



Sukkur Institute of Business Administration University

Department of Electrical Engineering

ESE-412: Digital System Design Lab

Handout#01: Vivado Design Flow, Verilog Gate Level & Hierarchical Modelling

Instructor: Dr. Safeer Hyder

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Learning Objectives:

After completing this session, student should be able to:

- ◆ Apply Vivado Design Flow and test a simple circuit using Nexys 4 DDR
- ◆ Identify logic gate primitives provided in the Verilog HDL
- ◆ Implement basic combinational circuits using gate-level modelling

Lab Hardware and Software Required:

1. Xilinx Vivado 2016.2
2. Digilent NEXYS 4 DDR FPGA Board
3. Desktop/Laptop Computer

Introduction:

This lab is divided into three parts, each part have its own goals to achieve and requirements to fulfill. In Part 1, the fundamentals of hardware developments board and software that will be used in this semester will be introduced to give you insights before your first use. In Part 2, you will basics of Verilog HDL, learn about the built-in gate-level logical modules, and move towards creating your projects using gate level modelling.

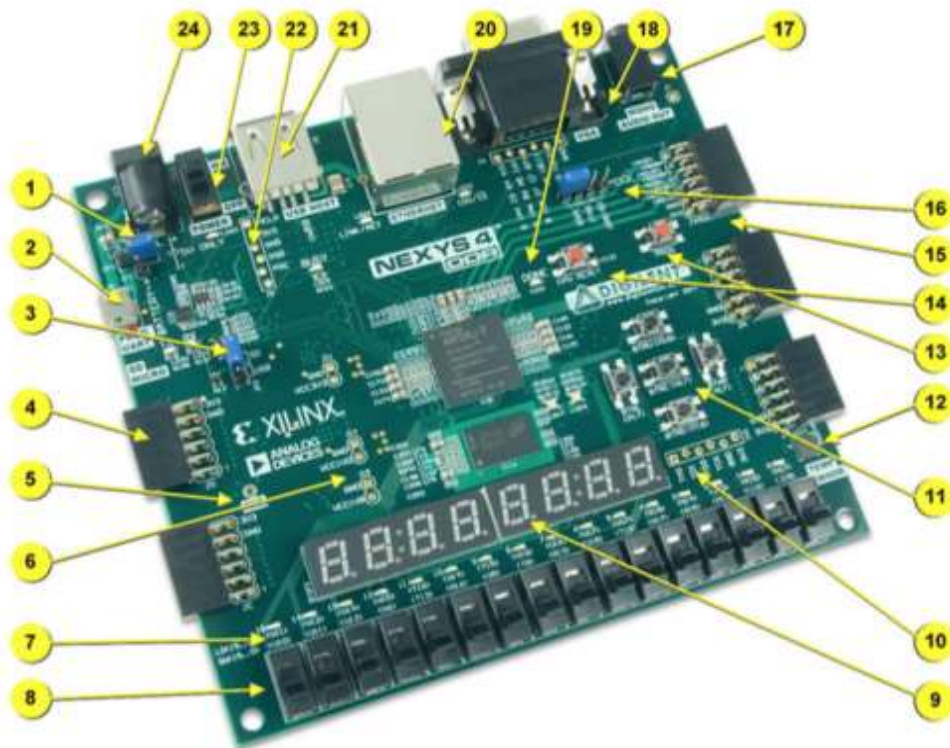
Verilog HDL modeling language supports three kinds of modeling styles: gate-level, dataflow, and behavioral. The gate-level and dataflow modeling are used to model combinational circuits whereas the behavioral modeling is used for both combinational and sequential circuits. This lab illustrates the use of all three types of modeling by creating simple combinational circuits targeting Basys3 and Nexys4 DDR boards and using the Vivado 2016.1 software tool. Please refer to the Vivado tutorial on how to use the Vivado tool for creating projects and verifying digital circuits.

Verilog is both a structural and behavioral language. Internals of each module can be defined at four levels of abstraction, depending on the need of the design. There are four levels of abstraction which include switch-level, gate-level, data flow, and behavioral or algorithm level. The switch level is the lowest abstraction level, where module can be implemented in terms of switches, storage nodes, and interconnections between them.

Part 1: Vivado Design Flow

In this part, you are going to learn fundamentals of development board and Vivado design Suite; from creating a project to implementing and running it on hardware. In this lab, the board being used is Nexys4 DDR. This board is shown below along with components names, and has following features:

- 128 MiB DDR 2 SDRAM
- 16 MB SPI (quad mode) PCM non-volatile memory
- 16 MB parallel PCM non-volatile memory
- 10/100 Ethernet PHYS
- USB-UART and USB-HID port (for mouse/keyboard)
- 8-bit VGA port
- 100MHz CMOS oscillator
- 72 I/O's routed to expansion connectors
- GPIO includes 8 LEDs, 5 buttons, 8 slide switches and 4-digit seven-segment display



Callout	Component Description	Callout	Component Description
1	Power select jumper and battery header	13	FPGA configuration reset button
2	Shared UART/ JTAG USB port	14	CPU reset button (for soft cores)
3	External configuration jumper (SD / USB)	15	Analog signal Pmod connector (XADC)
4	Pmod connector(s)	16	Programming mode jumper
5	Microphone	17	Audio connector
6	Power supply test point(s)	18	VGA connector
7	LEDs (16)	19	FPGA programming done LED
8	Slide switches	20	Ethernet connector

Figure 1: Nexys4 DDR development board

Vivado design flow, gate level & hierarchical modelling

The design flow chart for HDL-based modelling is given below in figure 2, this flow chart shows the simplified flow of HDL modelling, verification and synthesis. This will help to simply understand the process of HDL development.

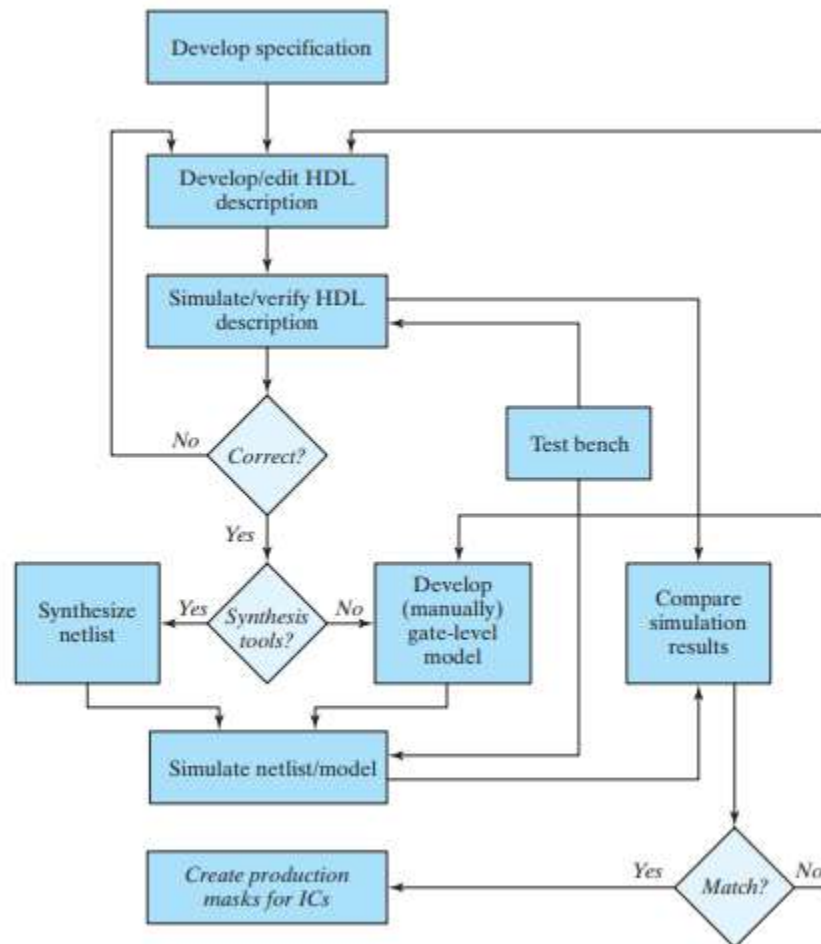


Figure 2: Flow-chart for HDL based modelling, verification & Synthesis

Vivado Design Suite Guide

Nexys4 DDR

After successful installation of your Vivado design suite, you will see icons of Vivado on your home screen, but you need to downloading and placement of Nexys4 DDR board files from internet. Proper installation and placement of board files is required to do this lab.

After completing above requirements, follow below given instructions to create a project:

1. Click on this icon and run Vivado 2016.2



Figure 3: Vivado

2. You will see a popup window, click on [Create New Project]



Figure 4: Vivado welcome screen

Vivado design flow, gate level & hierarchical modelling

3. In the New project popup window, click on [Next]

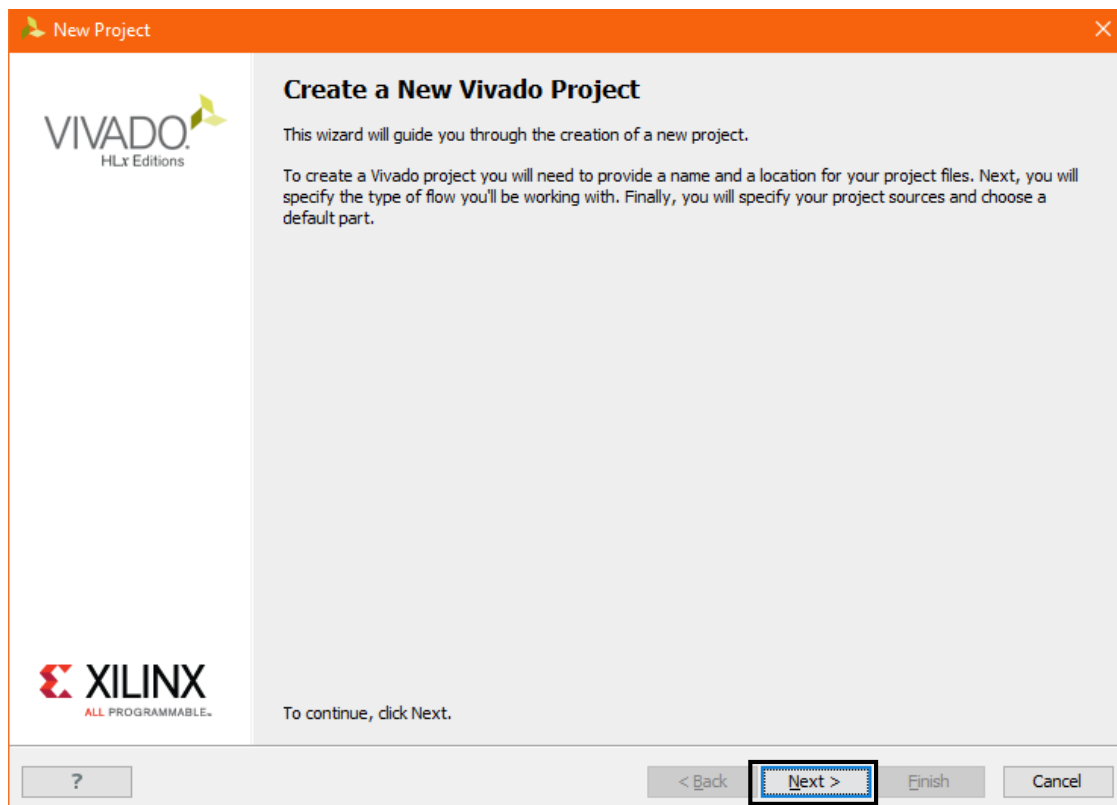


Figure 5: New project popup window

4. Write name of the project (for this example we write *Inverter*), select project location, and click [Next]

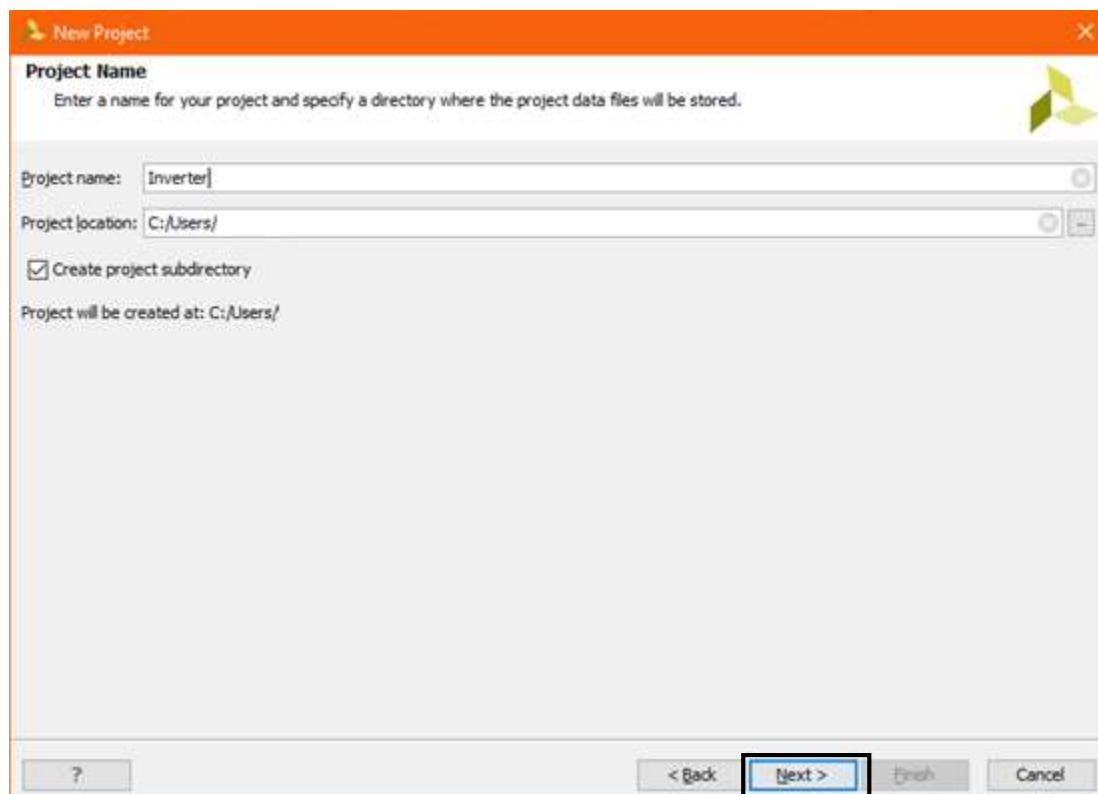


Figure 6: New project naming popup

Vivado design flow, gate level & hierarchical modelling

5. In next popup window, select [RTL Project], check '*Do not specify sources at this time*', and then click [Next].

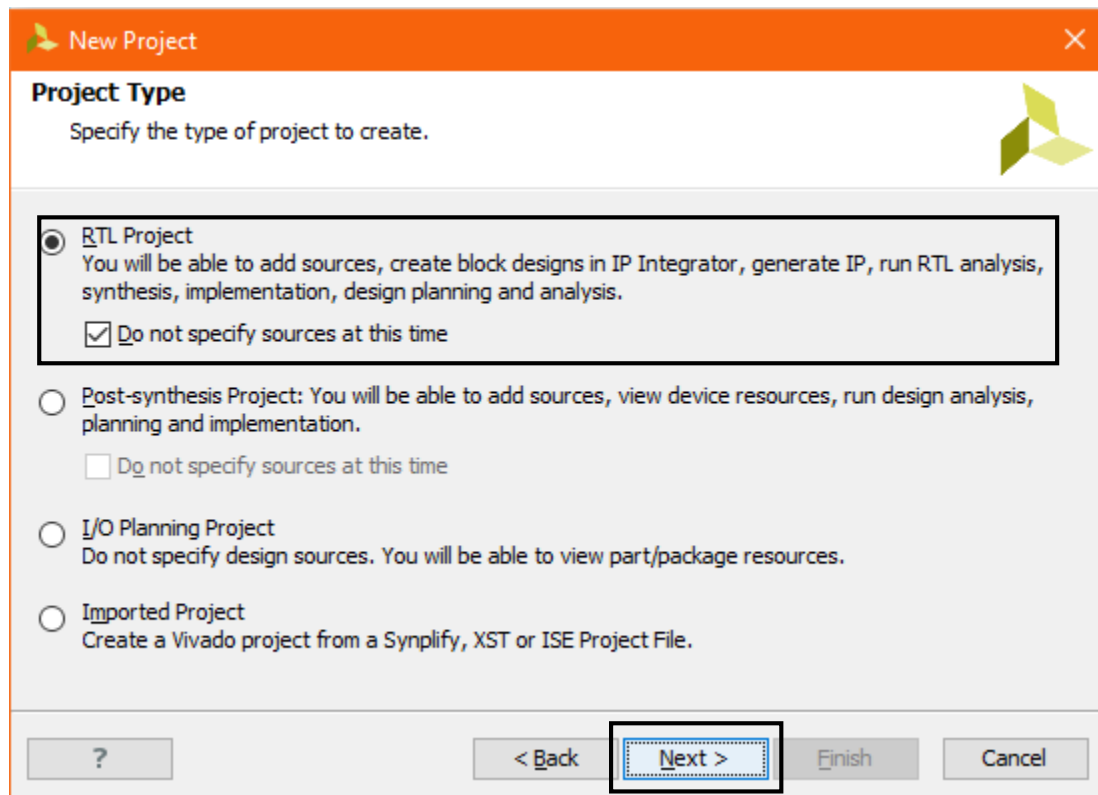


Figure 7: New project type selection

6. In next window select the board, from [boards].

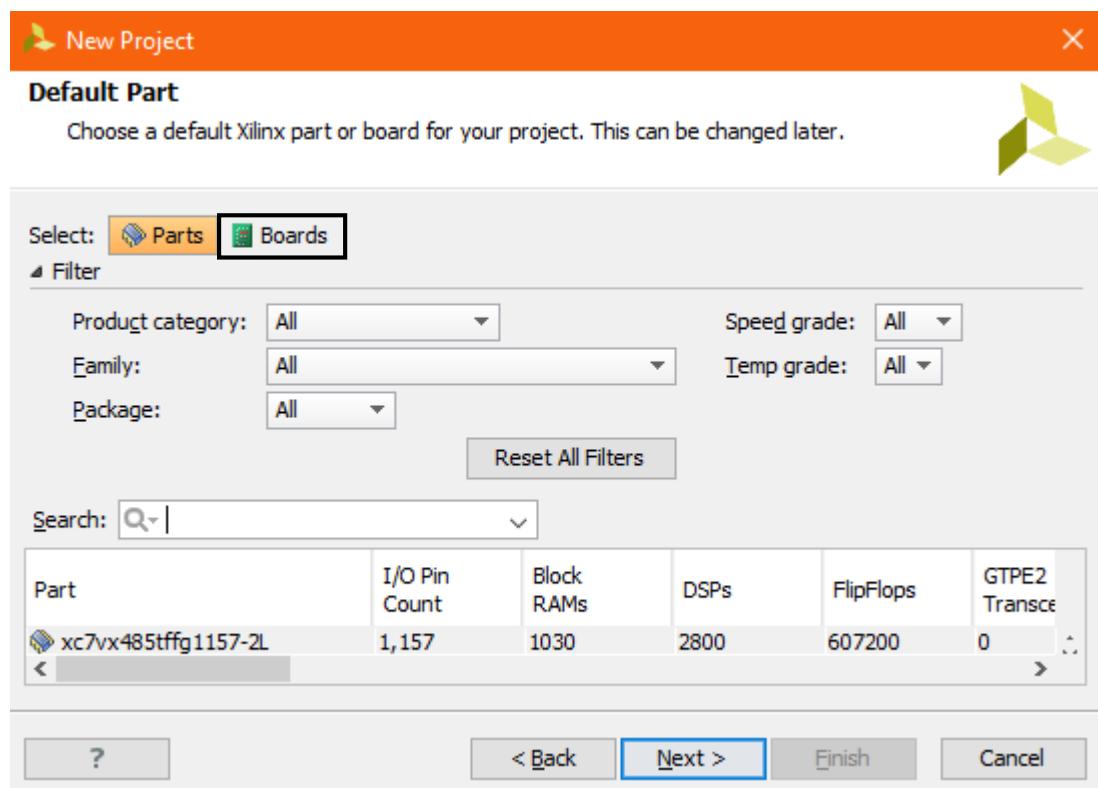


Figure 8: New project board selection

Vivado design flow, gate level & hierarchical modelling

7. From 'vendor' select 'digilentinc.com', then select Nexys4 DDR and click [Next].

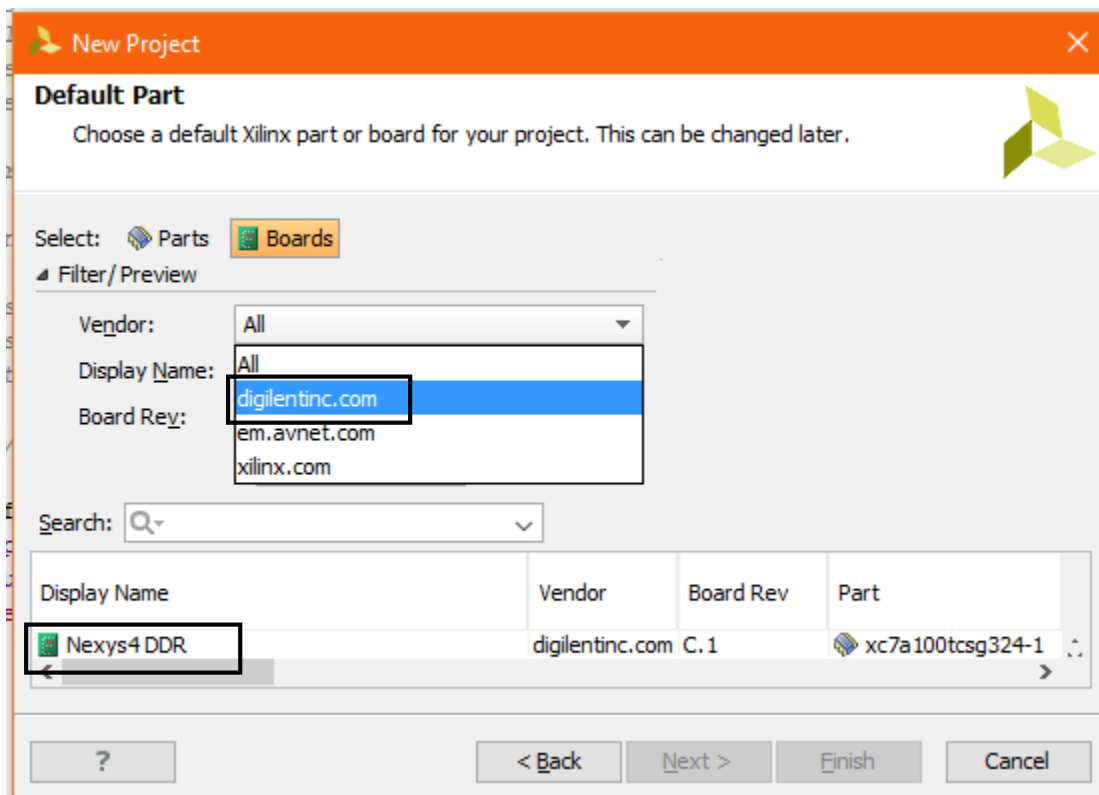


Figure 9: New project nexys4 DDR selection

8. Click [Finish]

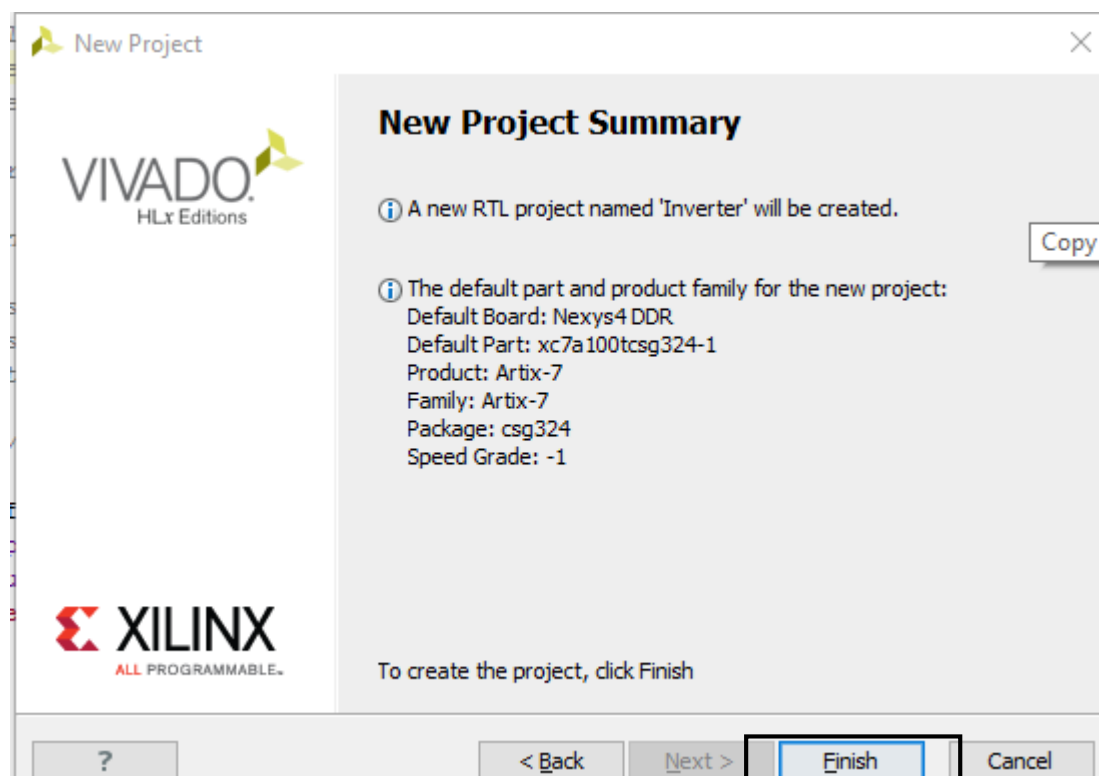


Figure 10: New project summary

Vivado design flow, gate level & hierarchical modelling

9. Click on [Add sources] to add code file.

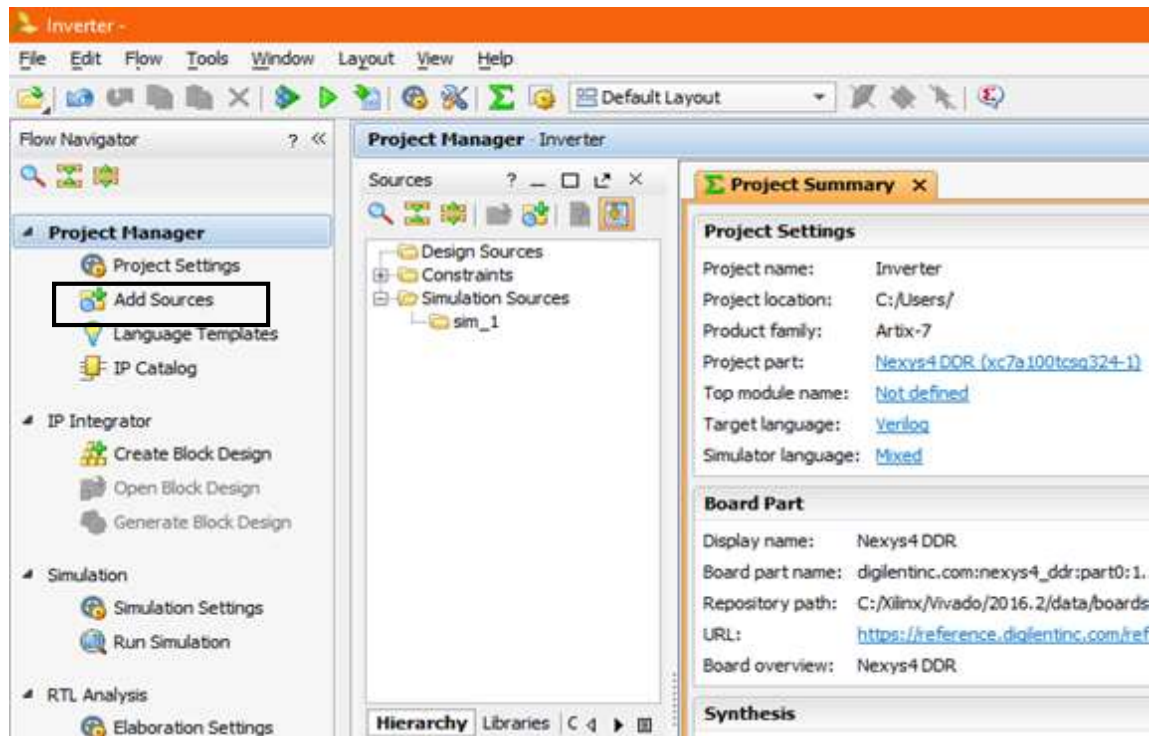


Figure 11: Adding sources to the project

10. Select [Add or create design sources] and then click [Next].

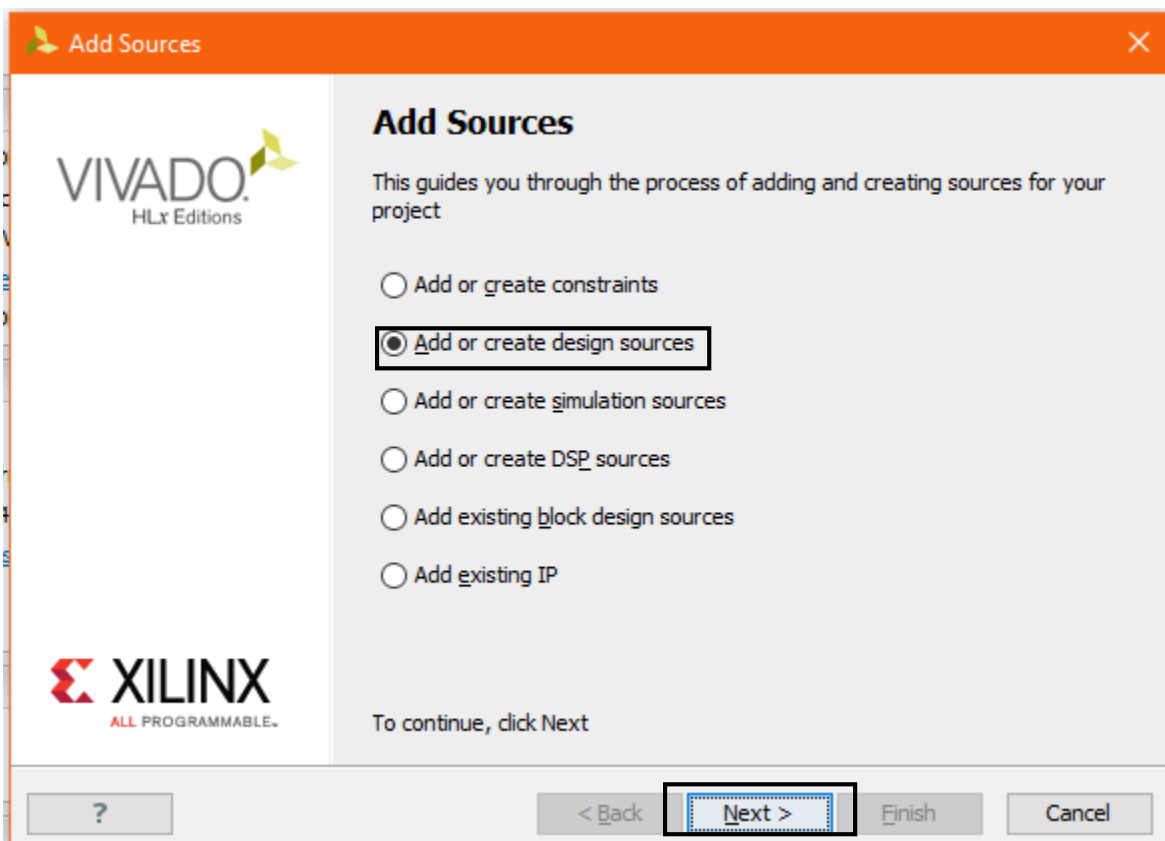


Figure 12: New project adding sources

Vivado design flow, gate level & hierarchical modelling

11. First click [Create file], then write 'Inverter' (source file name must be same as project name), click [OK] and in last click [Finish].

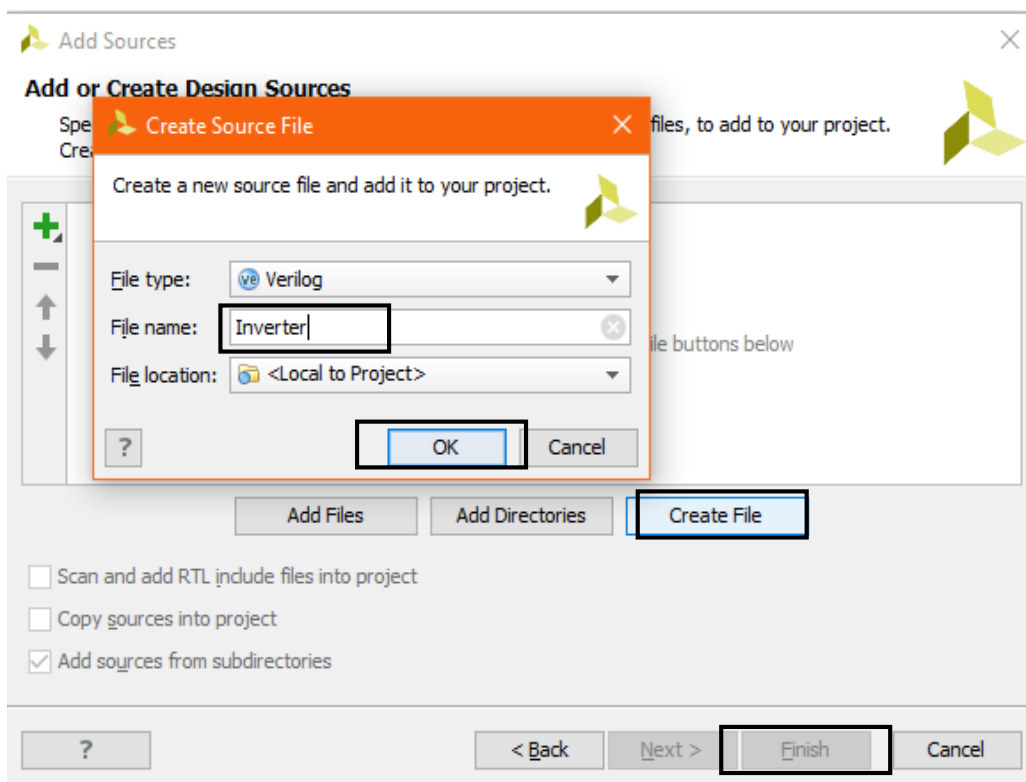


Figure 13: New Project source file creation

12. Click on [OK] and select [Yes] in the next popup window.

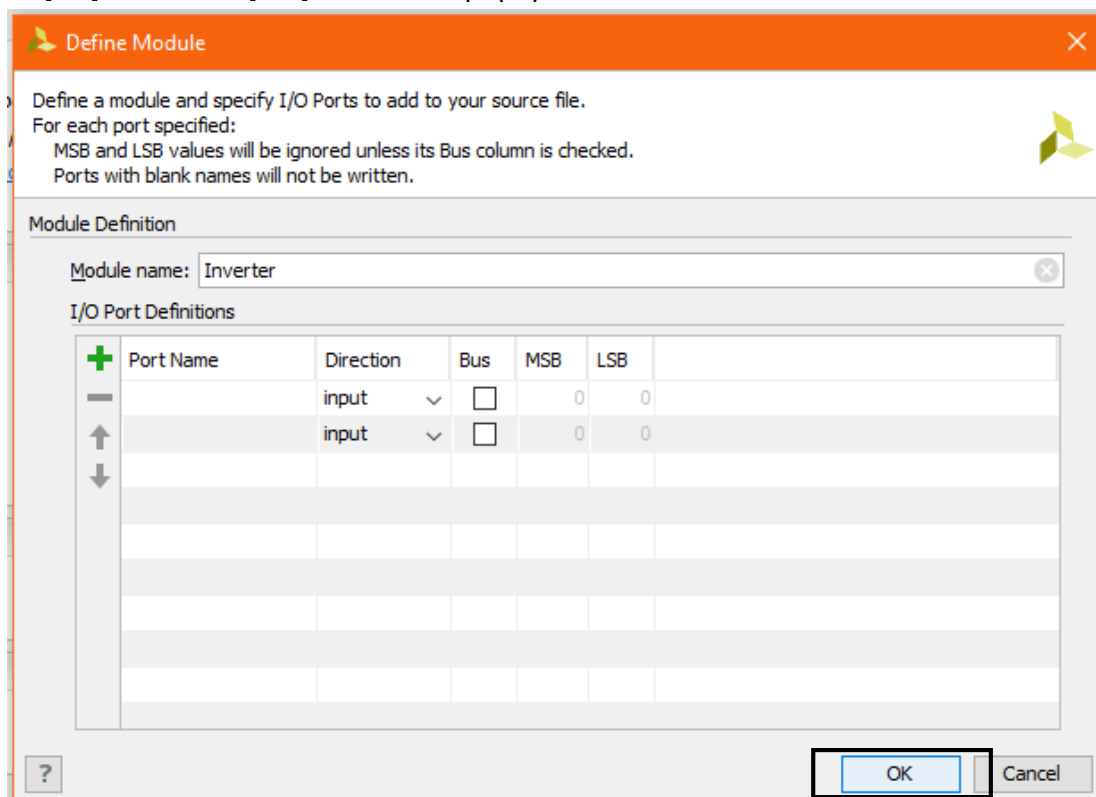


Figure 14: New project define module

Vivado design flow, gate level & hierarchical modelling

13. Double click on 'Inverter' file under 'design Sources' to open the Verilog code file.

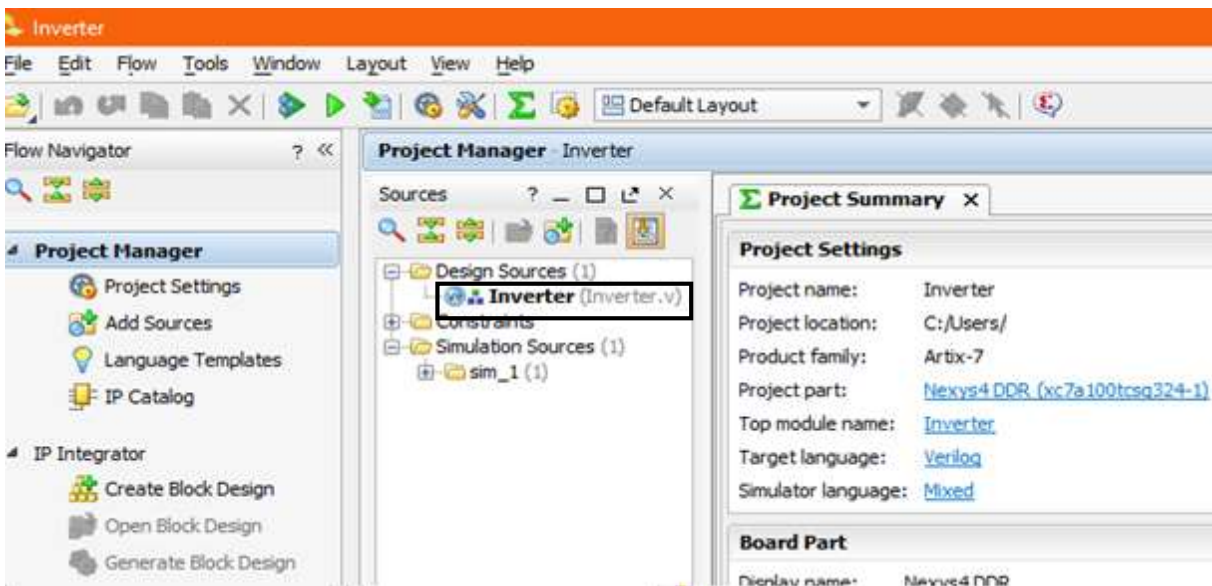


Figure 15: Verilog file opening

14. Write Code of inverter in the Verilog file

Listing: inverter

```

module Inverter(A_not, A);
input A;
output A_not;

    not not_gate (A_not, A);

endmodule

```

15. Save the source code file, click on [Run Simulation] from 'flow Navigator' and then click [Behavioral Simulation].

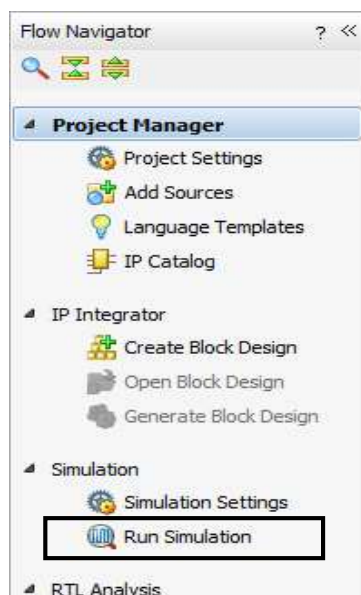


Figure 16: Running simulation

Vivado design flow, gate level & hierarchical modelling

16. Click on [Open Elaborated Design] from 'Flow Navigator'.

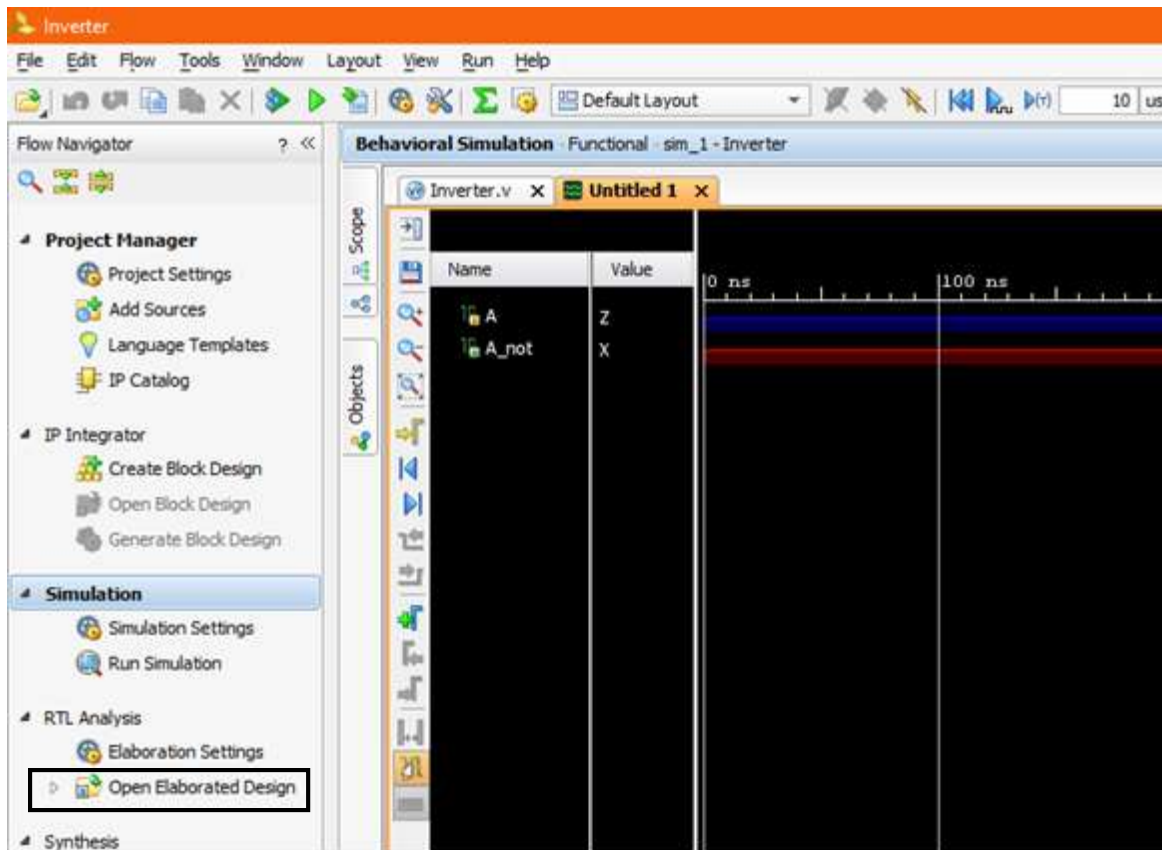


Figure 17: Opening elaborated design after simulation

17. Check your Schematic diagram with the respected gate, make sure the connections are fine. Then click on '2 I/O Ports' to declare input and output ports.

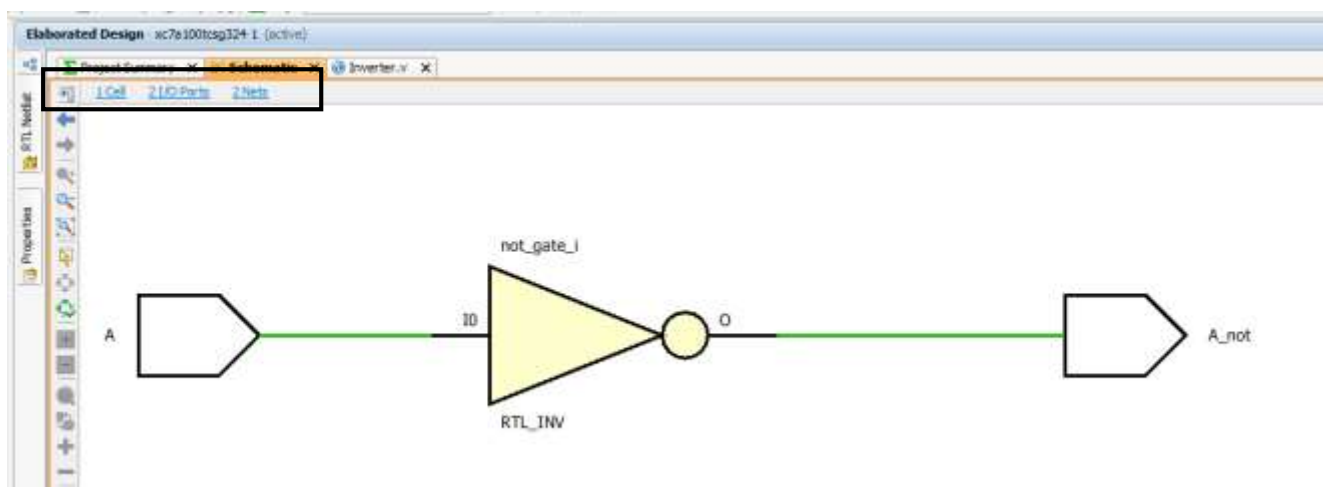
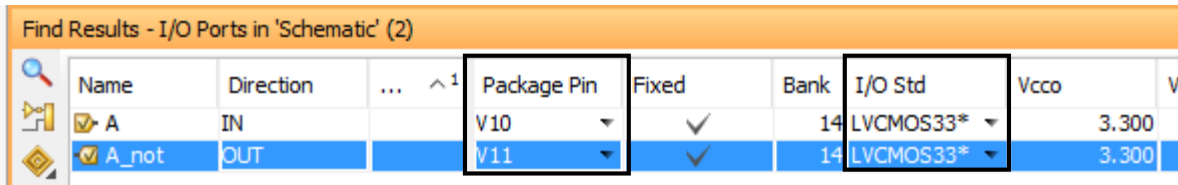


Figure 18: Schematic design and I/O ports

Vivado design flow, gate level & hierarchical modelling

18. Select any switch as input and any LED as output port from your board and change 'I/O std' to 'LVCMOS33*'.



Name	Direction	Package Pin	Fixed	Bank	I/O Std	Vcco	V
A	IN	V10	✓	14	LVCMOS33*	3.300	
A_not	OUT	V11	✓	14	LVCMOS33*	3.300	

Figure 19: I/O port assignment

19. Save Constraints file (I/O ports assignment file) by pressing 'ctrl+s', name the file and save it.

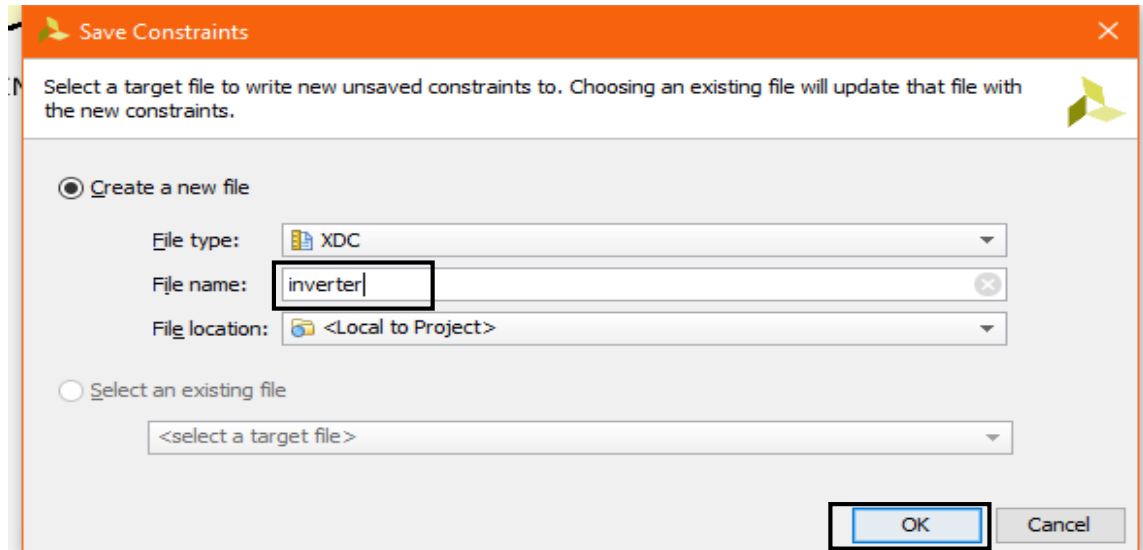


Figure 20: Save constraint file

20. After successfully saving constraints file, you can see it in sources window under the 'Constraints file'. Then, Click 'Run Synthesis', let the process happen once it is finished click Run Implementation. After Implementation is finished click 'Generate Bitstream'. These processes take a minute or two to finish. Click on 'generate Bitstream' File once synthesis and implementation is successfully done.

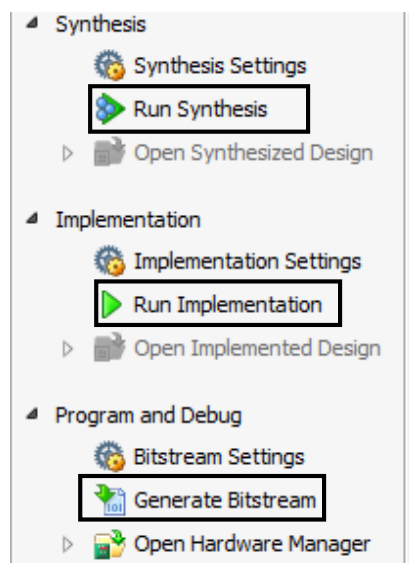


Figure 21: Run synthesis and implementation

21. Click on 'Open Hardware manager' File once Bitstream is successfully generated.

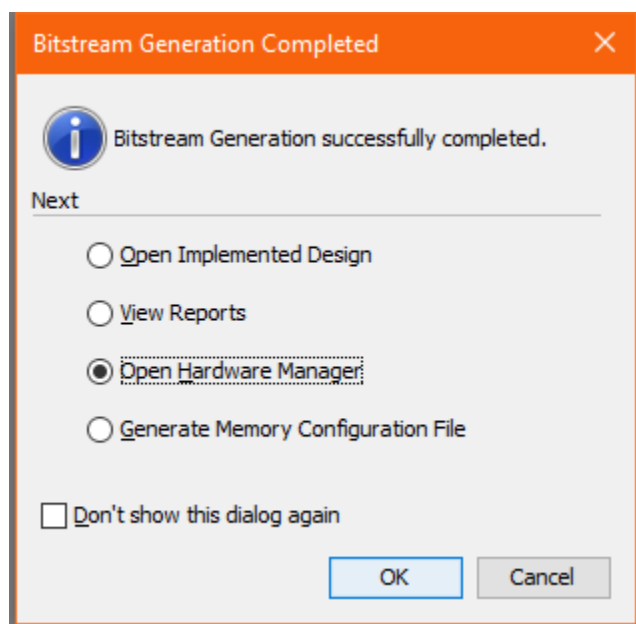


Figure 22: Opening hardware manager

22. Connect your Nexys4 DDR, Click on 'Open target', then select 'Auto Connect'.

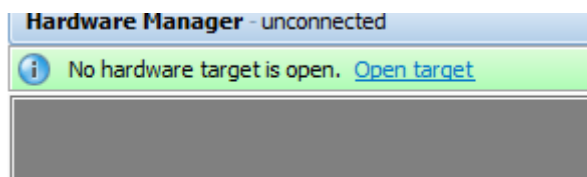


Figure 23: Hardware manager

23. Click on 'Program device' and click 'xc7a100_20' to program the bitstream file to the connected hardware.

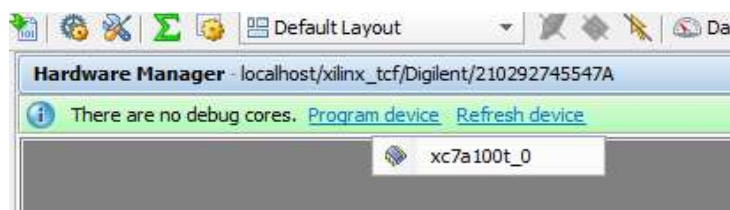


Figure 24: Program device and selection

24. Click on 'Program' to write the Verilog code to the development board.

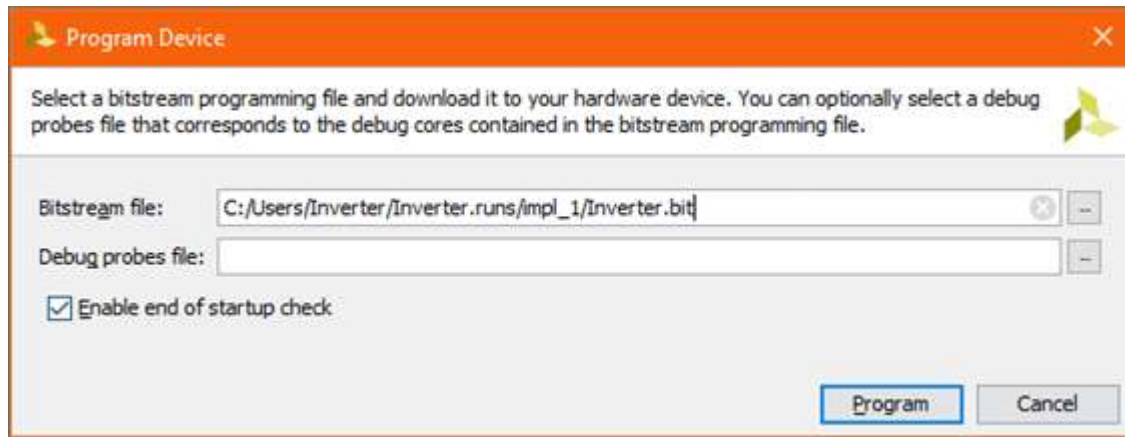


Figure 25: Programming Nexys4 DDR

The Verilog code of Inverter is now programmed to the development board. To check the implementation use the input switch that you declared in the constraints file and see it works fine.

Part 2 – Verilog HDL Gate Level Modelling

Gate-level Modelling:

Verilog is both a structural and behavioural language. It works on four levels of abstractions, switch level, gate level, data flow and behavioral modelling. Switch level is the lowest level of abstraction, as level of abstraction increases as the level of complexity increases. Gate level is implemented in terms of logic gates and interconnections between them.

In the digital design community, the term register transfer level (RTL) is used for Verilog description that uses a combination of behavioural and data flow modelling. Normally, the higher level of abstraction, design will be more flexible and technology-independent. Although, the lower level description provides high performance therefore as the design matures, high level modules are replaced with the gate-level modelling.

Verilog HDL supports built-in primitive gates modelling. The gates supported are multiple-input, multiple output, tri-state, and pull gates. The multiple-input gates supported are: **and**, **nand**, **or**, **nor**, **xor**, and **xnor** whose number of inputs are two or more, and has only one output. The multiple-output gates supported are **buf** and **not** whose number of output is one or more, and has only one input. The language also supports modelling of tri-state gates which include **bufif0**, **bufif1**, **notif0**, and **notif1**. These gates have one input, one control signal, and one output. The pull gates supported are **pullup** and **pulldown** with a single output (no input) only.

The basic syntax for each type of gates with zero delays is as follows:

and | **nand** | **or** | **nor** | **xor** | **xnor** [instance name] (out, in1, ..., inN); // [] is optional and | is selection

buf | **not** [instance name] (out1, out2, ..., out2, input);

bufif0 | **bufif1** | **notif0** | **notif1** [instance name] (outputA, inputB, controlC);

One can also have multiple instances of the same type of gate in one construct separated by a comma such as

and [inst1] (out11, in11, in12), [inst2] (out21, in21, in22, in23), [inst3] (out31, in31, in32, in33);

Verilog variables can only take one of four values, which represent the allowed binary states, logic high (1), logic low (0), do not care (X), and high impedance (Z). Verilog variable representation is diverse, to ease code development, by using the following format:

X' YZZZZZ

X is a number indicating the number of bits present in the string

- Y is a character symbol that indicates the number system (hex, octal, binary, etc.) that will be used to describe the value of the string.
- Z represents the series of numbers or characters that are used to represent the value of the literal in the current number system.

Once, we are familiar with the basic gate primitives, now we can start implementing basic combinational circuit such as Full Adder, as given below.

Lab Examples:

Example 1.1: Full Adder

The logic diagram for the 1-bit Full Adder is converted to a Verilog description.

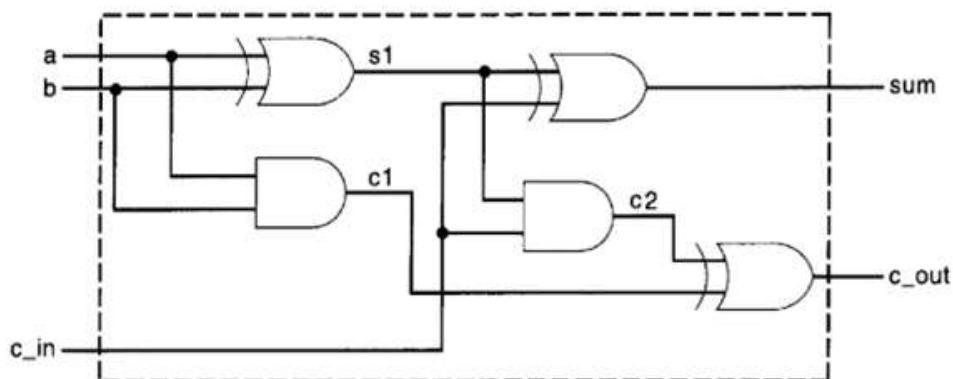


Figure 25: Full adder schematic diagram

Listing 1.1: Full Adder

```

module fulladd(sum, c_out, a, b, c_in);
    output sum, c_out;
    input a, b, c_in;
    wire s1, c1, c2;
    xor xor1(s1, a, b);
    and and1(c1, a, b);
    xor xor2(sum, s1, c_in);
    and and2(c2, s1, c_in);
    or or1(c_out, c2, c1);
endmodule

```

Example 1.2: bufif1 and bufif0

The logic diagram for the bufif1 and bufif0 is converted to a Verilog description.



Figure 26: Tristate switches bufif1 and bufif0

Listing 1.2: bufif1 bufif0

```
module tri_buf1(m_out, A, select);
    output m_out;
    input A, select;

    bufif1 buff1(m_out, A, select);
endmodule

module tri_buf0(m_out, B, select);
    output m_out;
    input B, select;
    Bufif0 buff0(m_out, B, select);
endmodule
```

Lab Exercises:

- 1) Design, and implement a 2-bit Comparator.
 - a) Create a truth table, then drive Boolean expressions and simplify the output expressions using K-maps. Submit the snapshots for handwritten work.
 - b) Implement the simplified design using Verilog gate-level modelling. Screenshot the codes and submit with the name signature.
 - c) Make sure codes are well commented with proper indentation.
- 2) Design, Implement, and test a 4-to-1 Multiplexer.
 - a) Create a truth table, then drive Boolean expressions. Submit the snapshots for handwritten work.
 - b) Implement the simplified design using Verilog gate-level modelling. Screenshot the codes and submit with the name signature.
 - c) Make sure codes are well commented with proper indentation.

Detailed readings and codes:

Verilog HDL: A Guide to Digital Design and Synthesis by Samir Palnitkar, 2nd Edition, chapter 5

<https://github.com/SAFEERHYDER/Digital-System-Design>

Part 3 – Hierarchical Modelling

Design Methodologies:

There are basically two design methodologies: a top-down design methodology and a bottom-up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build-up the top-level block. We further subdivide the sub-blocks until we come to leaf cells, which cannot be further divided, as shown in figure below.

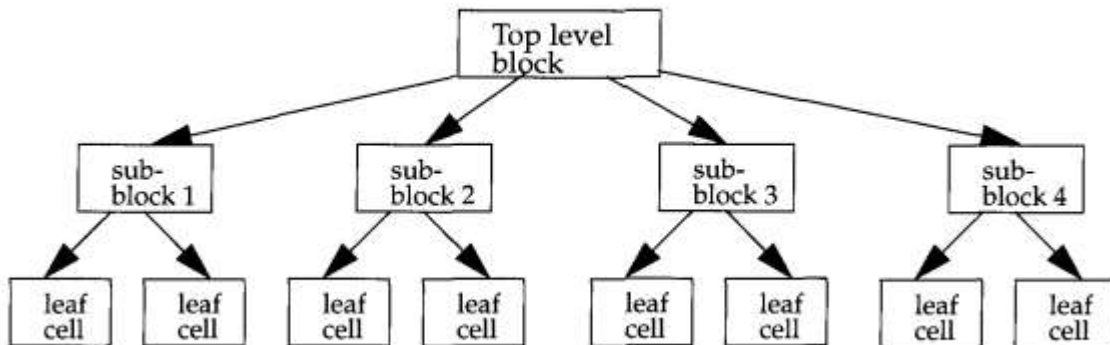


Figure 27 Top-down design Methodology

In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design. Figure 27 shows the bottom-up design process. Typically, a combination of top-down and bottom-up design flow is used.

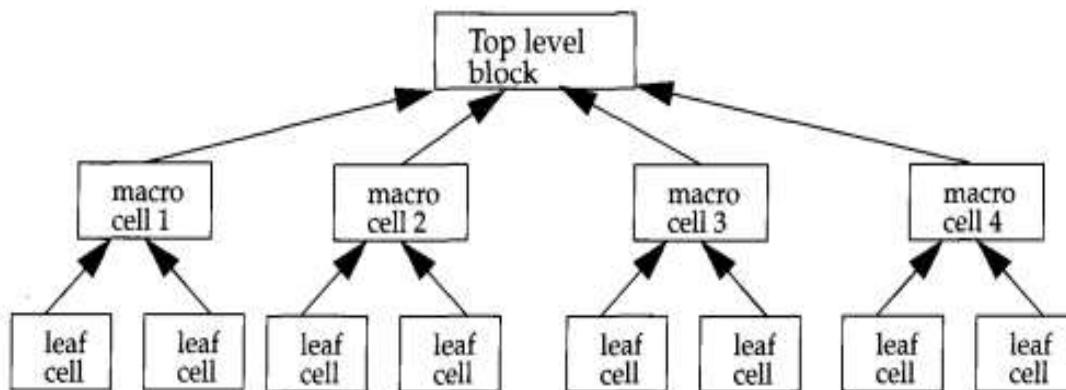


Figure 28 Down-top design methodology

Hierarchical Design

Hierarchical Design (HD) flow enable you to partition a design into smaller, more manageable modules to be processed independently. Using a modular approach to the hierarchical design lets you analyse modules independent of the rest of the design, and reuse modules in the top-down design. A team of users can iterate on specific sections of a design, achieving timing closure and other design goals, and reuse the results.

4-bit Ripple Carry Counter

The ripple carry counter shown in figure below is made up of negative edge-triggered toggle flip flops (T_FF). Each of the T_FF s can be made up from negative edge-triggered D-flip flop (D_FF) and inverters (assuming q_bar output is not available on the D_FF).

Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks.

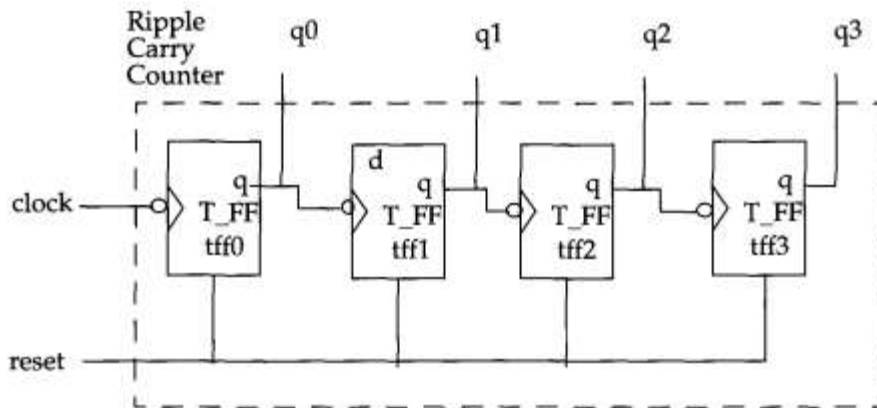


Figure 31 Ripple Carry Counter

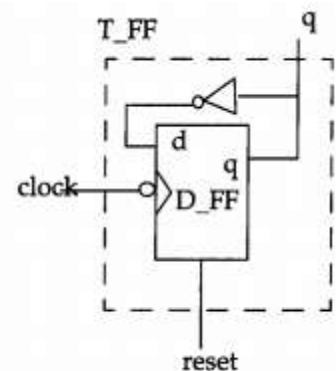


Figure 30 T-flipflop

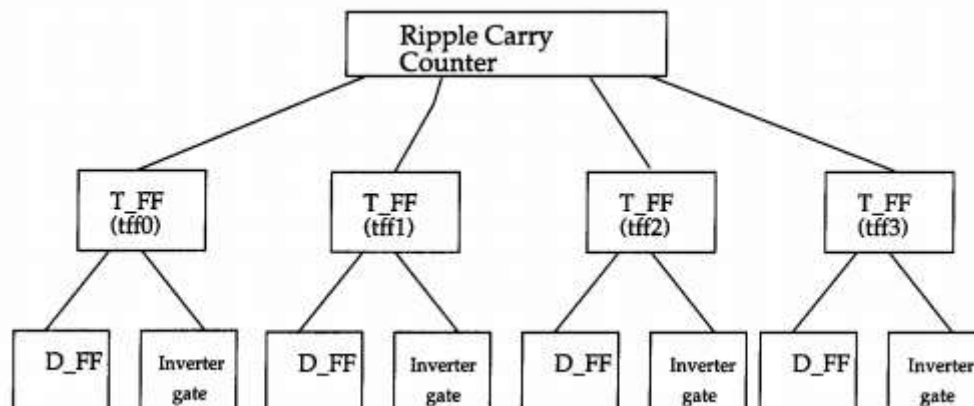


Figure 29: Design hierarchy for ripple carry adder

Difference between module and instance:

Modules

We now relate these hierarchical modelling concepts to Verilog. 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. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design. Ripple carry counter, T_FF , D_FF are examples of modules. 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.

Instances

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters, and I/O interface. The process of creating objects from a module template is called instantiation, and the objects are called instances. In example below, the top-level block creates four instances from the T-flip flop (*T_FF*) template. Each *T_FF* instantiates a *D_FF* and an inverter gate. Each instance must be given a unique name. Note that *//* is used to denote single-line comments.

Lab Examples:

Example 1.3: 4-bit ripple carry counter using hierarchical modelling

Listing 1.3: Ripple Carry Counter

```

module ripple_carry_counter(q, clk, reset);
    output [3:0] q;
    input clk, reset;
        T_FF tff0(q[0],clk, reset);
        T_FF tff1(q[1],q[0], reset);
        T_FF tff2(q[2],q[1], reset);
        T_FF tff3(q[3],q[2], reset);
endmodule

```

In the above module, four instances of the module *T_FF* (T-flip-flop) are used. Therefore, we must now define the internals of the module *T_FF*.

Example 1.4: Toggle flip flop

Listing1.4: Toggle Flip Flop

```

module T_FF(q, clk, reset);
    output q;
    input clk, reset;
    wire d;
        D_FF dff0(q, d, clk, reset);
        not n1(d, q);
endmodule

```

Example 1.5: D flip-flop

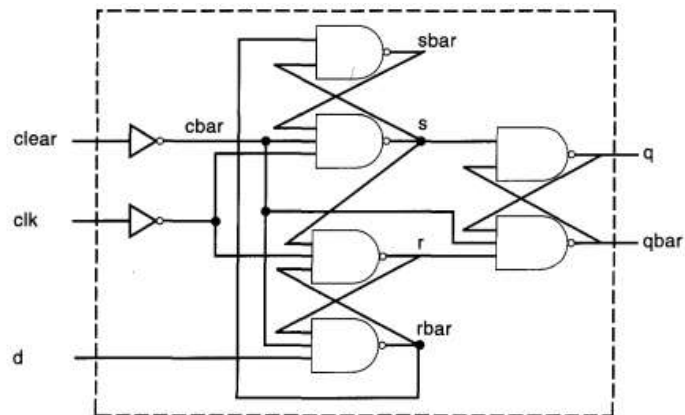


Figure 32: Edge-triggered D flip-flop

Listing 1.5: D Flip Flop

```

module D_FF(q, d, clk, reset);

output q;
input d, clk, reset;
wire s, sbar, r, rbar, cbar;           //Internal variables

assign    cbar = ~clear;
assign    sbar = ~(rbar & s),           //Input latches
            s = (sbar & cbar & ~clk),
            r = ~(rbar & ~clk & s),
            rbar = ~(r & cbar & d);

assign    q = ~(s & qbar),
            qbar = ~(q & r & cbar);

endmodule

```

Lab Exercises:

1. Design, Implement and investigate a 4-bit parallel adder by cascading four Full Adders.
 - a. Implement the Full Adder using gate-level modelling and test the design to make sure it is working. Submit the Verilog design and test bench code screenshots with name signatures. Also submit the tcl console and timing diagram outputs.
 - b. Implement the 4-bit adder by connecting pre-designed full adder modules. Submit the design codes screenshots with name signatures.
 - c. Test the design using Verilog test bench. Submit the test bench codes with name signatures along with the tcl console outputs and timing diagrams.
2. Design, implement and investigate a 16-to-1 multiplexer by interconnecting five 4-to-1 multiplexers using hierarchical modelling approach.
 - a. Provide the design with schematic diagram.
 - b. Implement the design using Verilog hierarchical modelling. Submit the design codes' screenshots with name signatures.
 - c. Test the design using both testbench simulation. Submit the test bench codes with tcl console and timing diagrams.
 - d. Test the design outputs using Nexys 4 DDR FPGA board. Submit output evidences.
 - e. Make sure codes are properly commented.

Detailed readings and codes:

Verilog HDL A Guide to Digital Design and Synthesis by Samir Palnitkar, 2nd Edition, chapter 2 and 5

<https://github.com/SAFEERHYDER/Digital-System-Design>