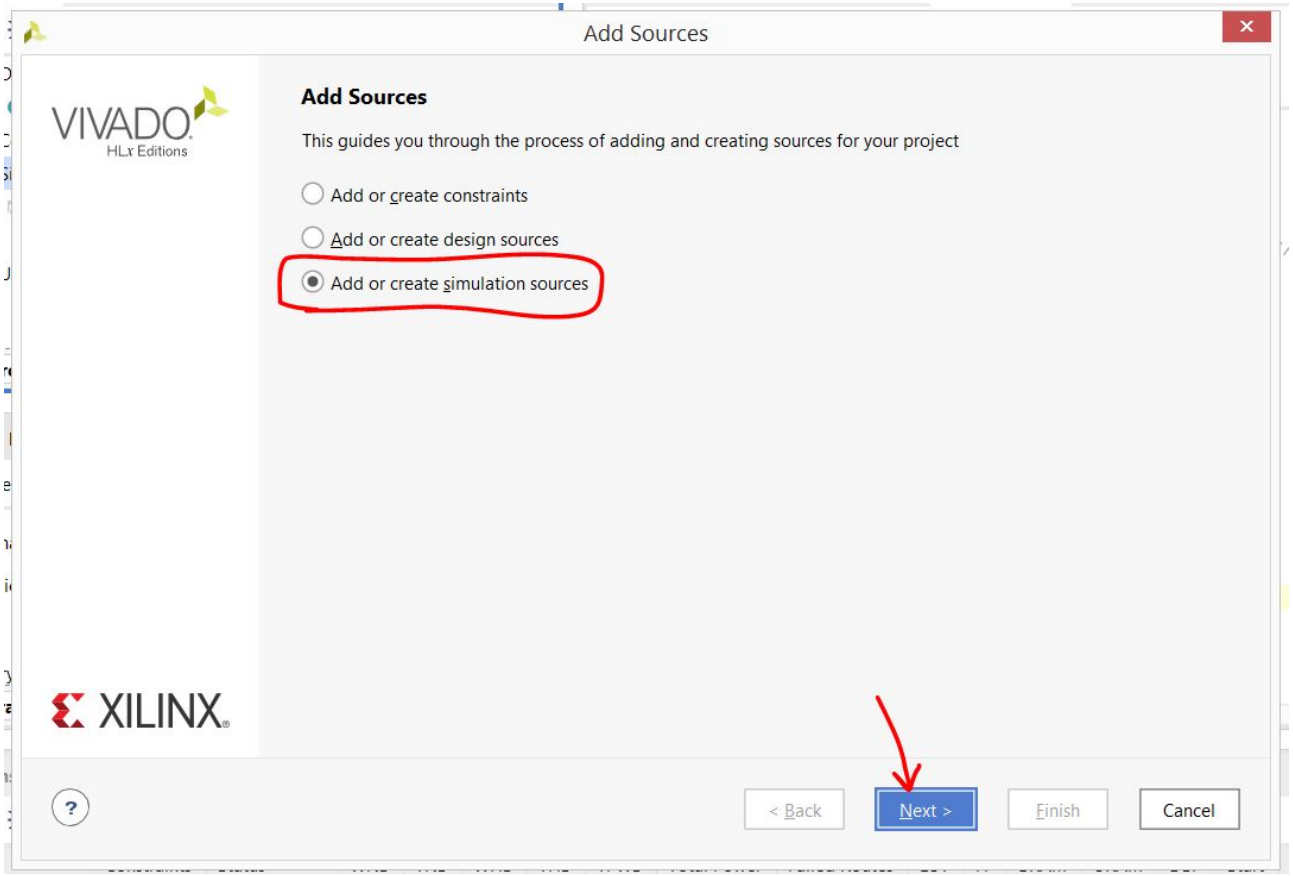
- Simulation

Add source > Create simulation source.



Vivado Tutorial

- simulation



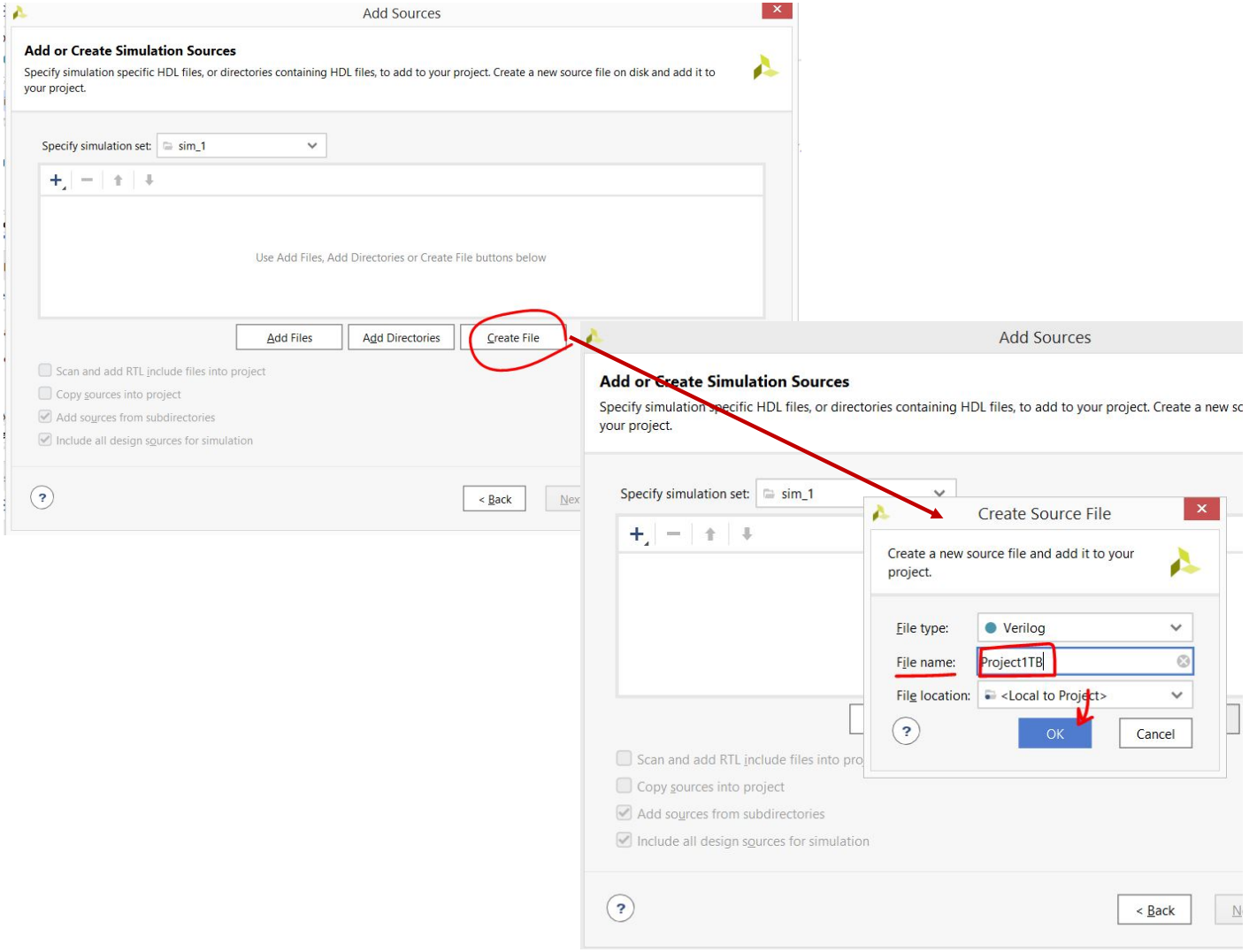
Vivado Tutorial

simulation

Create file >

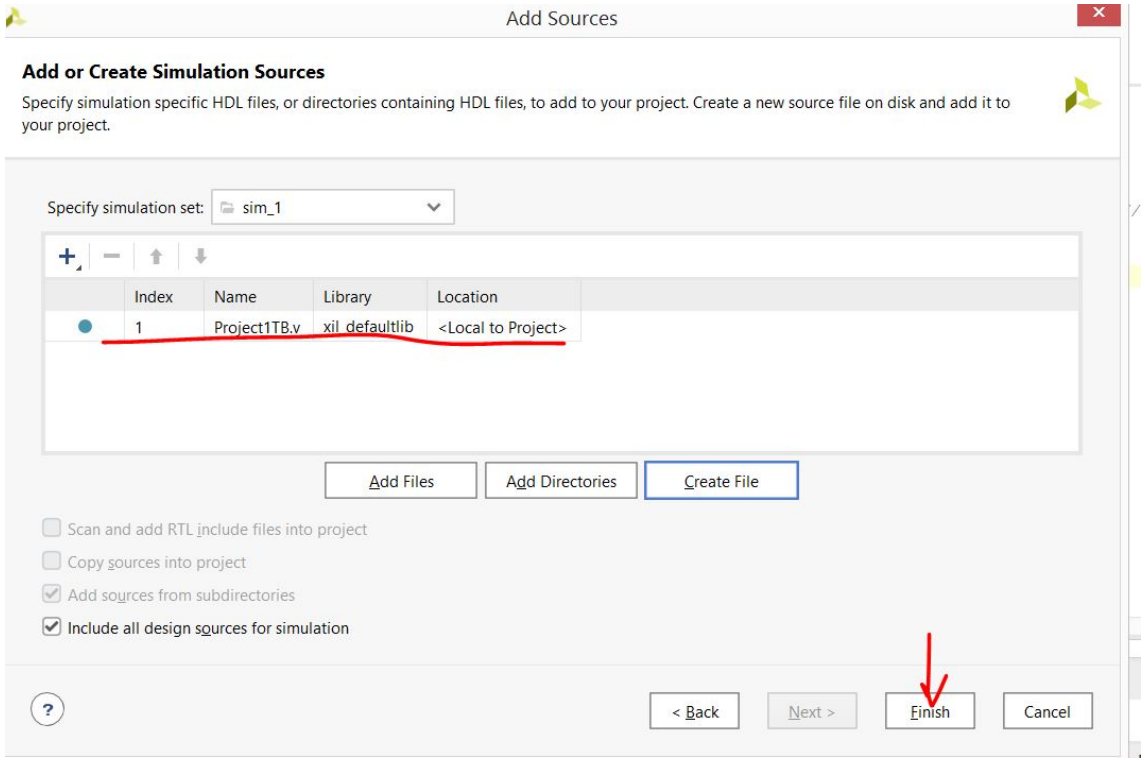
Write the name for the file.

Click OK



Vivado Tutorial

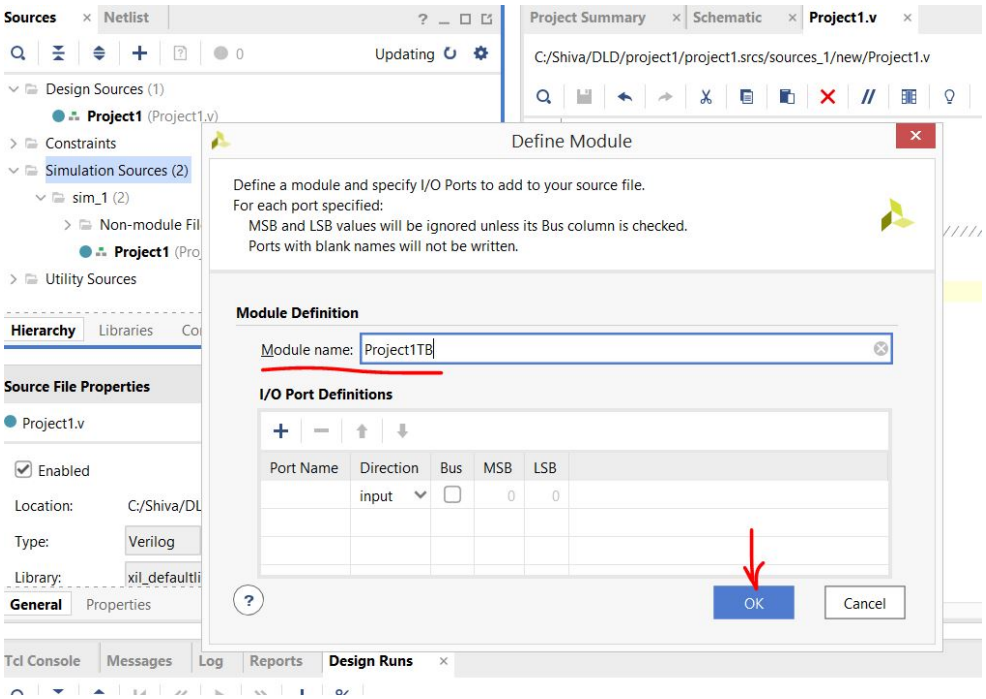
- simulation



Vivado Tutorial

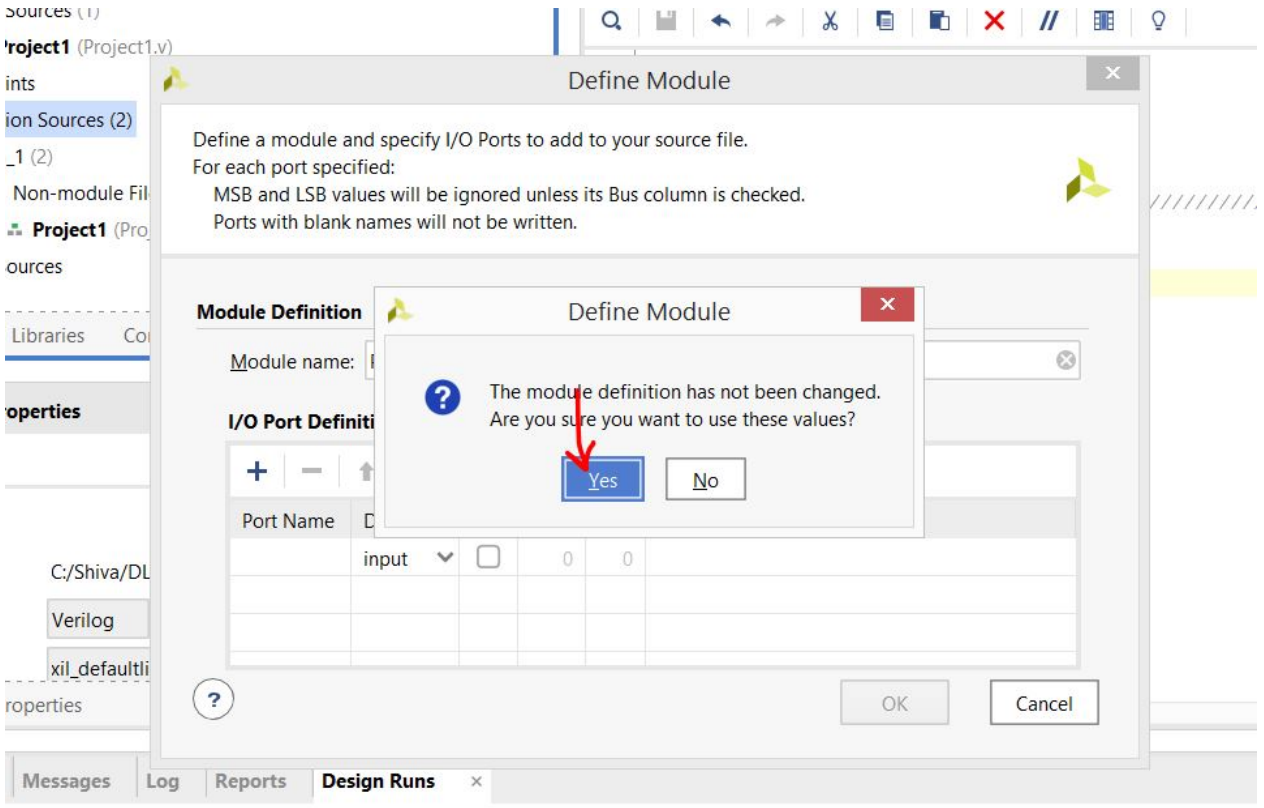
simulation

write the module name and OK !



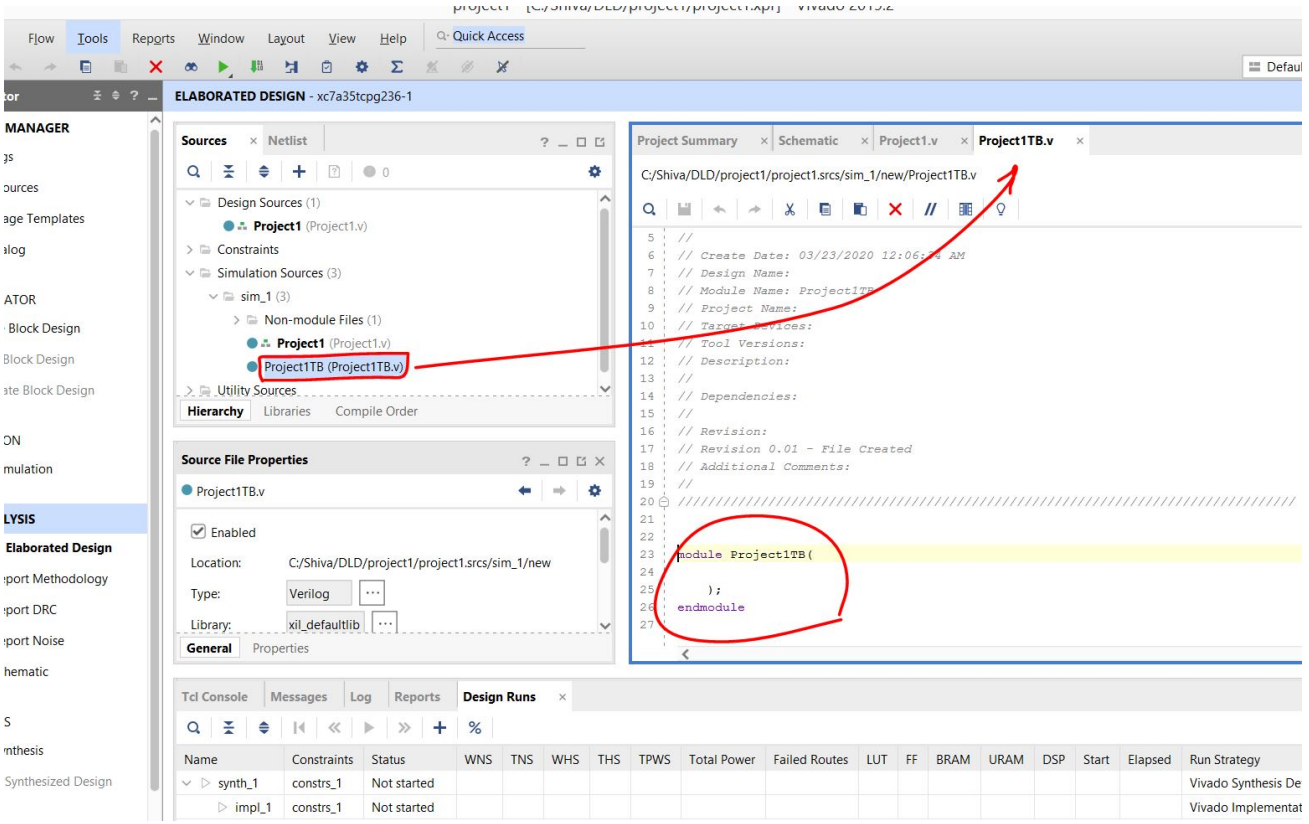
Vivado Tutorial

- simulation



Vivado Tutorial

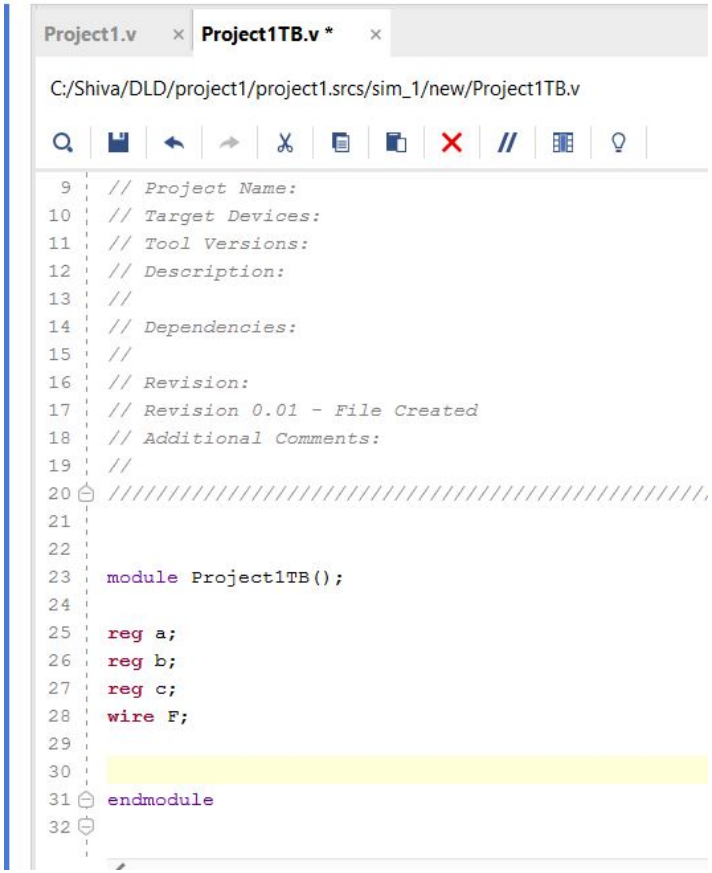
- Open the simulation file created, under simulation sources.
- Since the test bench won't be synthesized, it won't have any input/outputs they already are in the syntaxe module.



Vivado Tutorial

Simulation module

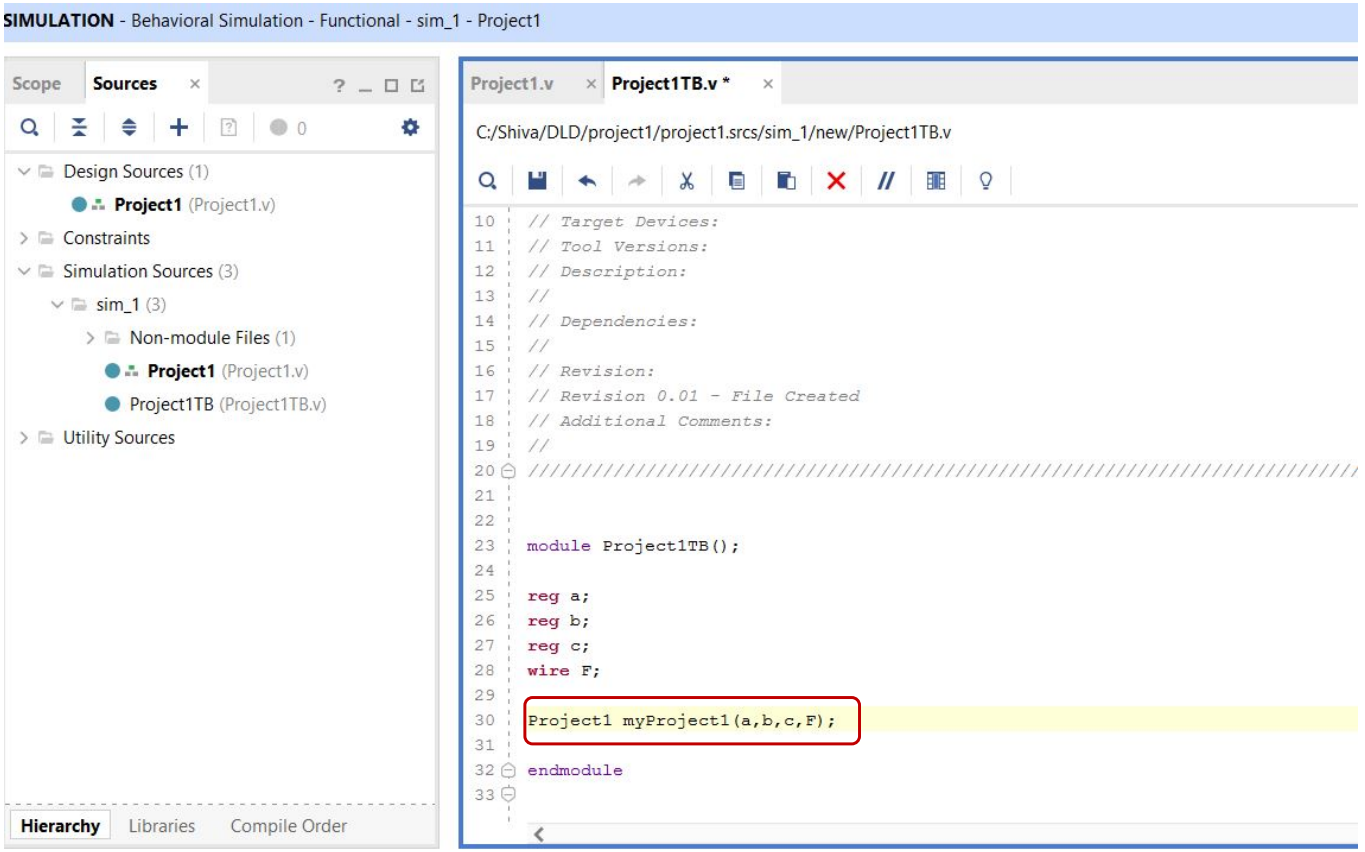
- Copy all the input and output
- Every **input** will be turned into *register* (**reg**).
- A register retains its value until a new value is assigned to it so it can store a value in it.
- All **outputs** will be turned into test **wires**, which are simple wires that can have values assigned to them but they don't store the values.



Vivado Tutorial

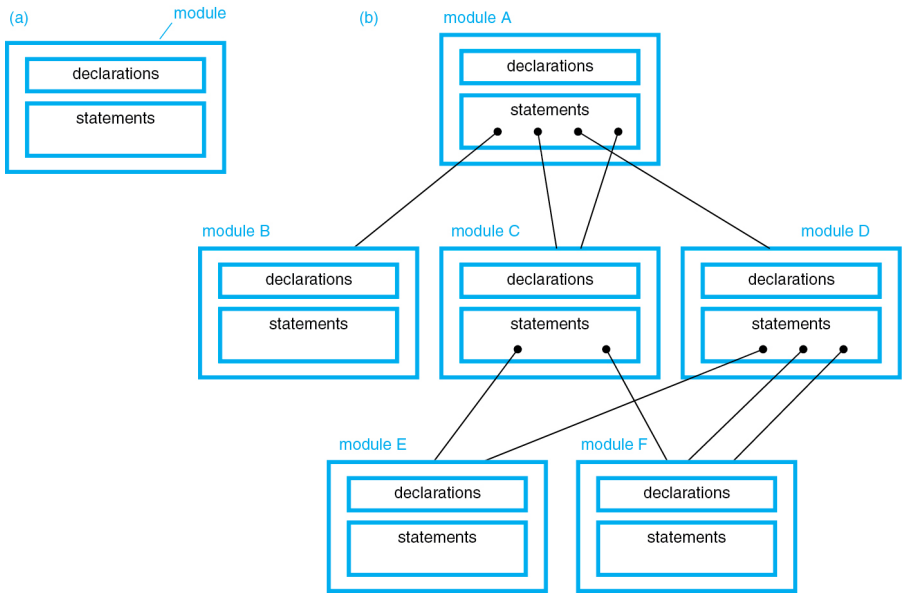
Simulation module

- We define which module we want to simulate.
- Give it a name and write the output and inputs.



modules instantiating other modules

- Verilog modules can use a mix of behavioral and structural models, and may do so hieratically as shown in the figure.
- A higher level module may use lower level module multiple times, and multiple top level modules may use the same lower level one.
- The value can be passed between the modules **only by using declared input and output signals.**



modules instantiating other modules

- Here we instantiate test1 twice in test2.
- We call the module of lower-level component that we instantiate
- Indicate it with a unique instance_name
- List the ports

<module_name> <instance_name> (port_list);

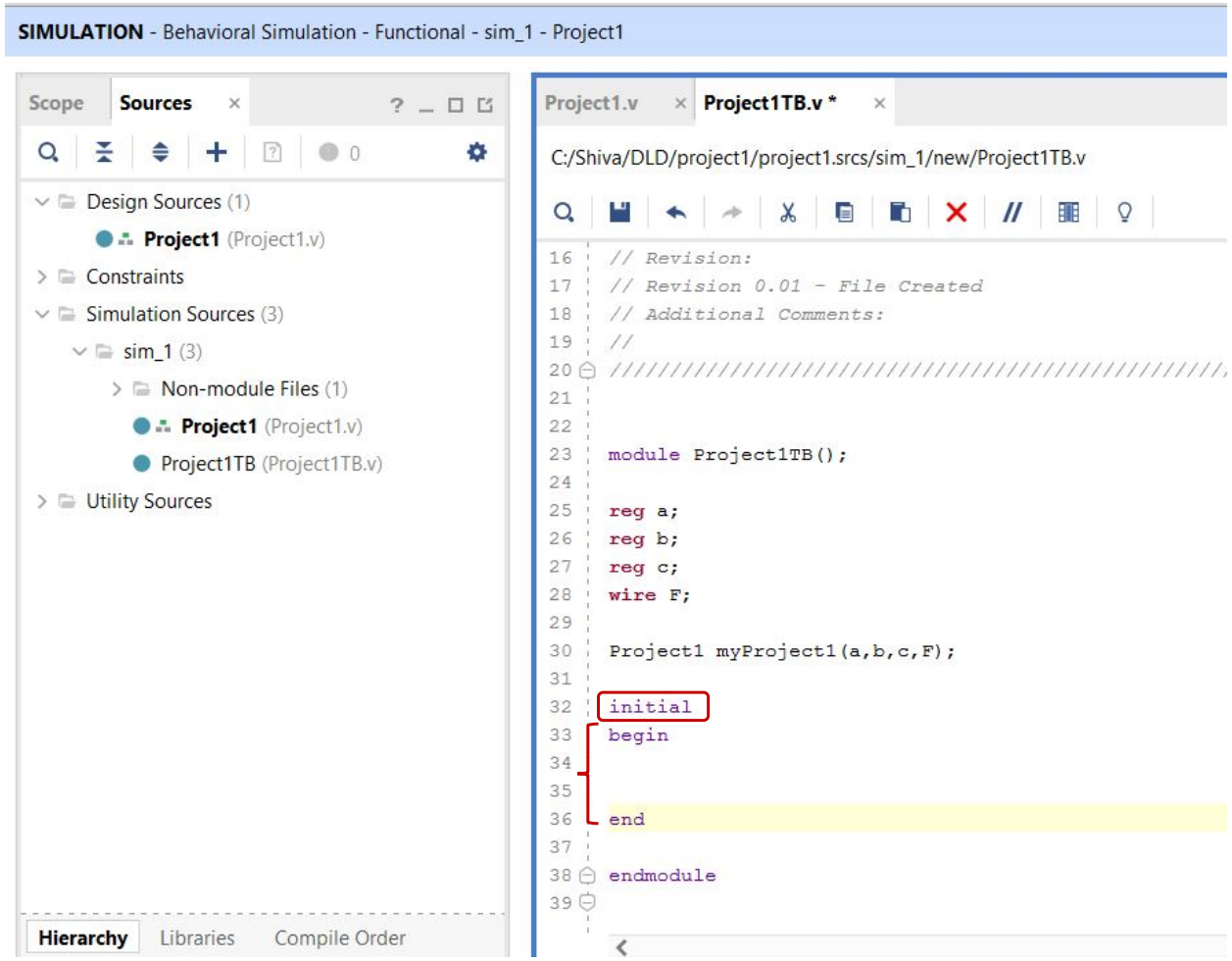
```
module test1 (a,b,c);
        input a,b;
        output c;
….
….
….
endmodule
```

```
module test2(d,e,f);
        input d,e;
        output f;
….
wire g
test1  T1(e,f,g);
Test1  T2(g,e,d);
endmodule
```

Vivado Tutorial

Simulation module

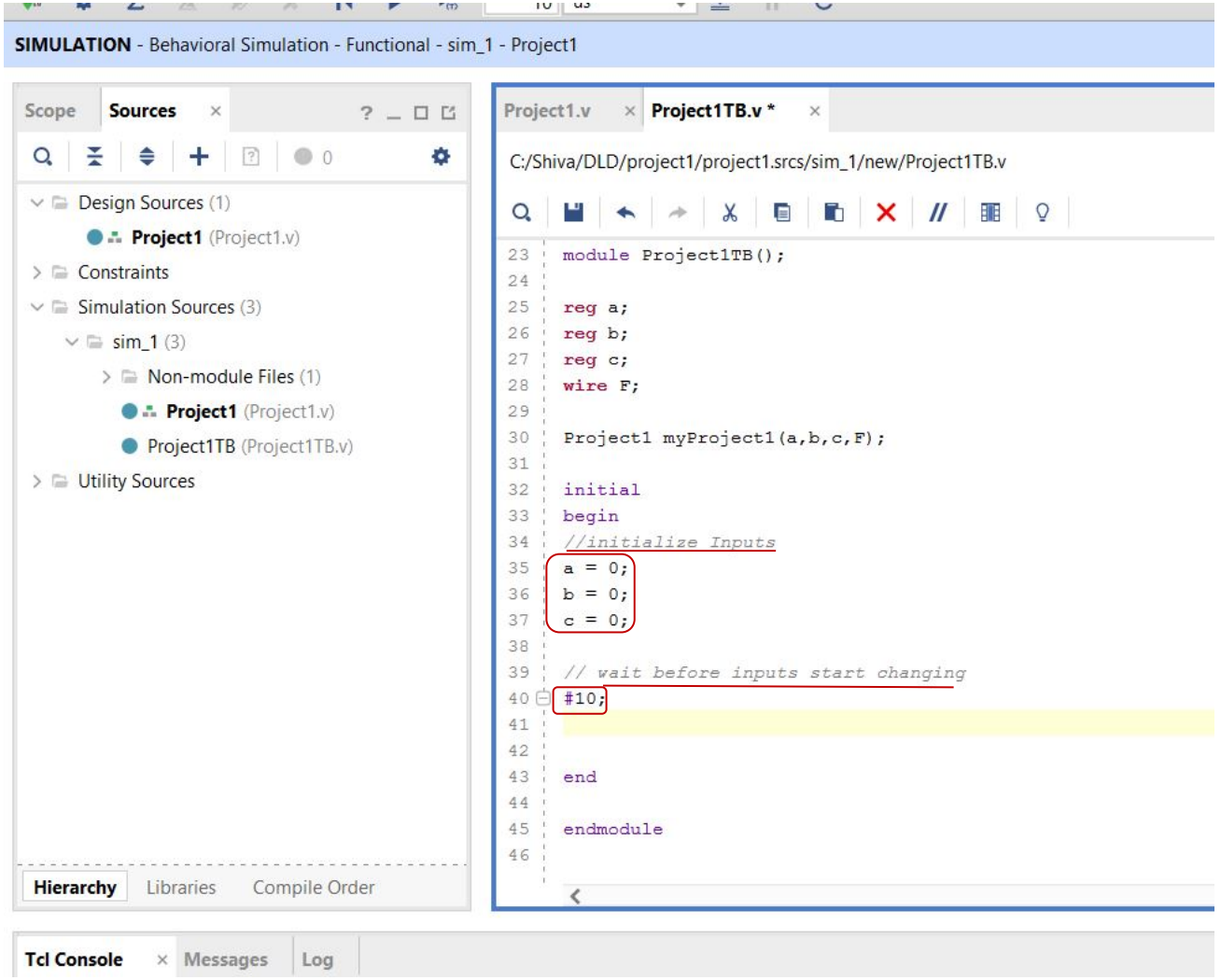
- Start **initial**



Vivado Tutorial

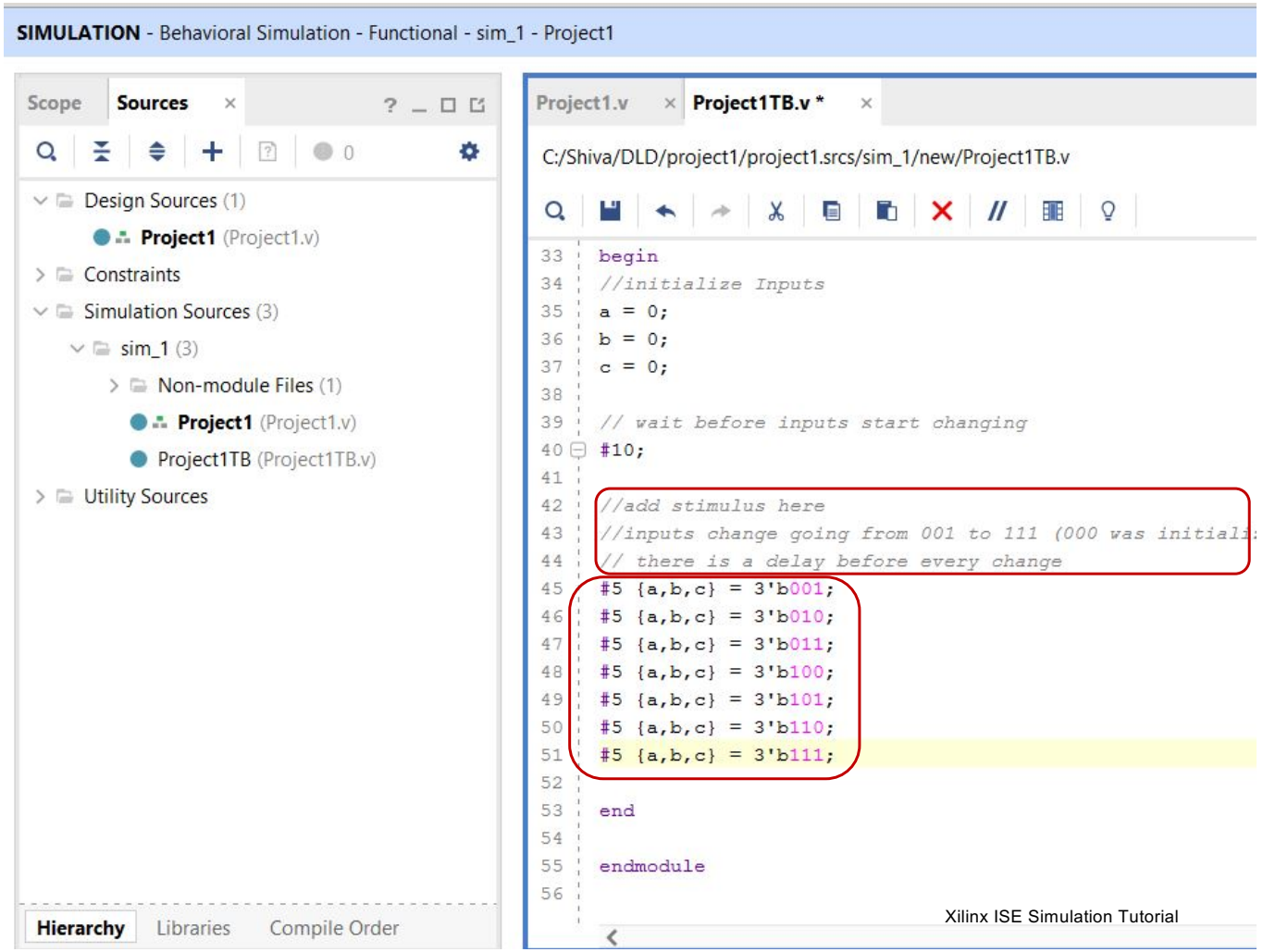
Simulation module

- Assign initial value to inputs.



Vivado Tutorial

Simulation module



Xilinx ISE Simulation Tutorial

Logic System

- Verilog uses a simple, four-valued logic system.
- 1-bit can take on one of only four possible values as shown:

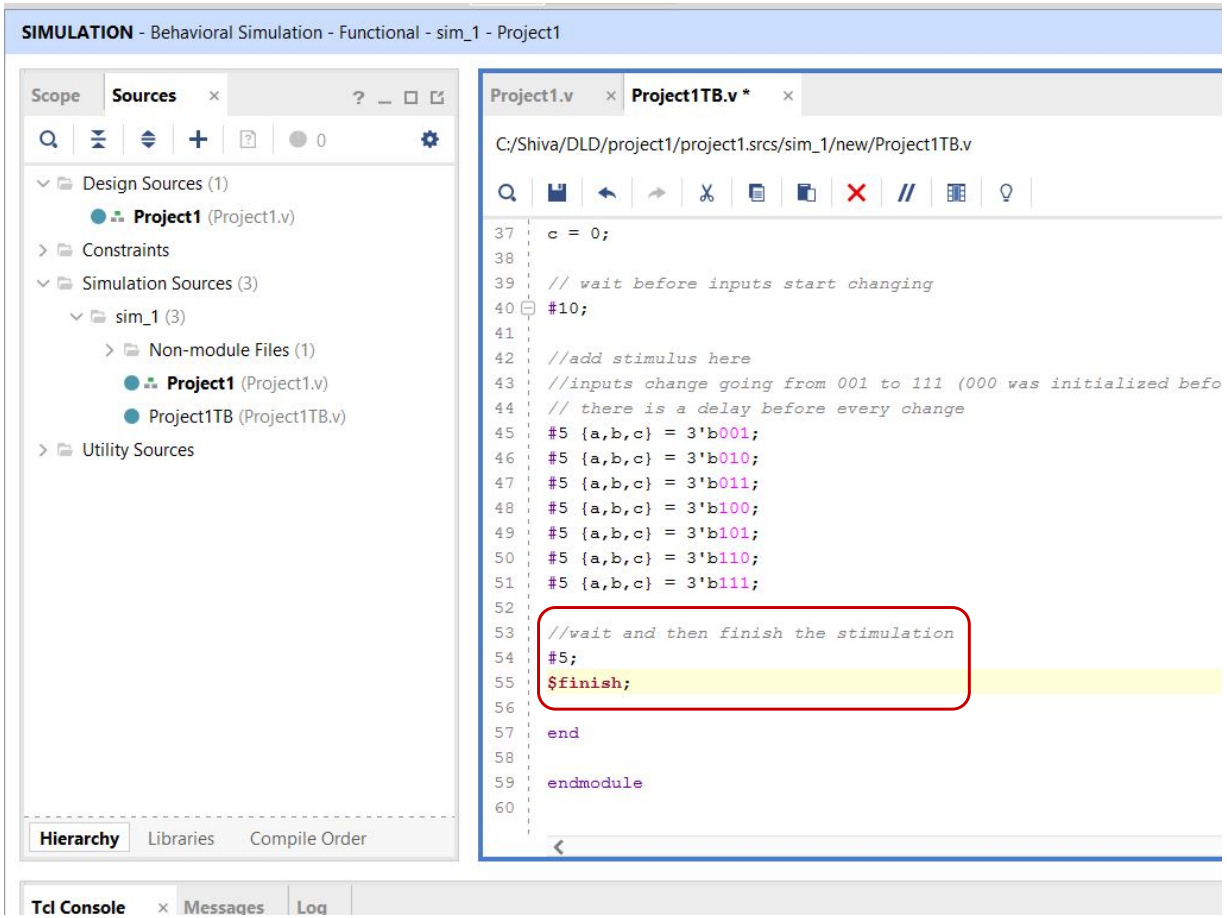
0	Logical 0, or false
1	Logical 1, or true
x	An unknown logical value
z	High impedance (as in three-state logic)

- Verilog uses <size>'<base format><number> to assign values
- 3'b010 = 3-bit binary number

Literal	*Meaning*
`1'b0`	A single 0 bit
`1'b1`	A single 1 bit
`1'bx`	A single unknown bit
`8'b00000000`	An 8-bit vector of all 0 bits
`8'h07`	An 8-bit vector of five 0 and three 1 bits
`8'b111`	The same 8-bit vector (0-padded on left)
`16'hF00D`	A 16-bit vector that makes me hungry
`16'd61453`	The same 16-bit vector, with less hunger
`2'b1011`	Tricky or an error, leftmost "10" ignored
`4'b1?zz`	A 4-bit vector with three high-Z bits
`8'b01x11xx0`	An 8-bit vector with some unknown bits

Vivado Tutorial

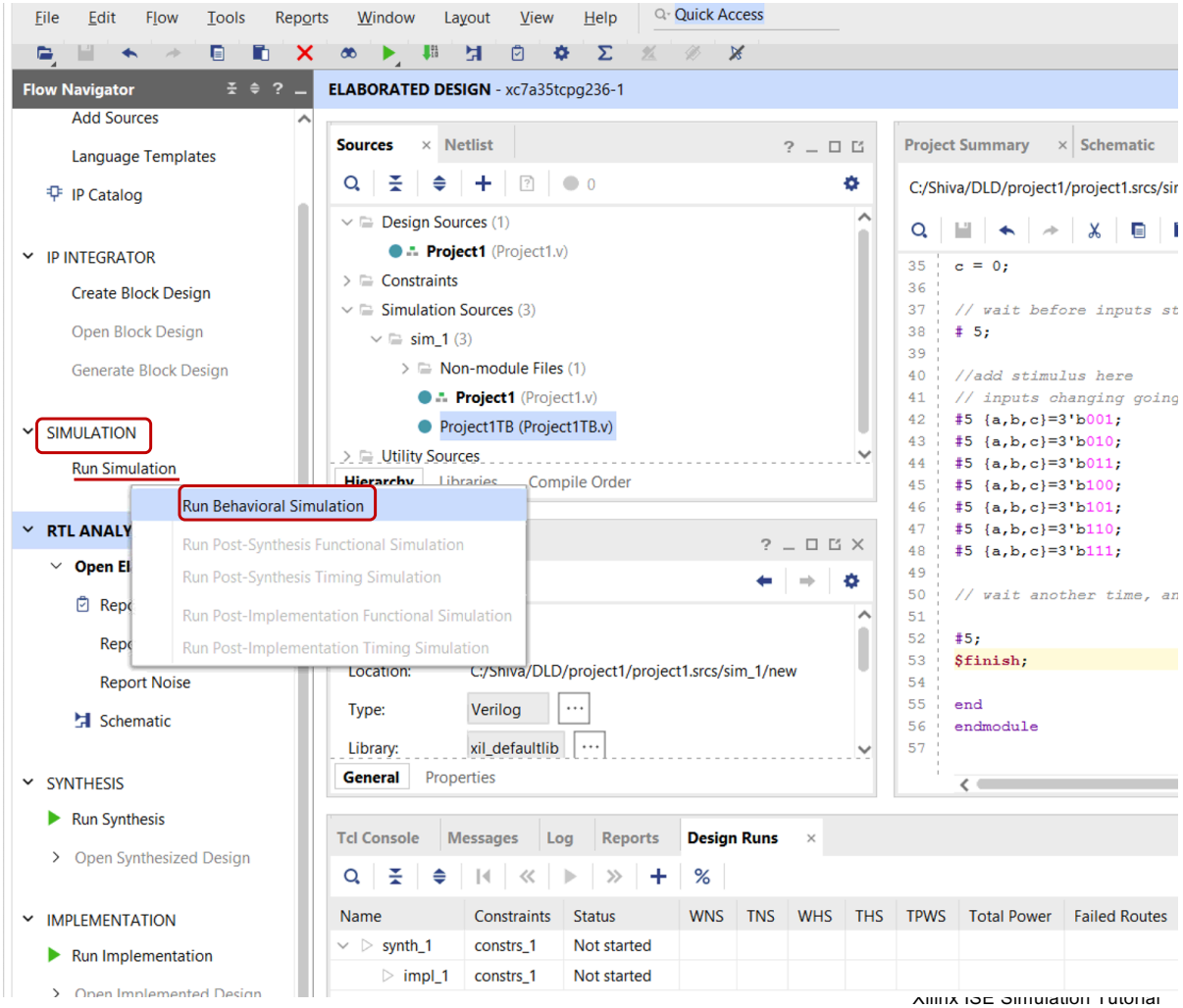
Simulation module



Vivado Tutorial

Simulation module

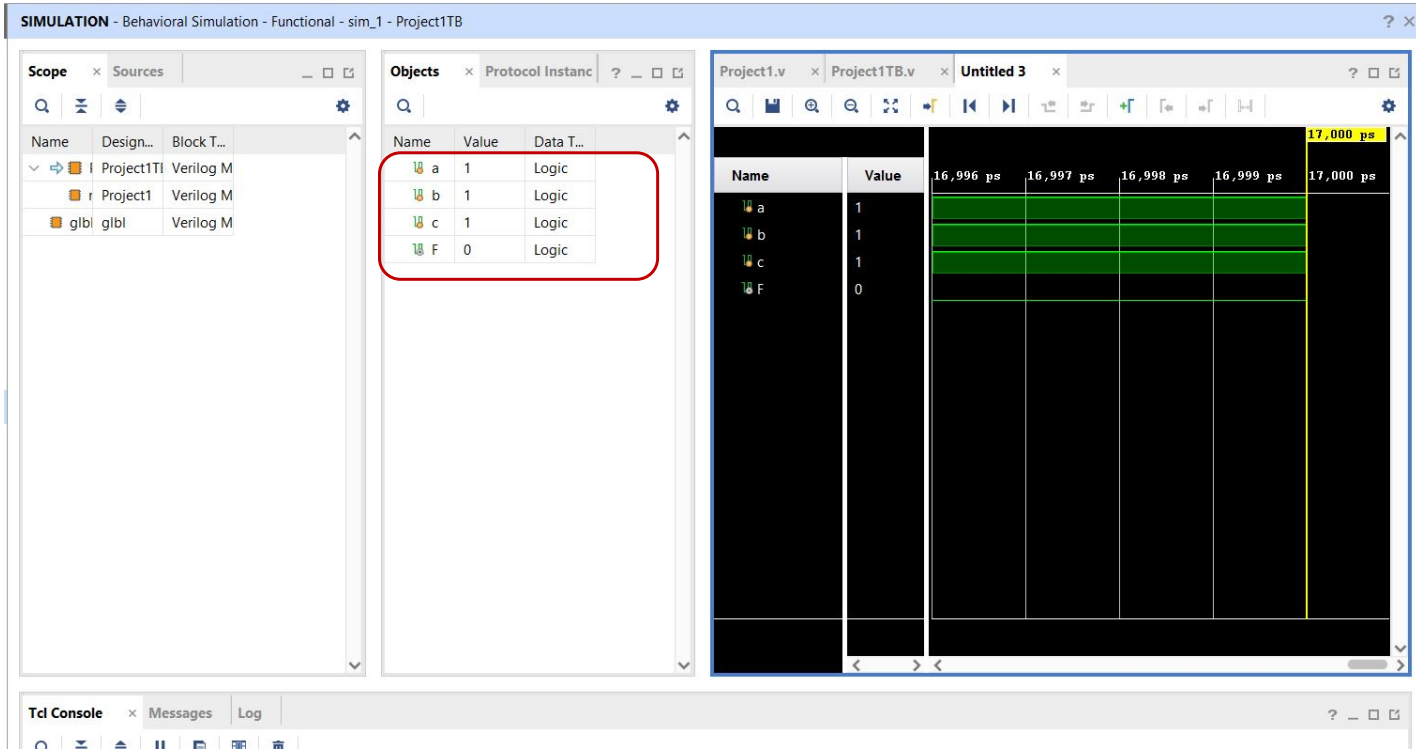
- Run the simulation



Vivado Tutorial

Waveforms

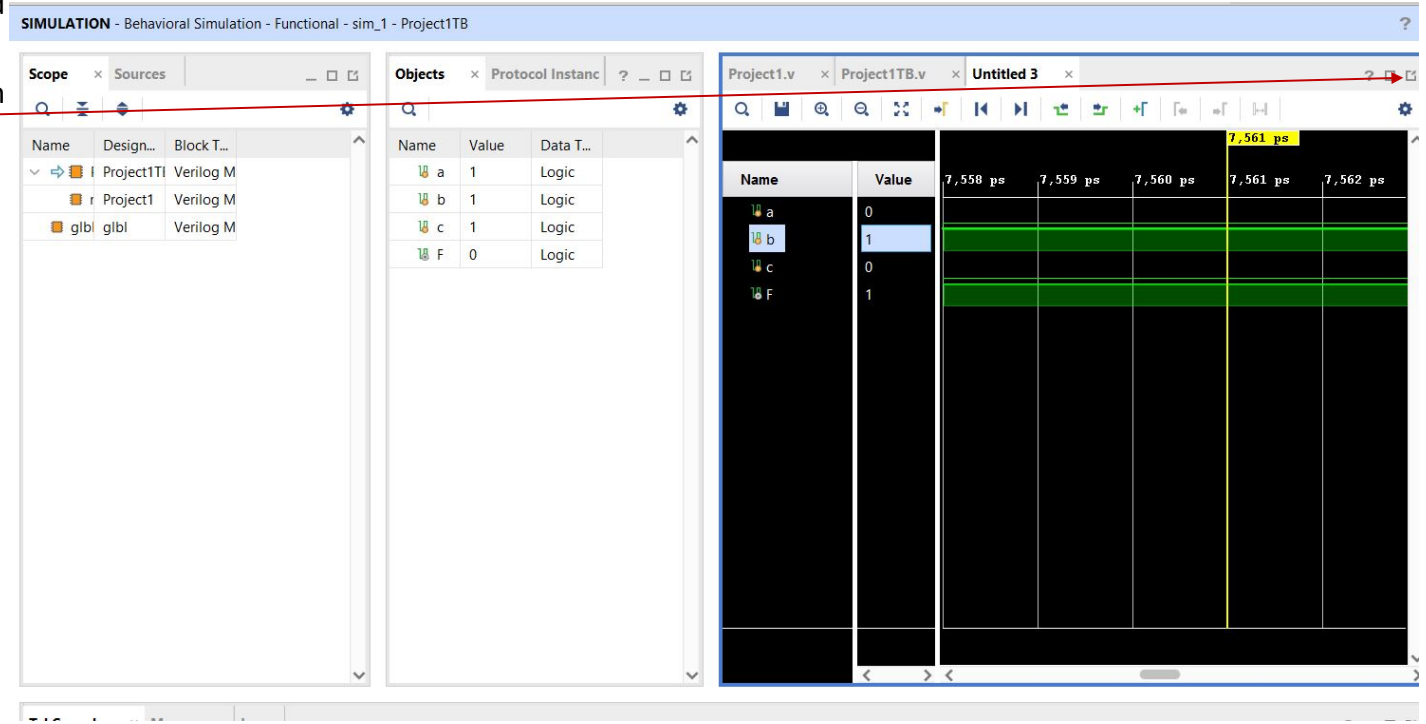
- The waveform dialog box will open.



Vivado Tutorial

waveforms

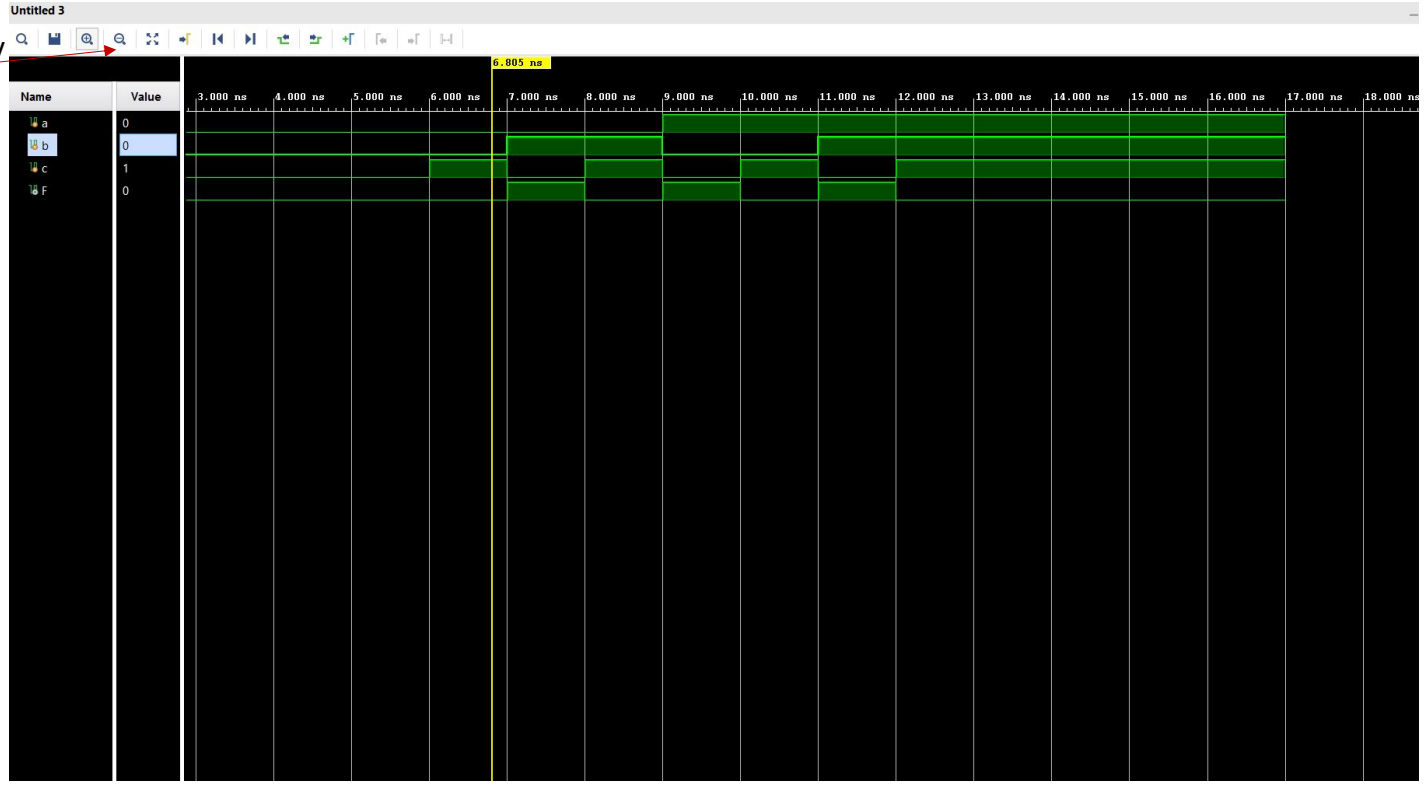
- By moving left and right on the wave form, you can see the different values you defined for the variables.
- You can undock the waveform window.



Vivado Tutorial

working on the project

- Try to see the whole signals by zooming out.

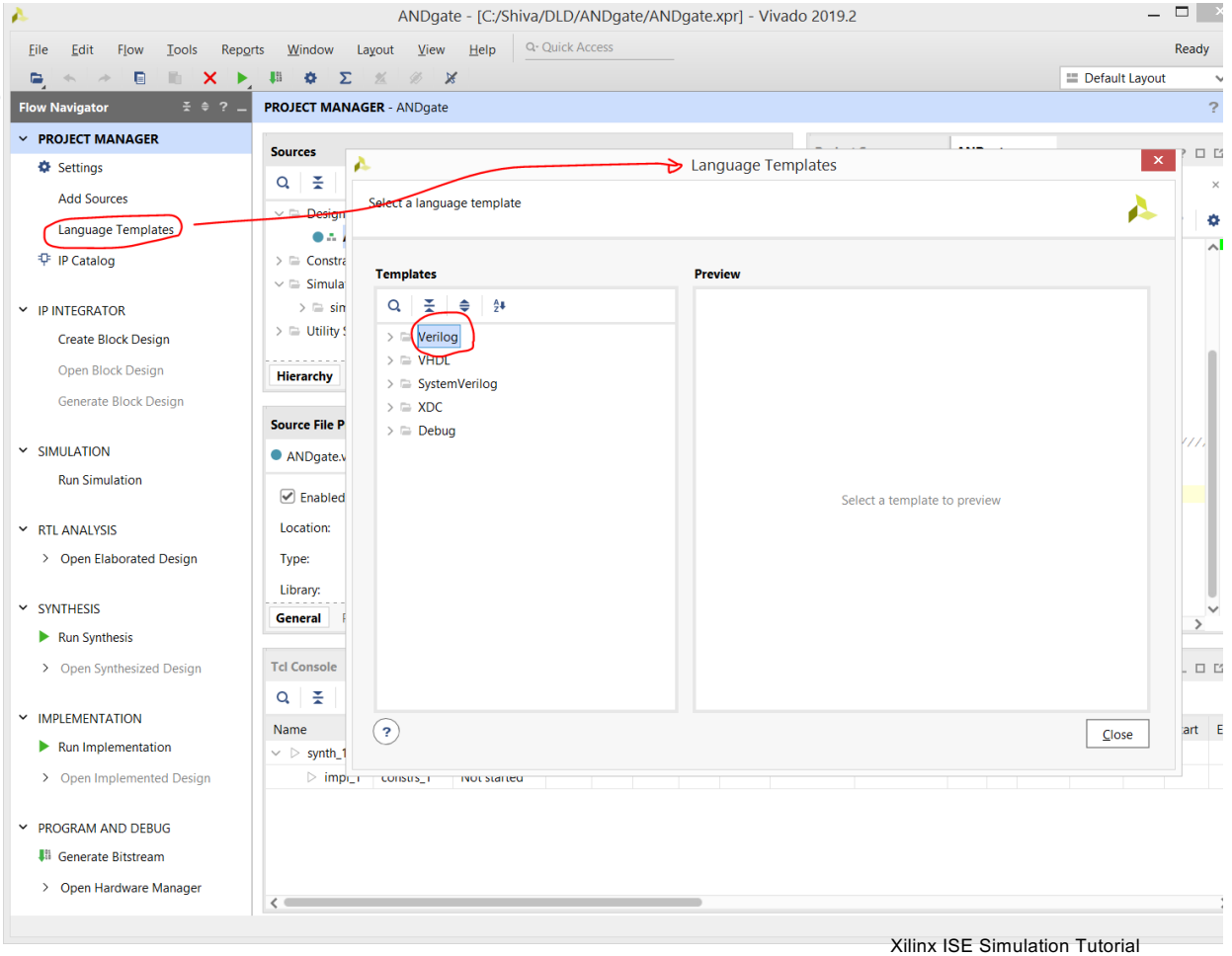


a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Vivado Tutorial

Look up Verilog Template Codes

- You can write code for your statement or you can also look up some template codes form “Language Templates” under Setting .
- Here we will look up the code just to show you how it works.

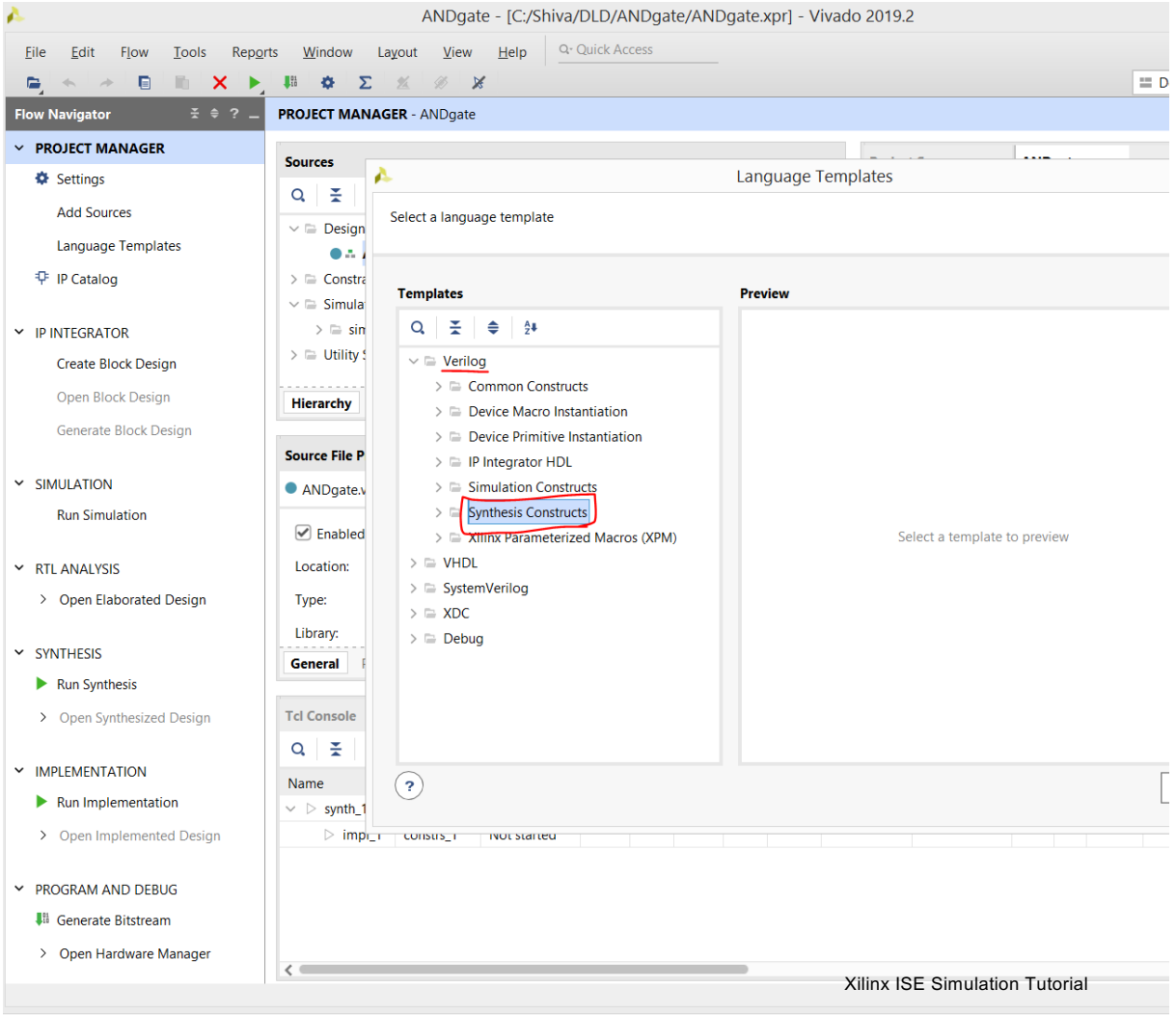


Xilinx ISE Simulation Tutorial

Vivado Tutorial

Look up Verilog Template Codes

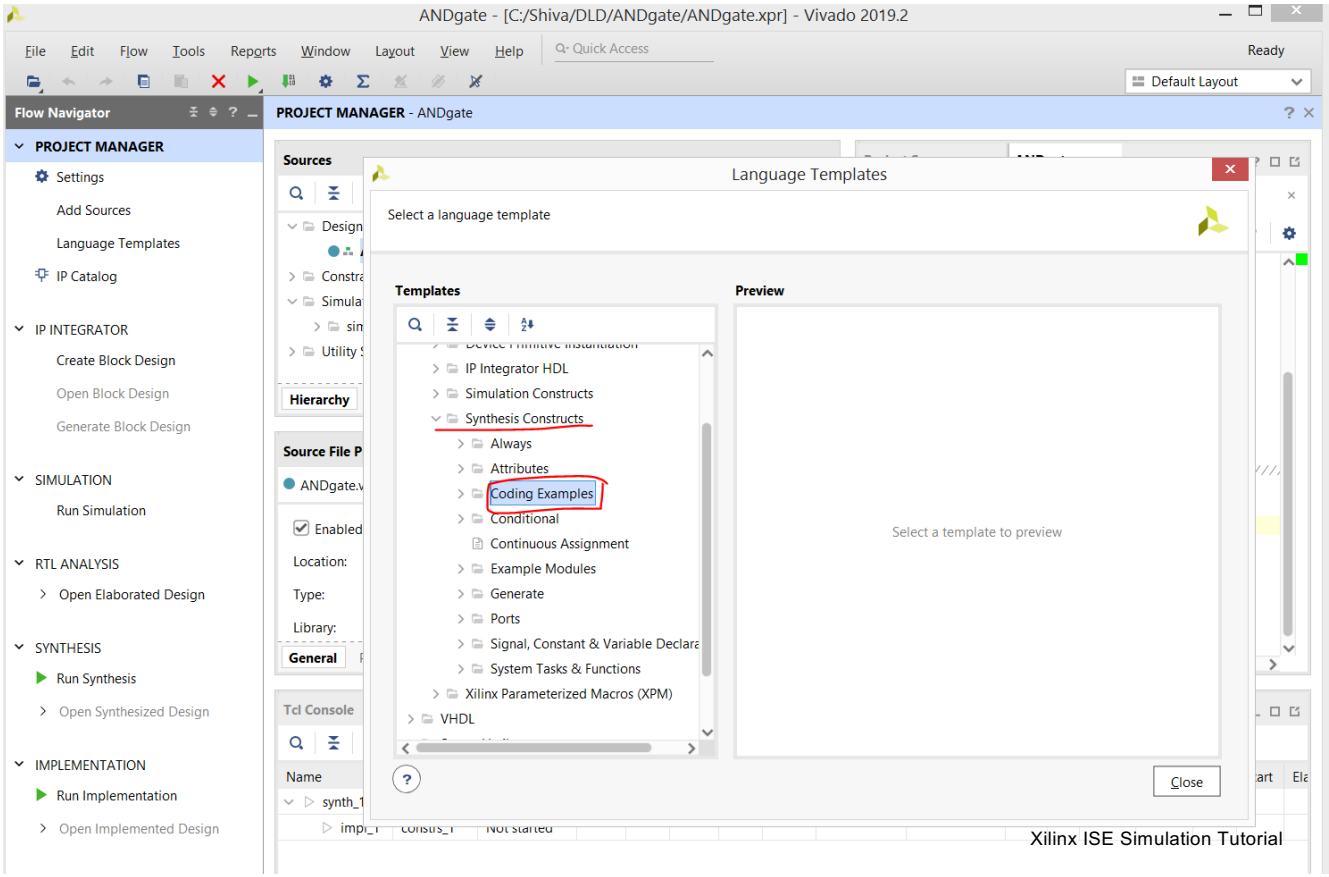
- You can look up some template codes form “Language Templates” under Setting .



Vivado Tutorial

Look up Verilog Template Codes

- Look up the “Coding Examples” from the list.

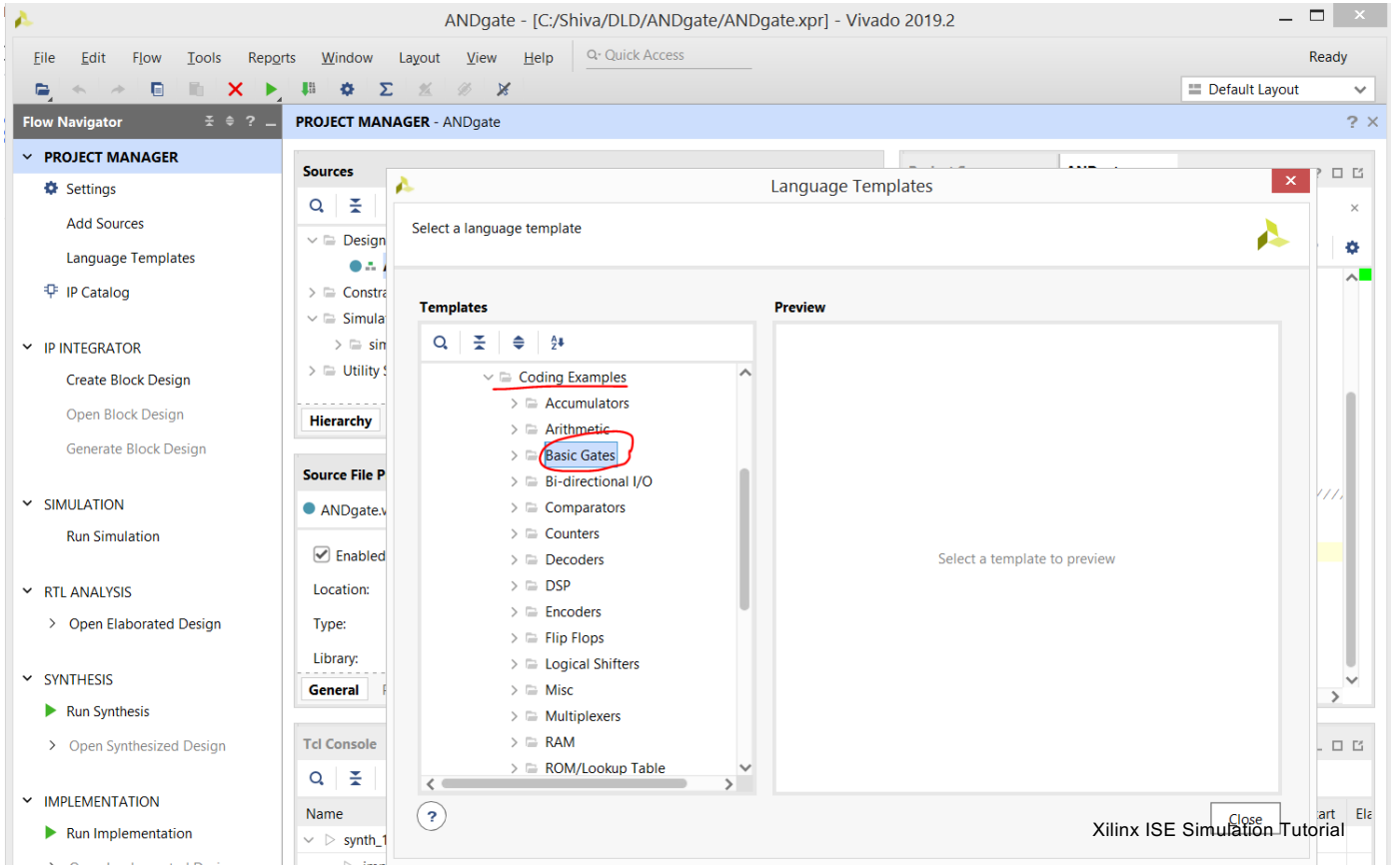


Xilinx ISE Simulation Tutorial

Vivado Tutorial

Look up Verilog Template Codes

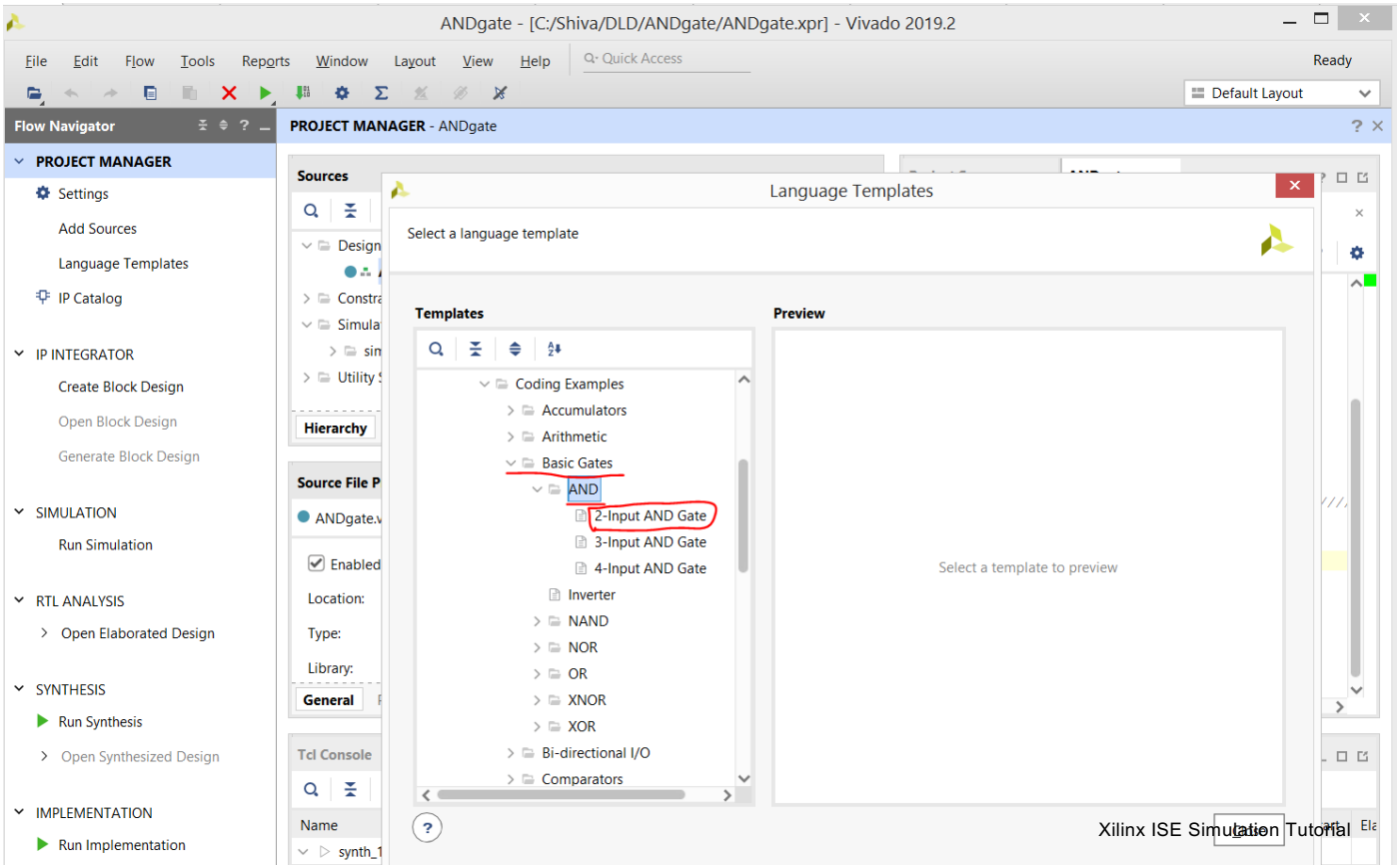
- You can look up the type of coding example that you are looking for from the list.



Vivado Tutorial

Look up Verilog Template Codes

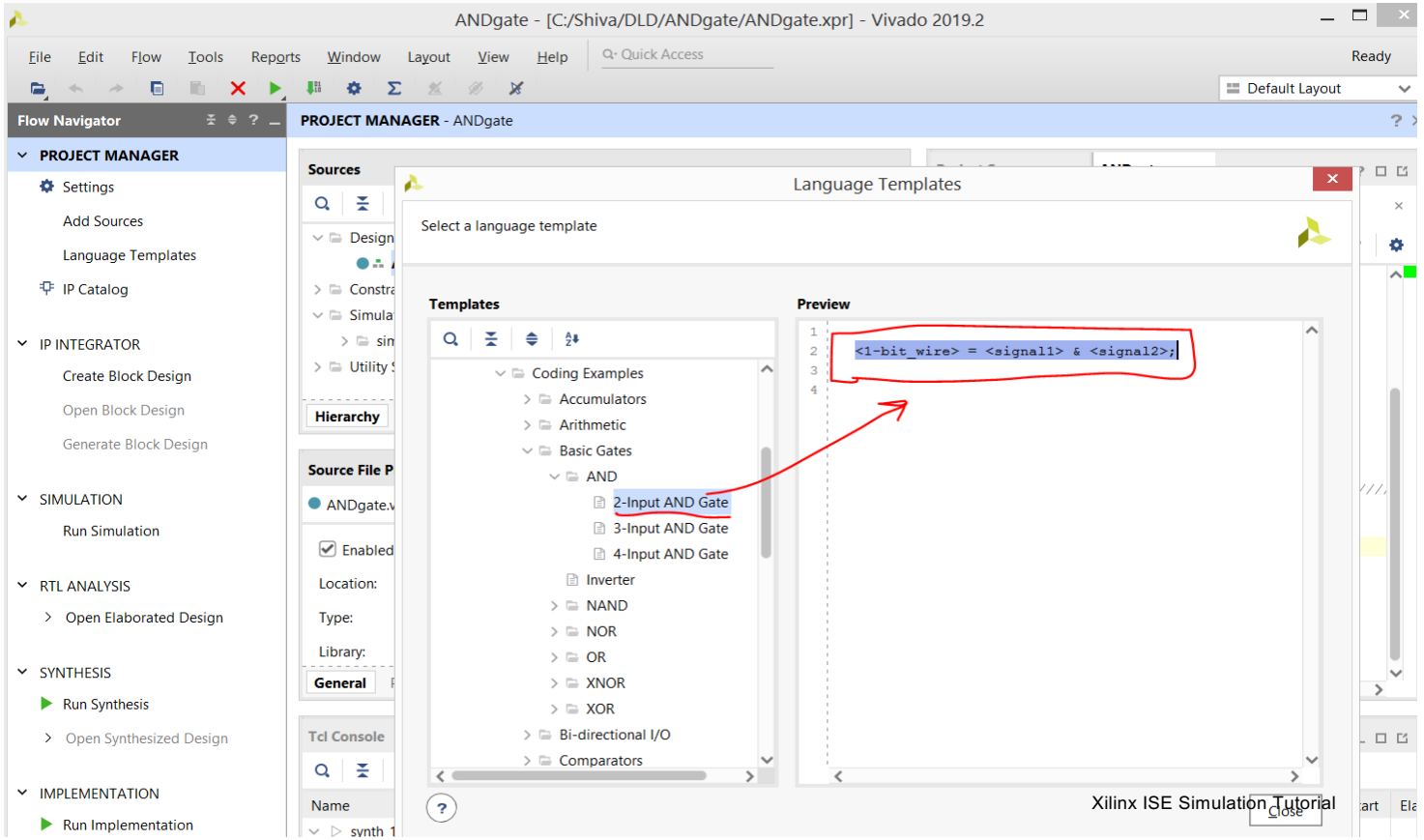
- Here we are looking for 2-input AND gate.



Vivado Tutorial

Look up Verilog Template Codes

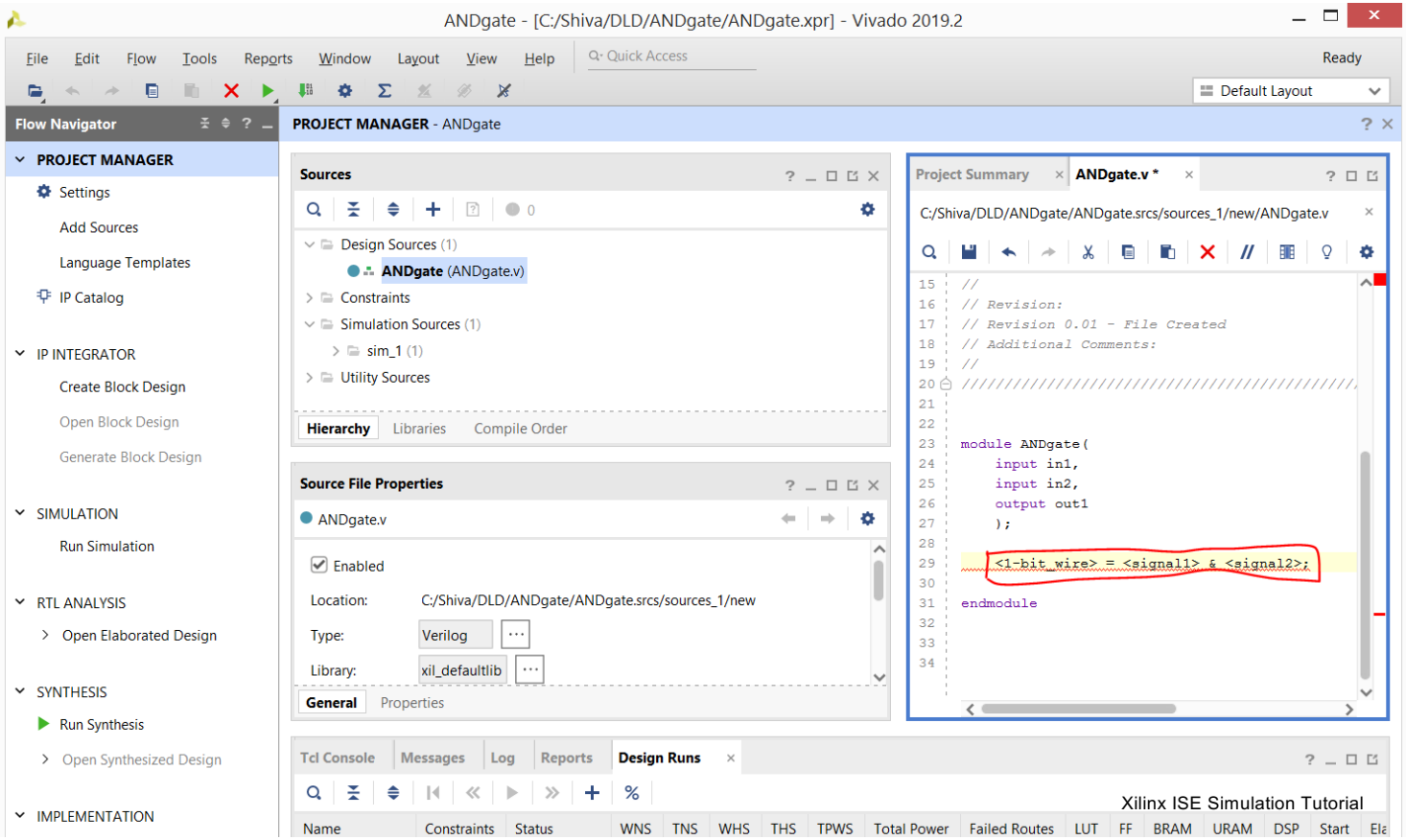
- Copy the template code.



Vivado Tutorial

Look up Verilog Template Codes

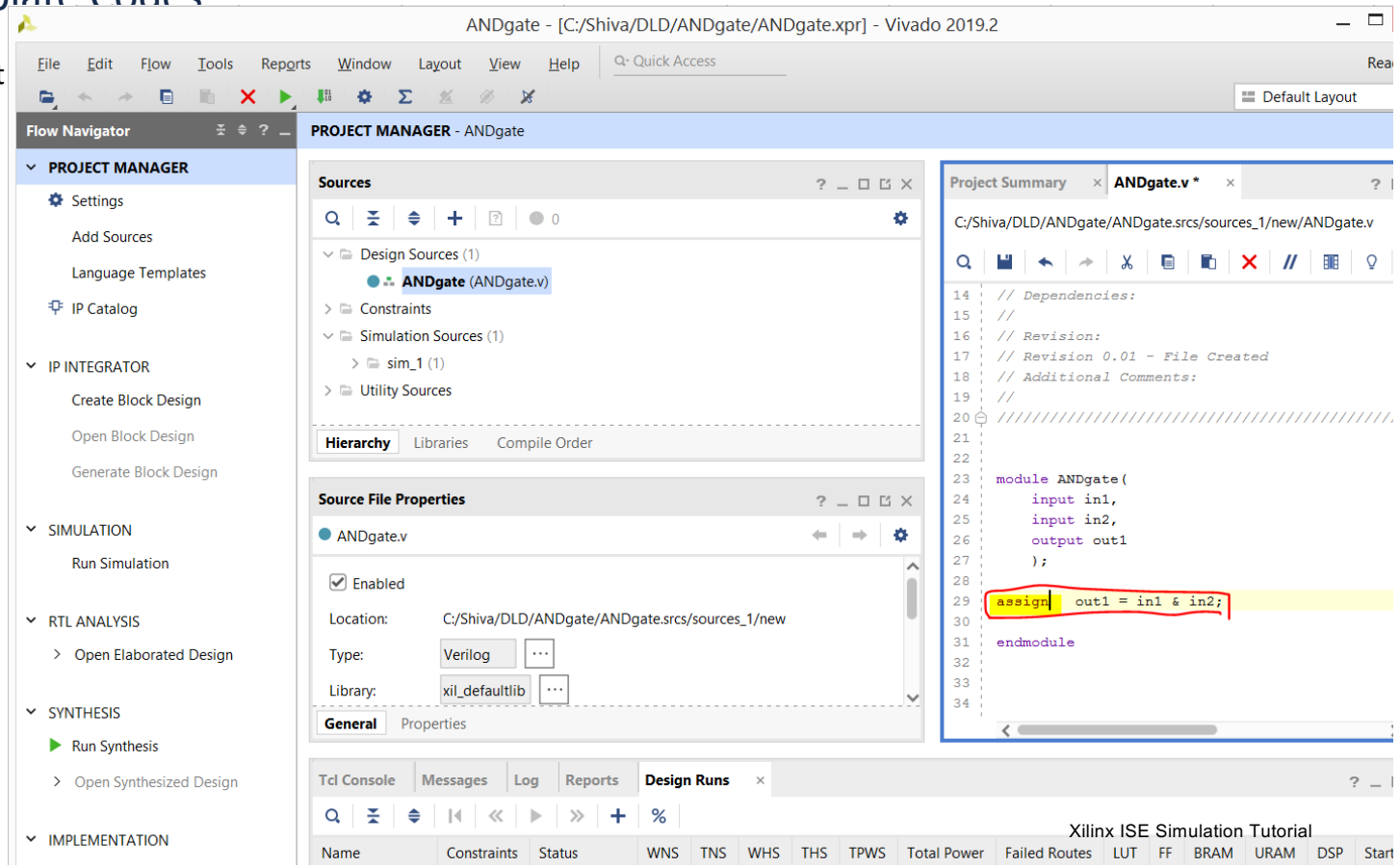
- Paste the template code in your module code body as shown.



Vivado Tutorial

Look up Verilog Template Codes

- Use *assign* keyword and put proper signal names.



Xilinx ISE Simulation Tutorial

what is Verilog HDL

- **Verilog HDL** is IEEE industry standardized Hardware Design Language (**HDL**), used to describe a digital system.
- Using Verilog, you can design, simulate, and synthesize just about any digital circuit, from a handful of combinational logic gates to a complete microprocessor-based system on a chip.
- When you use HDL there are two ways you can model your system:
  - **Behavioral Modeling**: A component is described by its input/output response.

    describes the functionality of the circuit and not the structure of it. A component is described by its input/output response.

    for example, by assigning new values to signals based on tests of logical conditions using *if* and *case* coding.
  - **Structural Modeling**: A component is described by interconnecting lower-level components.

    Both functionality and structure of circuit is defined by calling out specific hardware.
- Register Transfer Level (**RTL**) is a type of behavioral modeling, for the purpose of synthesis.
- Synthesis: Translating HDL (hardware design language) to a circuit and then optimizing the represented circuit
- RTL Synthesis: translating a RTL model of hardware into an optimized technology specific gate level implementation.

Verilog modules and models

- Basic unit of design and programming in Verilog is a ***module***.
- Module is a text file containing declarations and statements.
- It's case sensitive.
- All keywords are lowercases.
- Whitespace is used for readability.
- A Verilog module has declarations that describe:

1. the names and types of the module's inputs and outputs,
2. local signals, variables, constants,
3. Functions that are used strictly internally to the module and are not visible outside.

- The rest of the module contains statements that specify or model the operation of the module's outputs and internal signals.

```
module module_name (port list);

        port declarations;
        ...

        variable declaration;
        ...

        description of behavior

endmodule
```

module function statements

Module name

```
module ButGate
   (A, B, C, D, Y, Z);
input A, B, C, D;
output reg Y, Z;

always @ (A, B, C, D)
  begin
    if ((A==B)&&(C!=D))
      Y = 1;
    else Y = 0;
    if ((C==D)&&(A!=B))
      Z = 1;
    else Z = 0;
  end
endmodule
```

Syntax of a Verilog Module Declaration

- A basic syntax for a Verilog module declaration is shown. In the figure.
- ends with keyword **endmodule**.
- An input/output signal declared as described above is one bit wide. Multiple bit or “vector” signals can be declared by including a range specification, [msb:lsb], in the declaration.
- Here msb and lsb are integers that indicate the starting and ending indexes of the individual bits within a vector of signals.

```
module module-name (port-name, port-name, ..., port-name);
  input declarations
  output declarations
  inout declarations
  net declarations
  variable declarations
  parameter declarations
  function declarations
  task declarations

  concurrent statements
endmodule
```

```
module ButGate
  (A, B, C, D, Y, Z);
input A, B, C, D;
output reg Y, Z;

always @ (A, B, C, D)
  begin
    if ((A==B)&&(C!=D))
      Y = 1;
    else Y = 0;
    if ((C==D)&&(A!=B))
      Z = 1;
    else Z = 0;
  end
endmodule
```

Verilog modules and models

- Module Declaration:
  - Each module starts with keyword module
  - It ends with keyword endmodule
  - Provides module name
  - Declares port list
- Port Types:
  - Input
  - Output
  - Inout : bidirectional port
- Port declaration:
  - <port type> <portname>;

  Example:

  input in1, in2;

  or

  input in1;

  input in2;

  output out1, out2;

```
module ButGate
  (A, B, C, D, Y, Z);
input A, B, C, D;
output reg Y, Z;

always @ (A, B, C, D)
  begin
    if ((A==B)&&(C!=D))
      Y = 1;
    else Y = 0;
    if ((C==D)&&(A!=B))
      Z = 1;
    else Z = 0;
  end
endmodule
```

```
input  identifier, identifier, ..., identifier;
output identifier, identifier, ..., identifier;
inout  identifier, identifier, ..., identifier;

input  [msb:lsb] identifier, identifier, ..., identifier;
output [msb:lsb] identifier, identifier, ..., identifier;
inout  [msb:lsb] identifier, identifier, ..., identifier;
```

Verilog Data Types

- Net Data:
  - Represent the physical interconnection between structures. It represent the activity flow between the functional blocks.
- Variable Data:
  - Represent elements to store the data. They are store temporarily and just internally within the module.

```
input  identifier, identifier, ..., identifier;
output identifier, identifier, ..., identifier;
inout  identifier, identifier, ..., identifier;

input  [msb:lsb] identifier, identifier, ..., identifier;
output [msb:lsb] identifier, identifier, ..., identifier;
inout  [msb:lsb] identifier, identifier, ..., identifier;
```

```
wire identifier, identifier, ..., identifier;
wire [msb:lsb] identifier, identifier, ..., identifier;

tri identifier, identifier, ..., identifier;
tri [msb:lsb] identifier, identifier, ..., identifier;
```

```
reg [7:0] byte1, byte2, byte3;
reg [15:0] word1, word2;
reg [1:16] Zbus;
```

Net Data Type:

- wire: represent a node or connection
- tri: represent a tri-state node
- supply 0: logic 0
  supply 1: logic 1

Variable Data Type:

- reg: unsigned variable
- reg signed: signed variable
- integer: signed 32-bit variable
- time, real, realtime

Bus Declaration:

- <data type> [MSB: LSB] <signal name>
  Or
- <data type> [LSB:MSB] <signal name>

Example:
reg [7:0] out; //a 8-bit register
wire [7:0] out;

Parameter Declaration

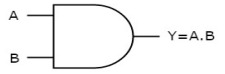
- Parameter:
  - It's a value assigned to a name
  - Can be overwritten

```
parameter identifier = value;
parameter identifier = value,
          identifier = value,
          ...
          identifier = value;
```

```
parameter size = 8;
parameter  length = 3;
```

- Assignment statement:
  - Specity output signals in terms of input signals
  - Can be overwritten

```
assign Y = A &B ;
```

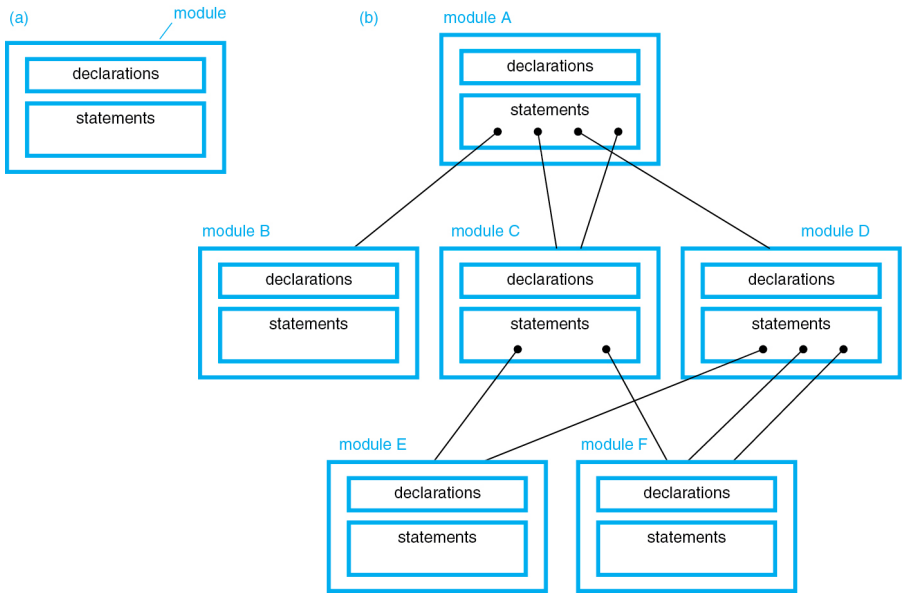



```
assign net-name = expression;
assign net-name[bit-index] = expression;
assign net-name[msb:lsb] = expression;
assign net-concatenation = expression;
```


Pearson

modules instantiating other modules

- Verilog modules can use a mix of behavioral and structural models, and may do so hieratically as shown in the figure.
- A higher level module may use lower level module multiple times, and multiple top level modules may use the same lower level one.
- The value can be passed between the modules **only by using declared input and output signals.**



modules instantiating other modules

- Here we instantiate test1 twice in test2.
- We call the module of lower-level component that we instantiate
- Indicate it with a unique instance_name
- List the ports

<module_name> <instance_name> (port_list);

```
module test1 (a,b,c);
        input a,b;
        output c;
….
….
….
endmodule
```

```
module test2(d,e,f);
        input d,e;
        output f;
….
wire g
test1  T1(e,f,g);
Test1  T2(g,e,d);
endmodule
```

Logic System

- Verilog uses a simple, four-valued logic system.
- 1-bit can take on one of only four possible values as shown:

0	Logical 0, or false
1	Logical 1, or true
x	An unknown logical value
z	High impedance (as in three-state logic)

- Verilog uses <size>'<base format><number> to assign values
- 3'b010 = 3-bit binary number

Literal	*Meaning*
`1'b0`	A single 0 bit
`1'b1`	A single 1 bit
`1'bx`	A single unknown bit
`8'b00000000`	An 8-bit vector of all 0 bits
`8'h07`	An 8-bit vector of five 0 and three 1 bits
`8'b111`	The same 8-bit vector (0-padded on left)
`16'hF00D`	A 16-bit vector that makes me hungry
`16'd61453`	The same 16-bit vector, with less hunger
`2'b1011`	Tricky or an error, leftmost "10" ignored
`4'b1?zz`	A 4-bit vector with three high-Z bits
`8'b01x11xx0`	An 8-bit vector with some unknown bits

Bitwise Boolean Operators

- Operates on each bit or bit pairs

Operator	*Operation*
&	AND
|	OR
^	Exclusive OR (XOR)
~^, ^~	Exclusive NOR (XNOR)
~	NOT

Verilog Logical Operators

- True and false values can be created and combined in expressions by the logical operators shown in the table.
- A logical operation yields a value depending on whether the result is true or false (‘0’ or ‘1’).

Operator	*Operation*
&&	logical AND
||	logical OR
!	logical NOT
==	logical equality
!=	logical inequality
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

Arithmetic and Shift Operators in Verilog

- Verilog also has explicit Shift operations for vectors as shown in the table.
- These are sometimes called logical shift operators.
- Results unknown if any operand is Z or X
- Treats vectors as a whole value

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)
**	Exponentiation
<<	(Logical) shift left
>>	(Logical) shift right
<<<	Arithmetic shift left
>>>	Arithmetic shift right

Verilog Built-In Gates

and	xor	bufif0
nand	xnor	bufif1
or	buf	notif0
nor	not	notif1

Example:

```
out = and(a,b);
and and1(o1,b1,a1);
and and2(o2,b2,a2);
```

Verilog "Case Equality" Operators

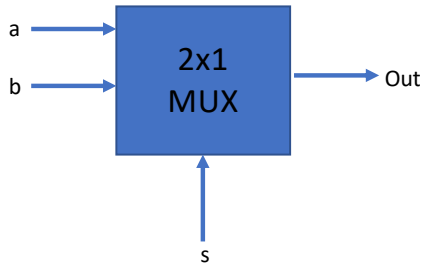
- Used to compare values
- Returns a one bit scalar value of Boolean true or false ('0' or '1')

Operator	*Operation*
===	case equality
!==	case inequality

Syntax of Verilog If Statements

```
if ( condition ) procedural-statement

if ( condition ) procedural-statement
else procedural-statement
```

```
If (s==0) begin
Out = b;
end
else begin
Out =a;
end
```



Syntax of Begin-End Blocks

```
begin
   procedural-statement
   ...
   procedural-statement
end

begin : block-name
   variable declarations
   parameter declarations

   procedural-statement
   ...
   procedural-statement
end
```

Syntax of initial blocks

- initial statement executes only once
- Used to initialize behavioral statements for simulation
- If clock contains more than one statement it should contain keywords begin and end

```
initial
  procedural-statement

initial begin
  procedural-statement
  ...
  procedural-statement
end
```

```
initial begin
sum = 0;
carry = 0;
end
```

Syntax of always block

- always statement executes in a loop
- Used to describe the functionality of behavioral statements
- If clock contains more than one statement it should contain keywords begin and end

```
always  @ (A or B) begin
sum = A^B;
carry = A &b;
end
```

Blocking vs. Noblocking Assignments

- Nonblocking operator (<=) is used for sequential logic
- Blocking operator is used for combinational logic

```
variable-name = expression ;    // blocking assignment
variable-name <= expression ;   // nonblocking assignment
```

<delay>
Delay through component

a=1 and b=2

```
initial
  begin
    a = #5 b;
    c = #10 a;
  end
```

@ 5 a = b = 2
@15 c=a=2

```
initial
  begin
    a <= #5 b;
    c <= #10 a;
  end
```

@ 5 a = b = 2
@10 c=a=1