

# **LABORATORY MANUAL**

**DEPARTMENT OF  
ELECTRICAL & COMPUTER ENGINEERING**



**UNIVERSITY OF CENTRAL FLORIDA**

**EEE 3342  
Digital Systems**

Revised  
January 2025

# **CONTENTS**

Safety Rules and Operating Procedures

Introduction

Experiment #1 Introduction to Xilinx's FPGA Vivado HLx Software

Experiment #2 Introduction to Testbenches

Experiment #3 Boolean Simplification & Digital Circuit Area Cost

Experiment #4 Hardware Applications of Binary Adders

Experiment #5 Multiplexers & Decoders

Experiment #6 Latches & Flip-Flops

Experiment #7 Hardware Memory Elements

Experiment #8 Finite State Machines

Experiment #9 Designing with D-Flip flops: Shift Register and  
Sequence Counter

Experiment #10 Sequential Circuit Design: Counter with Inputs

Experiment #11 Sequential Design

# Safety Rules and Operating Procedures

1. Note the location of the Emergency Disconnect (red button near the door) to shut off power in an emergency. Note the location of the nearest telephone (map on bulletin board).
2. Students are allowed in the laboratory only when the instructor is present.
3. Open drinks and food are not allowed near the lab benches.
4. Report any broken equipment or defective parts to the lab instructor. Do not open, remove the cover, or attempt to repair any equipment.
5. When the lab exercise is over, all instruments, except computers, must be turned off. Return substitution boxes to the designated location. Your lab grade will be affected if your laboratory station is not tidy when you leave.
6. University property must not be taken from the laboratory.
7. Do not move instruments from one lab station to another lab station.
8. Do not tamper with or remove security straps, locks, or other security devices. Do not disable or attempt to defeat the security camera.
9. When touching the FPGA development boards please do not touch the solid-state parts on the board but handle the board from its edge.
- 10. ANYONE VIOLATING ANY RULES OR REGULATIONS MAY BE DENIED ACCESS TO THESE FACILITIES.**

I have read and understand these rules and procedures. I agree to abide by these rules and procedures at all times while using these facilities. I understand that failure to follow these rules and procedures will result in my immediate dismissal from the laboratory and additional disciplinary action may be taken.

---

Signature

---

Date

---

Lab #

# Laboratory Safety Information

## Introduction

The danger of injury or death from electrical shock, fire, or explosion is present while conducting experiments in this laboratory. To work safely, it is important that you understand the prudent practices necessary to minimize the risks and what to do if there is an accident.

## Electrical Shock

Avoid contact with conductors in energized electrical circuits. The typical can not let-go (the current in which a person can not let go) current is about 6-30 ma (OSHA). Muscle contractions can prevent the person from moving away the energized circuit. Possible death can occur as low 50 ma. For a person that is wet the body resistance can be as low as 1000 ohms. A voltage of 50 volts can result in death.

Do not touch someone who is being shocked while still in contact with the electrical conductor or you may also be electrocuted. Instead, press the Emergency Disconnect (red button located near the door to the laboratory). This shuts off all power, except the lights.

Make sure your hands are dry. The resistance of dry, unbroken skin is relatively high and thus reduces the risk of shock. Skin that is broken, wet, or damp with sweat has a low resistance.

When working with an energized circuit, work with only your right hand, keeping your left hand away from all conductive material. This reduces the likelihood of an accident that results in current passing through your heart.

Be cautious of rings, watches, and necklaces. Skin beneath a ring or watch is damp, lowering the skin resistance. Shoes covering the feet are much safer than sandals.

If the victim isn't breathing, find someone certified in CPR. Be quick! Some of the staff in the Department Office are certified in CPR. If the victim is unconscious or needs an ambulance, contact the Department Office for help or call 911. If able, the victim should go to the Student Health Services for examination and treatment.

## Fire

Transistors and other components can become extremely hot and cause severe burns if touched. If resistors or other components on your proto-board catch fire, turn off the power supply and notify the instructor. If electronic instruments catch fire, press the Emergency Disconnect (red button). These small electrical fires extinguish quickly after the power is shut off. Avoid using fire extinguishers on electronic instruments.

## First Aid

A first aid kit is located on the wall near the door. Proceed to Student Health Services, if needed.

---

## Introduction

## When in Doubt, Read This

---

Laboratory experiments supplement class lectures by providing exercises in analysis, design and realization. The objective of the laboratory is to present concepts and techniques in designing, realizing, debugging, and documenting digital circuits and systems. The laboratory begins with a review of Xilinx's VIVADO FPGA development environment, which will be used extensively during the laboratory. Experiment #1 introduces the student to the fundamentals of the VIVADO and its tool set such as the synthesizer, the test-bench user input program for the simulator, the VIVADO simulator, and the FPGA implementation. Xilinx's FPGA development tools support VERILOG. In Experiment #2, the basic operations found in the VIVADO will be used to design and simulate a simple Boolean expression that will be using the student experimenter kit using 74LSXXXX parts. Experiments #3 through #7 are experiments that deal with the design and hardware implementation of combinational logic circuits. These circuits will be designed using the VIVADO and implemented solely using an FPGA. Experiments #8 through #11 deal with the design and hardware implementation of sequential logic circuits and will also be designed and implemented using the FPGA and VIVADO development tools. The VIVADO also offers an extensive set of manuals under the Help menu. In addition to the electronic version of the manual and the Quick Start Tutorial, Xilinx offers many tutorials that are available on the web site at [www.xilinx.com](http://www.xilinx.com).

### Laboratory Requirements:

This laboratory requires that each student obtains a copy of this manual, a bound quad-ruled engineering notebook and have access to Xilinx's VIVADO version 2017.4. The student can use the VIVADO program on the laboratory computers or the student can go to <https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2017-4.html> in order to download the Vivado Design Suite - Hx Editions.

The student is to prepare for each laboratory by reading the assigned topics. The laboratory notebook should contain the necessary tables, graphs, logic and pin assignment diagrams and identify the expected results from the laboratory exercise as directed by the pre-laboratory preparation assignments or by the laboratory instructor. Depending on the laboratory assignment, the pre-laboratory preparation may be due at the beginning of the laboratory period or may be completed during the assigned laboratory period. Be informed that during each laboratory period the instructor will grade your notebook preparation.

During the laboratory period you are expected to construct and complete each laboratory assignment, record data, and deviations from your expected results, equipment used, laboratory partners, and design changes in your laboratory notebook. A laboratory performance grade will be assigned by the laboratory instructor upon successful completion of the above-described tasks for each experiment.

Each student will be assigned to a computer with an FPGA board connected to it. Each student is responsible for his or her own work including design and documentation. A Laboratory Report, following the guidelines presented in this handout is due the laboratory period following the completion of the in-laboratory work or when the instructor designates. A numeric grade will be assigned using the attached laboratory-grading sheet.

Laboratory reports will be due before the start of each laboratory. A penalty of five points will be charged for those late by up to one week (0-7 days). No credit will be given for laboratory reports over-due by more than one week. However, a student must complete each assigned experiment in order to complete the laboratory. By not turning in a laboratory report, a student will receive an incomplete for that report, which results in an incomplete for the laboratory grade.

Students who miss the laboratory lecture should make arrangements to make up the laboratory at a later time. Points may be taken off the laboratory experiment and the student might not be allowed to attend the remainder of the laboratory because this will burden the laboratory instructor and the rest of the laboratory that day.

Students who are late to their laboratory section will not receive the five pre-laboratory points. A penalty of five points will also be charged for turning in a late laboratory report. If, for some reason, a student cannot attend the regularly scheduled laboratory time period, then he / she must make arrangements to make up the laboratory experiment at a later time and hand in the laboratory report and pre-laboratory early to avoid a ten point penalty.

### **Laboratory Point Breakdown**

#### **In-Laboratory Grade (10 points):**

- |                                   |          |
|-----------------------------------|----------|
| 1. Pre-Laboratory Assignment..... | 5 points |
| 2. Design Completion.....         | 5 points |

#### **Laboratory Report Grade (15 points):**

- |   |                  |
|---|------------------|
| 3. Problem or Objective Statement, Block Diagram, and Apparatus List..... | 1 point          |
| 4. Procedure and Data or Design Steps.....                                | 3 points         |
| 5. Results Statement and Logic Schematic Diagram.....                     | 4 points         |
| 6. Design Specification Plan .....  | 2 points         |
| 7. Test Plan .....  | 2 points         |
| 8. Conclusion Statement .....   | 3 points         |
| <b>TOTAL.....</b>   | <b>25 points</b> |

The final laboratory grade can be a percentage, an incomplete or a failing grade. If the student receives an incomplete or failing grade for the laboratory, an incomplete may be assigned for the whole course.

### **Guidelines for Laboratory Reports:**

The laboratory report is the record of all work pertaining to the experiment, which includes any pre-laboratory assignments, schematic diagrams, and Xilinx's ISE printouts when applicable. This record should be sufficiently complete so that you or anyone else of similar technical background can duplicate the experiment by simply following your laboratory report. **Original work is required by all students (NO PHOTOCOPIES OR DUPLICATE PRINTOUTS).** Your laboratory report is an individual effort and should be unique. The laboratory notebook must be used for recording data. Do not trust your memory to fill in the details at a later time. An engineer will spend 75 percent of his/her time for documentation.

Organization in your report is important. It should be presented in chronological order with descriptive headings to separate and identify the various parts of the experiment. A neat, organized and complete record of the experiment is just as important as the experimental work. **DO NOT SECTION OFF DIAGRAMS, PROCEDURES, AND TABLES.**

The following are general guidelines for your use. Use the standard paper prescribed by your instructor. A cover page is required for each laboratory including your name, PID, name and number of the experiment, date of the experiment and the date the report is submitted. Complete the required information and attach to the front of each report. If a cover page is not included with a report, then points may be taken off.

The report should contain the following (not segmented or necessarily in this order):

- **Heading:** The experiment number, your name, and date should be at the top right hand side of each page.
- **Objective:** A brief but complete statement of what you intend to design or verify in the experiment should be at the beginning of each experiment.
- **Block Diagram:** A block diagram of the circuit under test and the test equipment should be drawn and labeled so that the actual experiment circuitry could be easily duplicated at any time in the future.
- **Apparatus List:** List the items of equipment, including IC devices, with identification numbers using the UCF label tag, make, and model of the equipment. It may be necessary later to locate specific items of equipment for rechecks if discrepancies develop in the results. Also include the computer used and the version number of any software used.
- **Procedure and/or Design Methodology:** In general, lengthy explanations are unnecessary. Be brief. Keep in mind the fact that the experiment must be reproducible from the information given in your report. Include the steps taken in the design of the circuit: Truth Table, assumptions, conventions, definitions, Karnaugh Map(s), algebraic simplification steps, etc.

- **Design Specification Plan:** A detailed discussion on how your design approach meets the requirements of the laboratory experiment should be presented. Given a set of requirements there are many ways to design a system that meets these requirements. The Design Specification Plan describes the methodology chosen and the reason for the selection (why). The Design Specification Plan is also used to verify that all the requirements of the project have been implemented as described by the requirements.
- **Detailed Schematic Diagram:** A detailed schematic diagram should be presented. Standard symbols should be used. For logic diagrams, inputs should enter at the left side or top of the diagram and the outputs at the bottom or right side of the diagram. Data flows left to right and top to bottom. If switches and LEDs are used for logic inputs and to test logic outputs respectively, the switch numbers and LED numbers should be identified. The switches and LEDs should be organized to simplify the testing of the circuit. A location diagram should be included at the bottom of each schematic diagram. For Experiment #2 74LSXXXX parts will be used in addition to the FPGA BASYS board. The BASYS board layout with the appropriate FPGA pins layout must be included in the laboratory report. See the Sample Schematic Diagram in Appendix C for a good example of a detailed diagram and Appendix D for the pin layout of the switches, LED's and Clock signals for the BASYS development board.
- **Test Plan:** A test plan describes how to test the implemented design against the given requirement specifications. This plan gives detailed steps during the test process defining which inputs should be tested and verifying for these inputs that the correct outputs appear. The laboratory instructor will use this test plan to test your laboratory experiment.
- **Results:** The results should be presented in a form, which makes the interpretation easy. Large amounts of numerical results are generally presented in a graphical form. Tables are generally used for a small amount of results. Theoretical and experimental results should be on the same graph or arranged in the same table for easy correlation of these results. For digital data, prepare a simulation and response table and record logic levels as "1"s and "0"s. The above table is similar to a Karnaugh Map or State Transition Table. Identification of the size of a logic circuit, in terms of inputs, gates and packages is often required in a design-oriented experiment.
- **Conclusion:** This is your interpretation of the objectives, procedures and results of the experiment, which will be used as a statement of what you learned in performing the experiment. This is not a summary. Be brief and specific but complete. Identify the advantages and/or disadvantages of your solution in design-oriented experiments. The conclusion also includes the answers to the questions presented in each experiment.



---

## EXPERIMENT #1

### Introduction to Xilinx's FPGA Vivado HLx Software

---

**Goals:** To introduce the modeling, simulation and implementation of digital circuits using Xilinx's FPGA VIVADO HLx Editions design tools.

**References:** Within the VIVADO, there are several documentation and tutorials that are available. In particular, the “Quick Take Videos” provides basic instructions such as how to create a file using either VHDL or VERILOG. The VIVADO also offers an extensive set of manuals under the Help menu. In addition to the electronic version of the manual and the Quick Sort Tutorial, Xilinx offers many tutorials that are available on the web site at [www.xilinx.com](http://www.xilinx.com).

**Equipment:** The Xilinx's FPGA VIVADO HLx Editions design tools are available in the laboratory. These tools can also be downloaded from Xilinx's web site at [www.xilinx.com](http://www.xilinx.com). The WebPack version of this tool that we use for the laboratory experiments are located under the support download section at the related website (<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2017-4.html>). Please take note that the file download size exceeds 1 GB and also during the installation process updates may have to be installed. It can take you up to a few hours to download and install the software on your computer. The user does not need to have the BASYS development board interface to the computer to design and simulate an FPGA.

**Pre-laboratory:** Read this experiment carefully to become familiar with the procedural steps in this experiment.

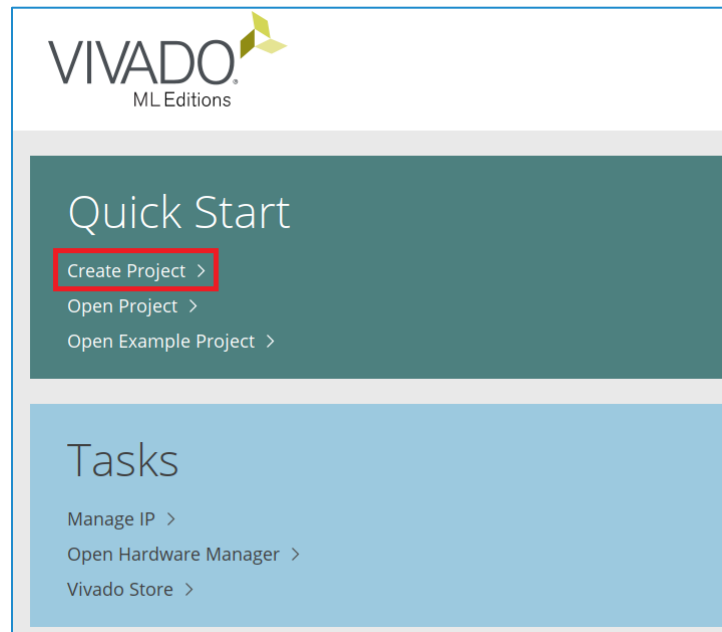
**Discussion:** Xilinx's Vivado is an FPGA design, simulation and implementation tool set that allows the designer the ability to develop digital systems using either schematic capture or HDL using VERILOG or VHDL. These digital systems are then verified using simulation tools that are part of the development system. Once the simulation outputs meet the design requirements, implementation is simply assigning the inputs and outputs to the appropriate pins on the FPGA. Appendix D gives the pin configuration for the BASYS board by Digilent Inc. ([www.digilentinc.com](http://www.digilentinc.com)) relating the LED and switch connections to the FPGA pin assignments.

Experiment #1 is divided into three sections. Part 1 of this experiment will guide the student through the steps required to create a digital design using Verilog. Next, the steps required to simulate this design are given along with the steps required to synthesize and implement the design on the FPGA BASYS 3 board. Part 2 is an expansion of part 1 for additional logic gates (NAND, OR, XOR, NOT). In part 3, a two-input five-output logic system is required to be designed and implemented.

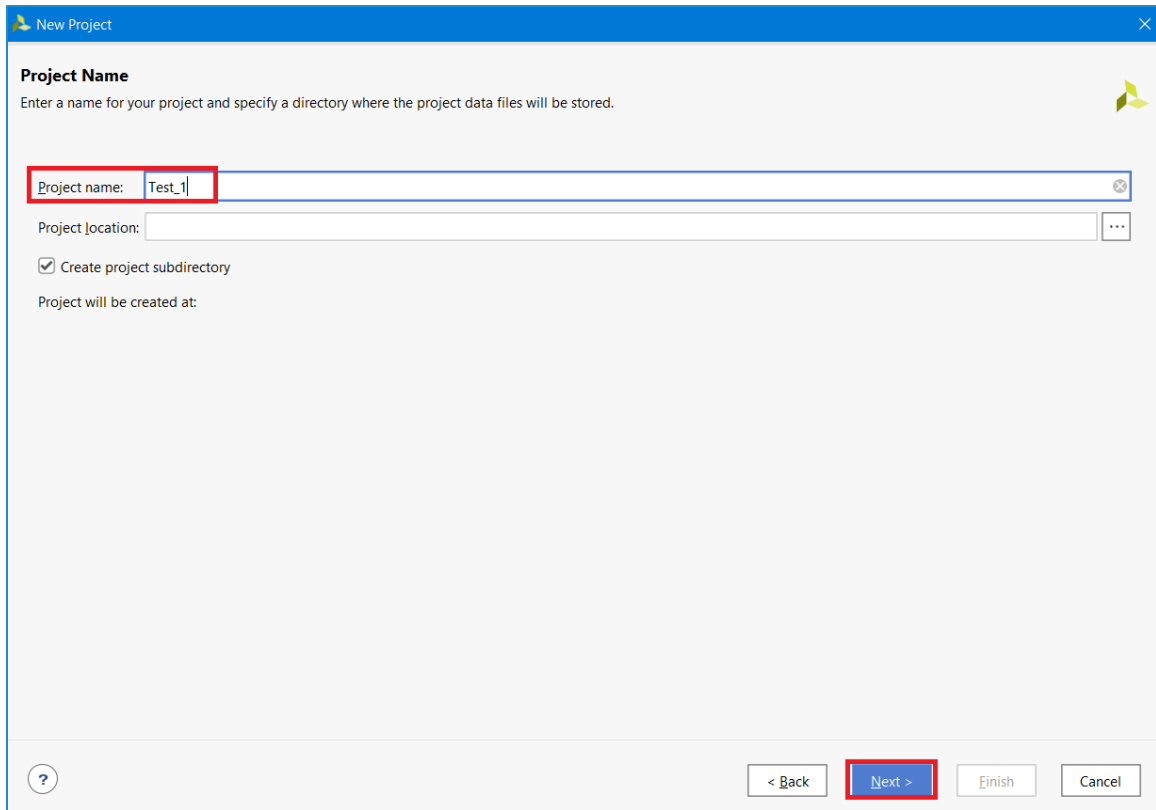
## Part 1. Introduction to the XILINX Vivado

In this part, you will use Xilinx's Vivado to design, simulate and implement a simple 2 input AND gate digital circuit. Once completed, this 2 input AND gate implementation will be downloaded to the BASYS board and then tested using the on board LED's and switches.

1. 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”.



2. Now, click on “Create Project” to create a new project. You have to be careful about where to save your project file in the computer lab. The computers in the lab run a hard disk protection program that could interfere with Xilinx. So, if you save your project in a preserved folder, Xilinx might have problem with running the simulation. You have two choices: (1) either save the project directly on your USB flash disk. This option is good since your USB disk have normal read/write access and Xilinx runs correctly. However, this option can be slow for USB flash disks. The option (2) is to save the project in a folder that's in the desktop. Then, compress the folder into a ZIP file and email it to yourself. Start by creating a folder on the desktop called 'Lab\_1'. Create this folder in Windows, not from Xilinx. Then, in Xilinx, create a new project inside “**Lab\_1**”. Name your project, “**Test\_1**” and it will be in the folder “**\Desktop\Lab\_1\Test\_1**”. When you finish your lab, you can copy your project on your flash disk.



**New Project**

**Project Name**  
Enter a name for your project and specify a directory where the project data files will be stored.

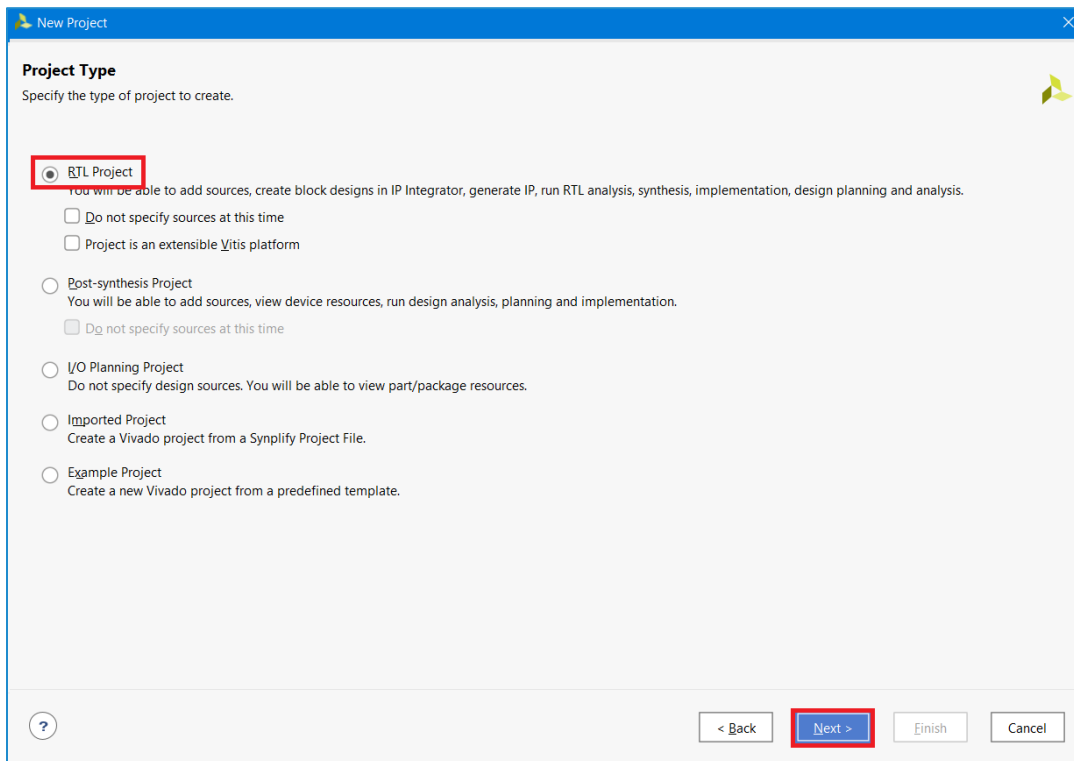
Project name:

Project location:  ...

☒ Create project subdirectory

Project will be created at:

3. In the next window, choose “RTL Project” as the project type. You can the description of this type in the window.



**New Project**

**Project Type**  
Specify the type of project to create.

☒ **RTL Project**  
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.  
☐ Do not specify sources at this time  
☐ Project is an extensible Vitis platform

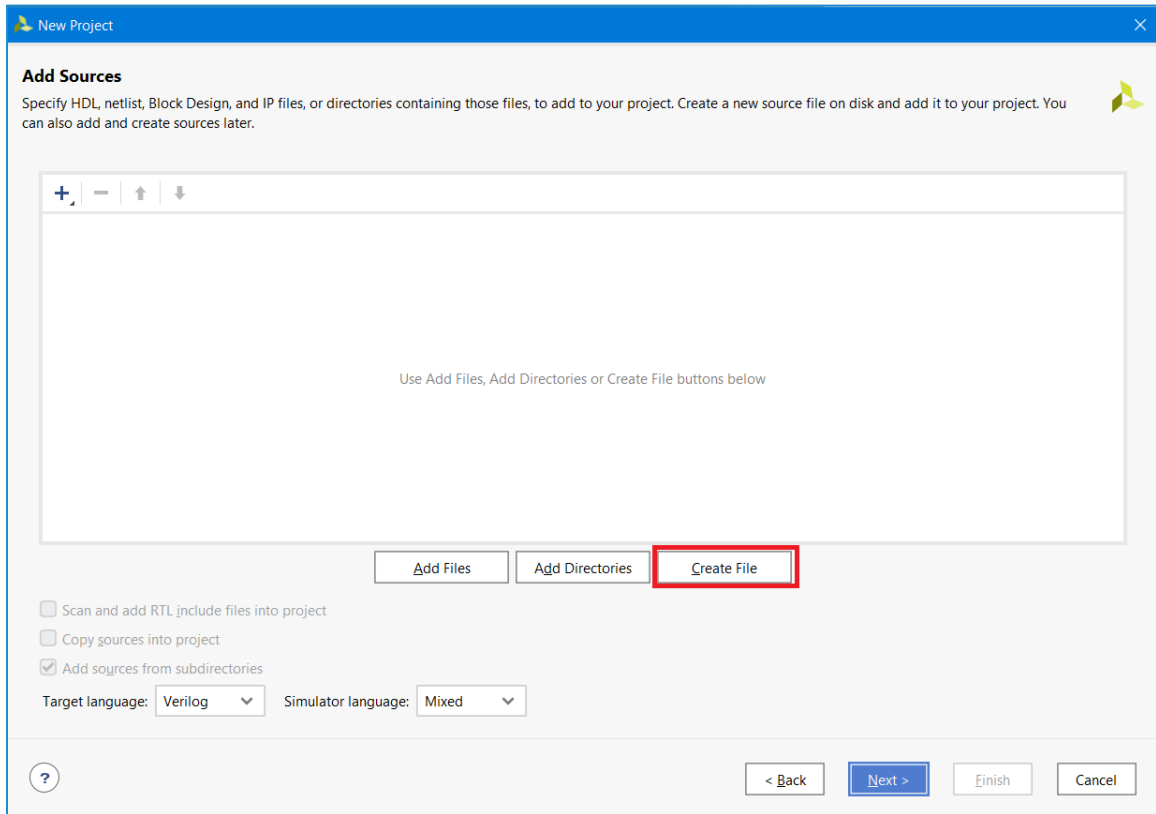
☐ **Post-synthesis Project**  
You will be able to add sources, view device resources, run design analysis, planning and implementation.  
☐ Do not specify sources at this time

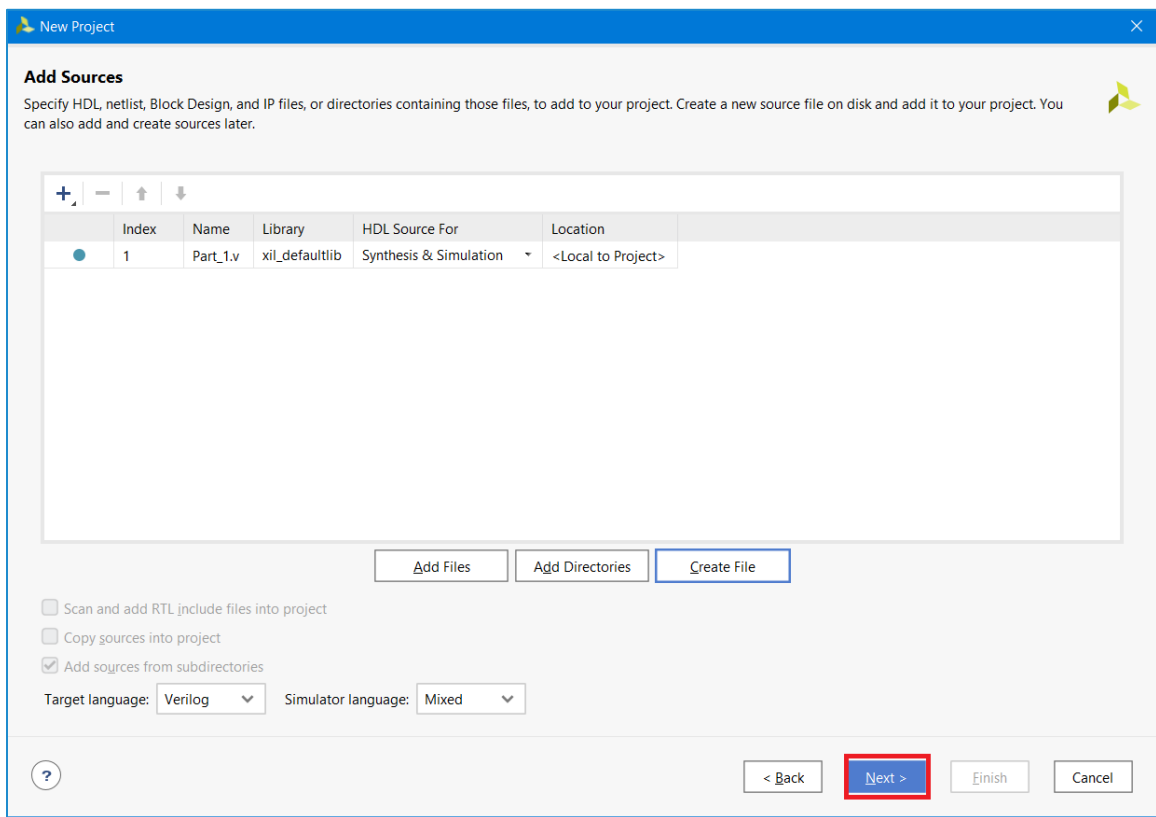
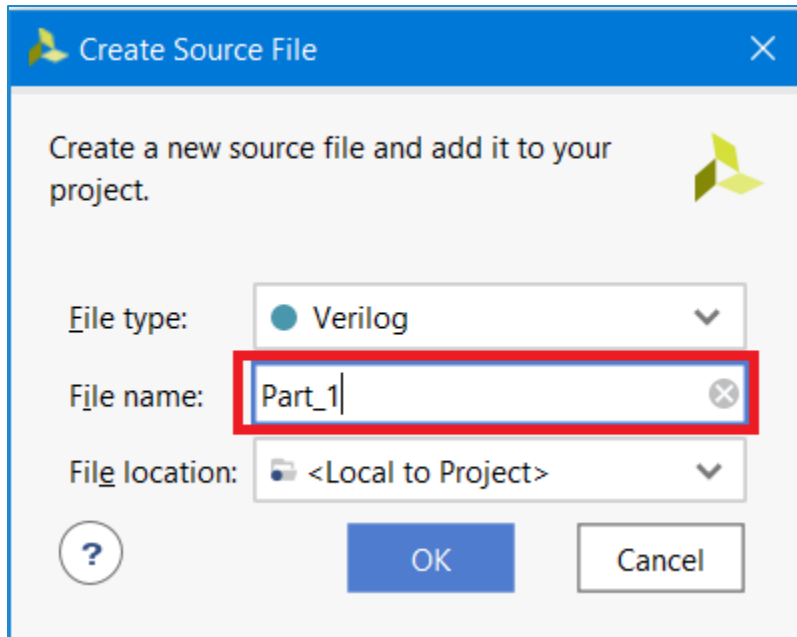
☐ **I/O Planning Project**  
Do not specify design sources. You will be able to view part/package resources.

☐ **Imported Project**  
Create a Vivado project from a Synplify Project File.

☐ **Example Project**  
Create a new Vivado project from a predefined template.

4. In the opened window, you can create source file (Verilog/Verilog Header/SystemVerilog/VHDL/Memory File) for your new project or add sources from the existing projects. Click on “Create File”, and in the opened window choose “Verilog” for the “File type”, write a name for your file (“Part\_1”), and click on “Ok”. Continue clicking on Next until reaching the “Default Part” window.





- In this window, choose “Artix-7” for the “Family”, “-1” for “Speed grade”, and “cpg236” for “Package”. In the shown parts, select “xc7a35tcpg236-1”. Take a look at the configuration of this part for your own familiarity.

New Project

### Default Part

Choose a default Xilinx part or board for your project.

Parts

Boards

Reset All Filters

Category: All

Family: Artix-7

Package: cpg236

Speed: -1

Temperature: All Remaining

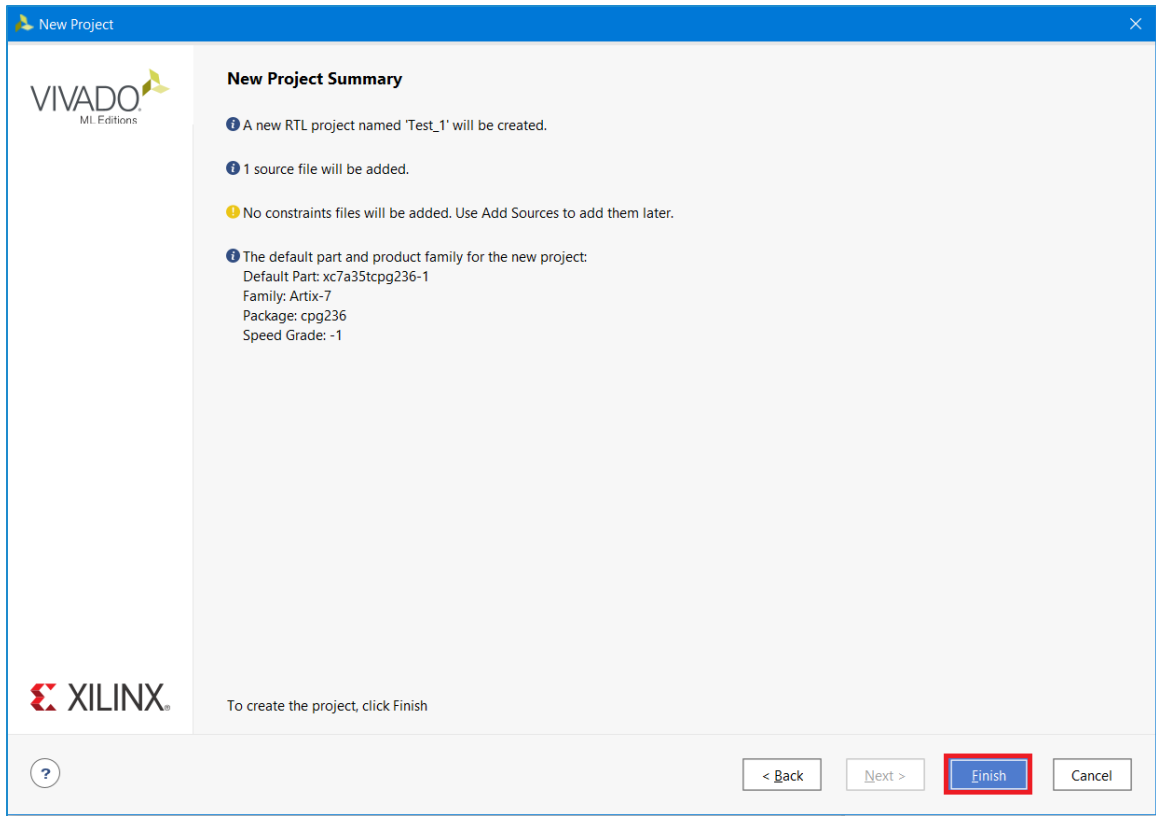
Static power: All Remaining

Search: Q-

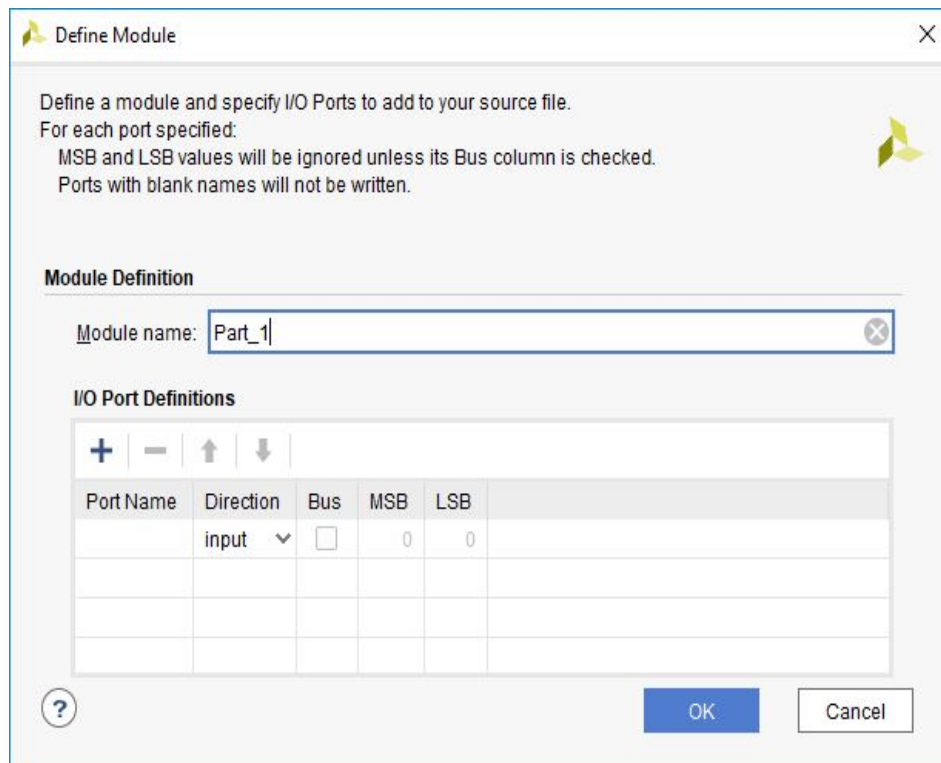
Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	HNICX	BUFGs	DDRMC
xc7a15tcbg236-1	236	106	10400	20800	25	0	45		32	
xc7a35tcbg236-1	236	106	20800	41600	50	0	90		32	
xc7a50tcbg236-1	236	106	32600	65200	75	0	120		32	

?
< Back
Next >
Finish
Cancel

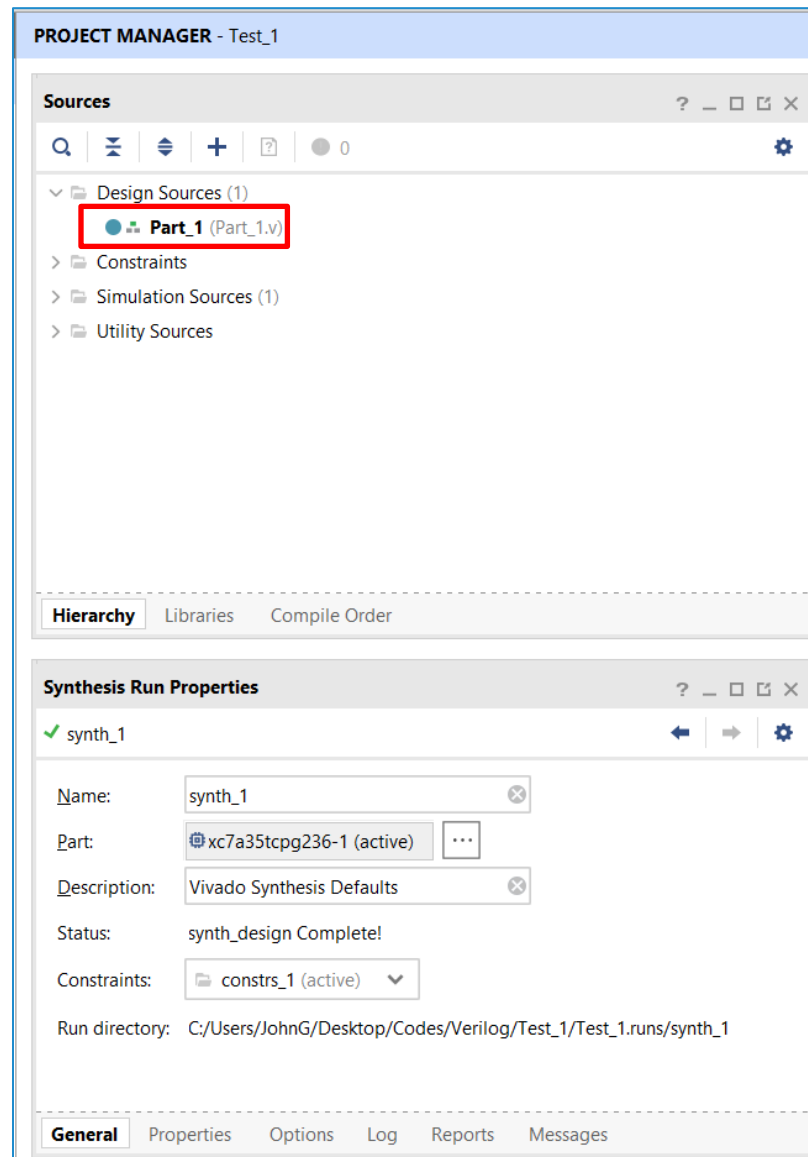




7. Define the input and the output ports of your module according to the shown window.

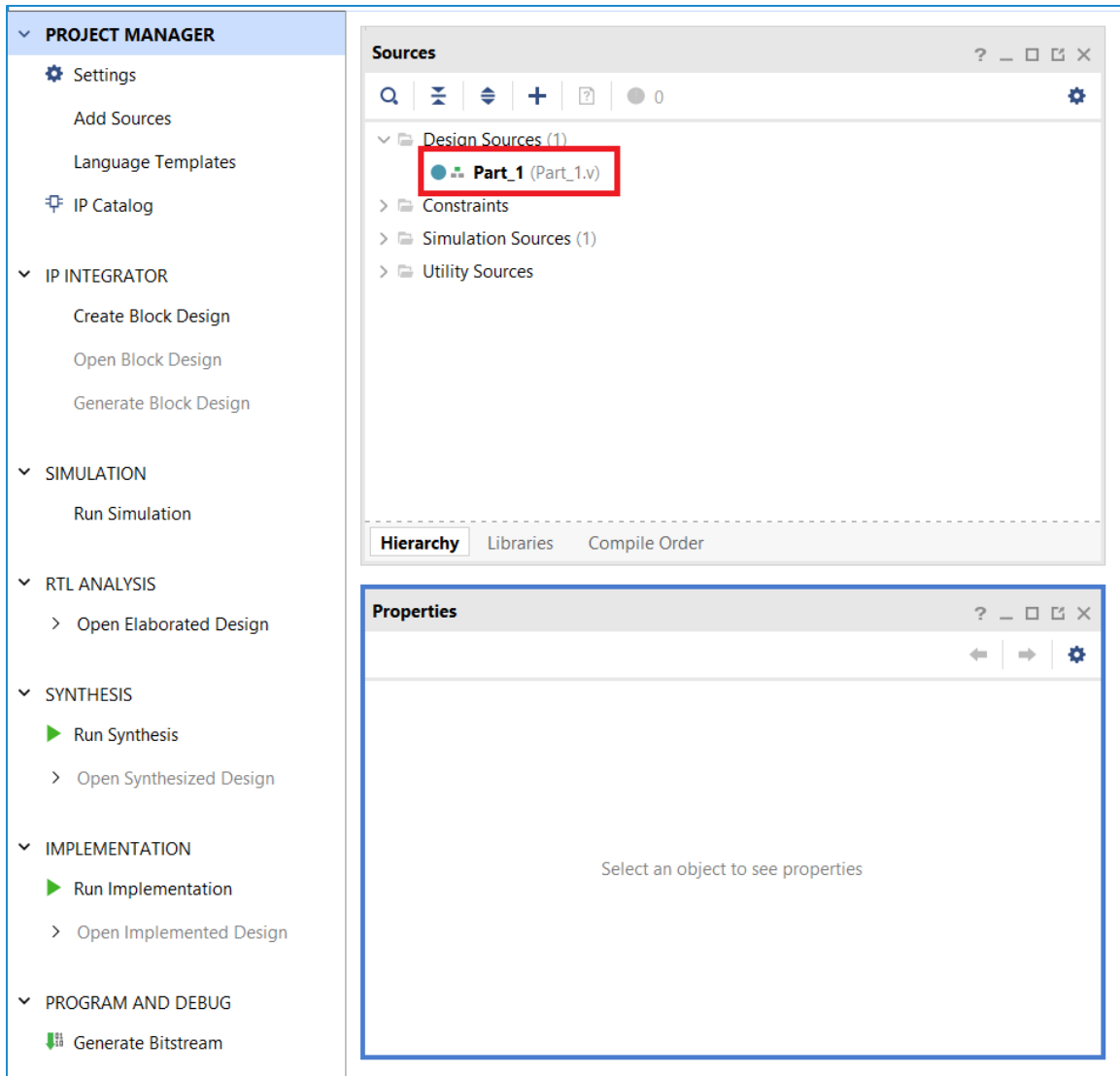


8. The opened window is the main environment for your project that is called “Project Manager”. You can explore it by seeing the options of each category in the toolbar on top of the window. In the left side, you can see the “Settings”, “Add Sources”, “Language Template”, “IP Catalog”, “IP Integrator”, “Simulation”, “RTL Analysis”, “Synthesis”, “Implementation”, and “Program and Debug”. Each of these serves a part of the digital design flow. In the middle, you can see the windows for “Sources”, “Properties”, “Project Summary”, and the reports and summaries for the execution of the project files.



9. Double click on the “Part\_1.v” file (\*.v) in the “Sources” window. The VERILOG source file appears where the window is located in right side. Note that the module shows the defined inputs and outputs that were selected previously.





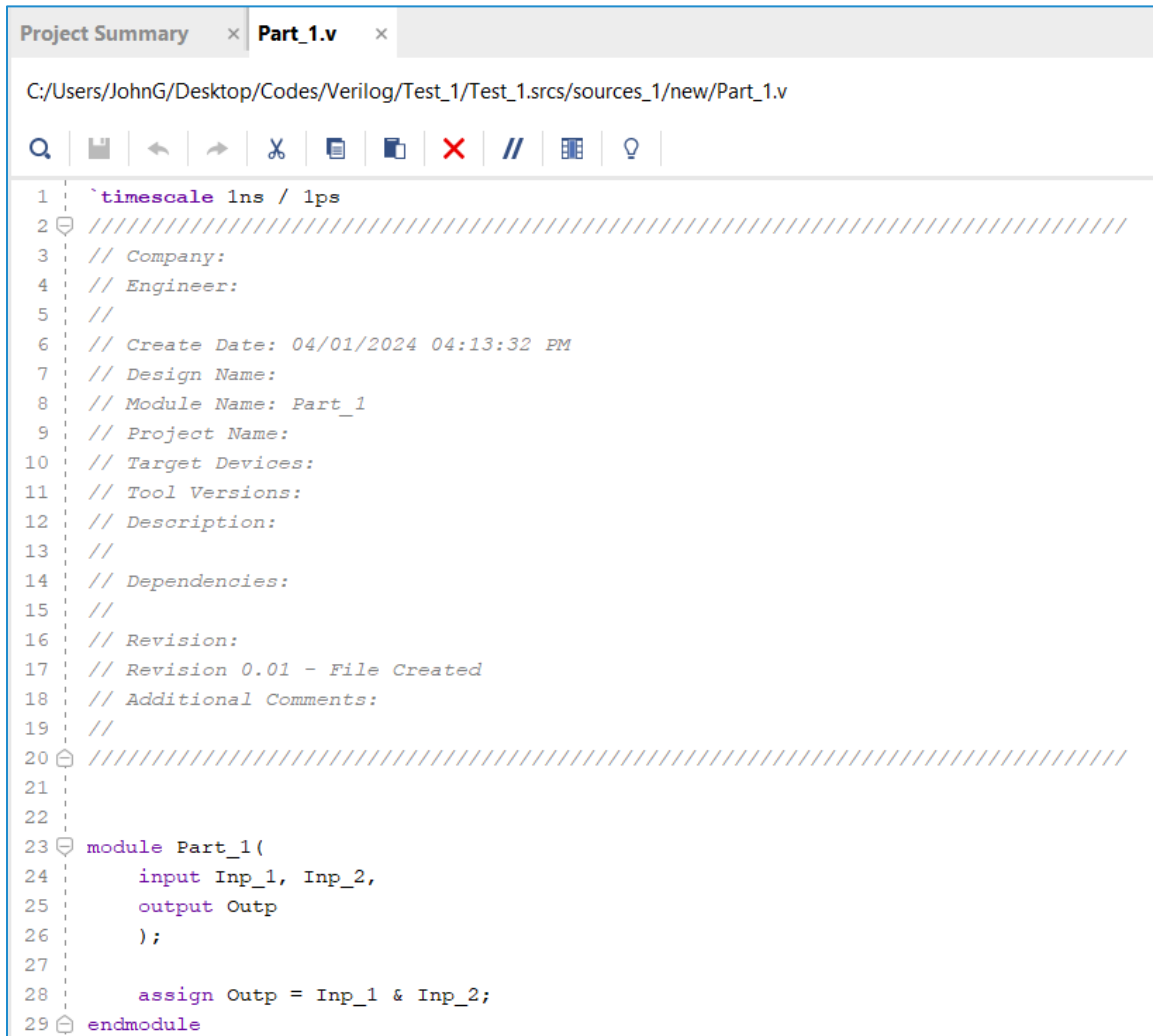
The syntax for VERILOG is very similar to C programming language. All lines must end in a semicolon, and all comments use either `//` or `/* */`. One difference is that all inputs and outputs need two definitions. The first defines if the I/O connection is an input or an output port and the second defines if this input is a "wire" or a register "reg". For this experiment, only wires will be used. Also, to set an output equal to an input the "assign" function must be used (for combinational circuits). A summary of the VERILOG syntax is given in Appendix E. Also, a reference website for Verilog is described as (i.e. <http://www.asic-world.com>). The `'~'` symbol is the **NOT** operator, the `'|'` symbol is the **OR** operator, the `'&'` symbol is the **AND** operator and `'^'` symbol is the **XOR** operator. Add the following lines to the VERILOG program after the input and output definitions:

```

    wire Inp_1, Inp_2, Outp;
    assign Outp = Inp_1 & Inp_2;

```

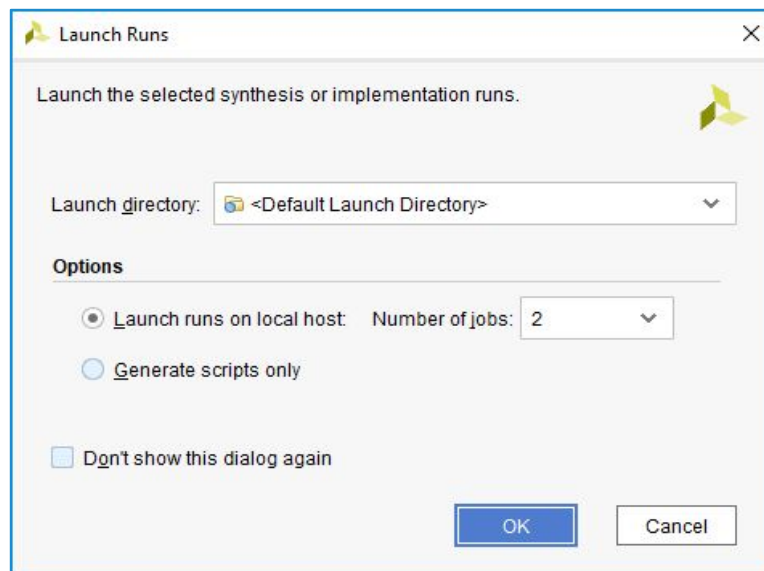
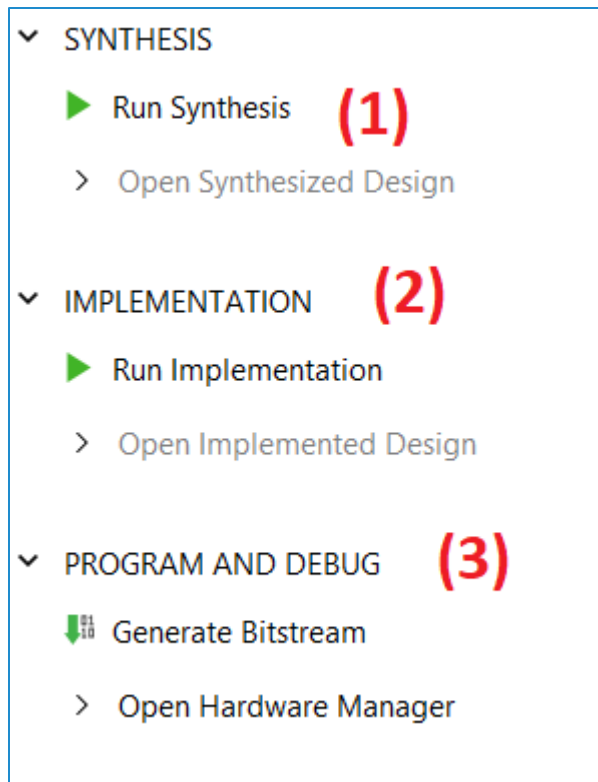
This defines the code required to implement a two-input AND gate:

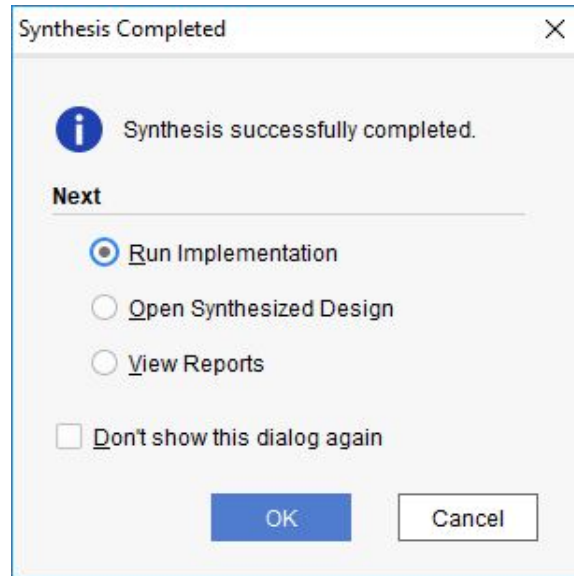


The screenshot shows a Verilog code editor with a tab labeled 'Part\_1.v'. The file path is 'C:/Users/JohnG/Desktop/Codes/Verilog/Test\_1/Test\_1.srscs/sources\_1/new/Part\_1.v'. The code defines a module 'Part\_1' with two inputs, 'Inp\_1' and 'Inp\_2', and one output, 'Outp'. The output is assigned the value of 'Inp\_1 & Inp\_2'. The code is as follows:

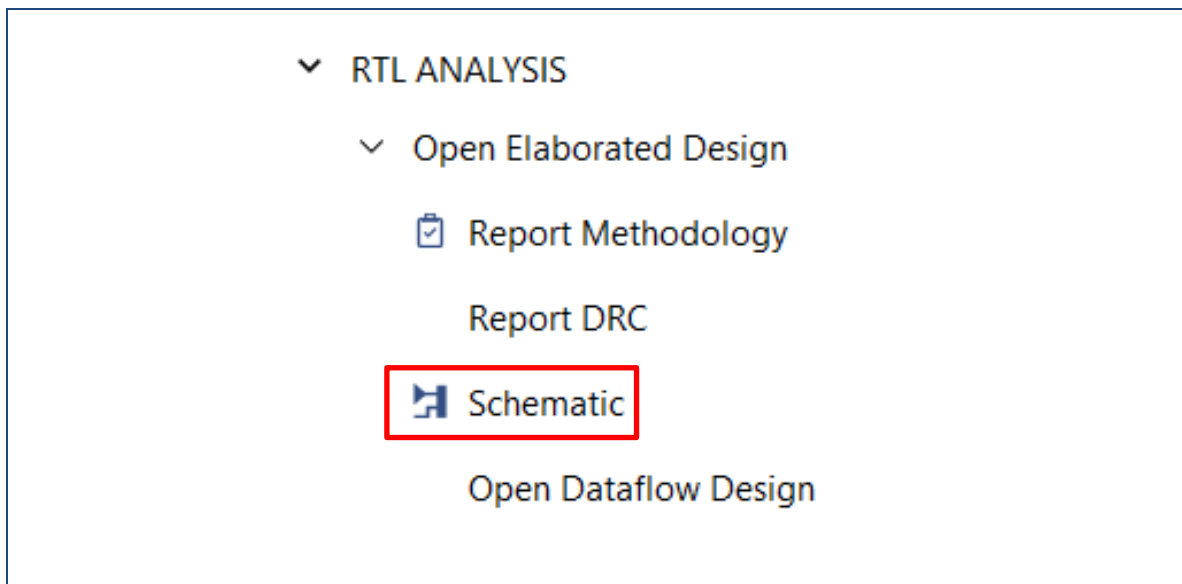
```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 04/01/2024 04:13:32 PM
7  // Design Name:
8  // Module Name: Part_1
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////
21
22
23 module Part_1(
24     input Inp_1, Inp_2,
25     output Outp
26 );
27
28     assign Outp = Inp_1 & Inp_2;
29 endmodule
```

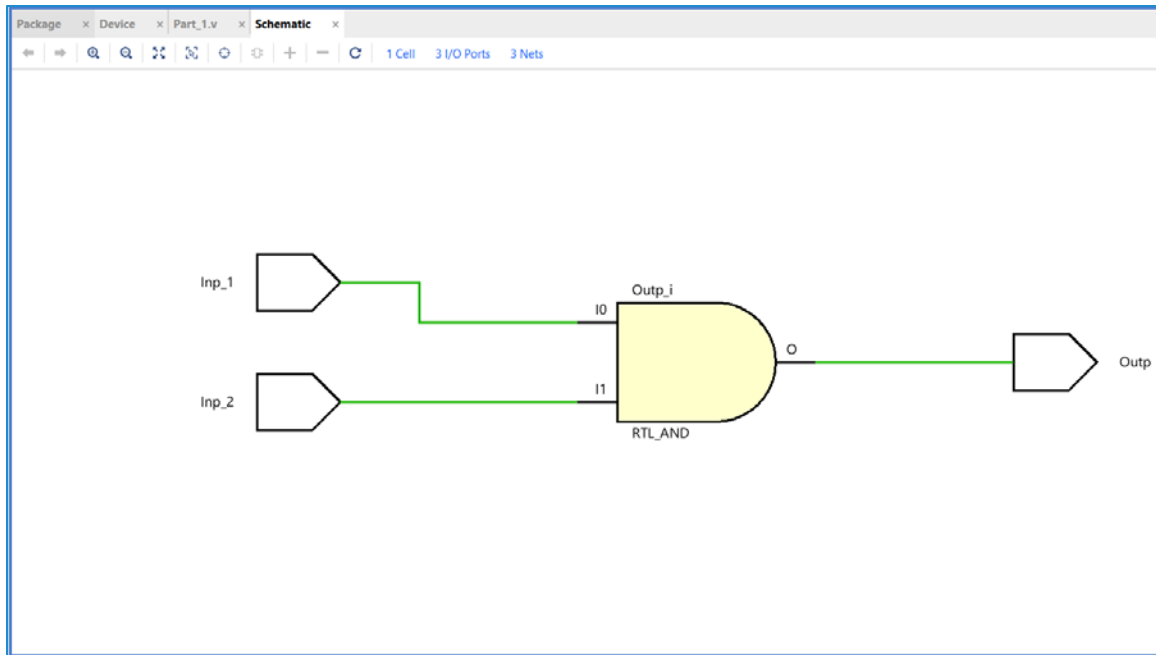
Now that the design is finished, you must build the project. Click Run Synthesis (1) on the left hand menu towards the bottom, on successful completion click on Run Implementation. On successful completion of implementation click on “Open Implemented Design” in the dialog box that appears. Proceed to the next steps.”



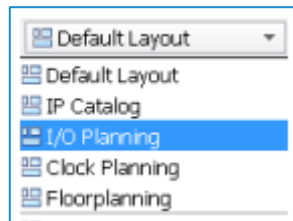


10. We do the “RTL Analysis”. Expand the Open Elaborated Design entry under the RTL Analysis tasks of the Flow Navigator pane and click on Schematic. The model (design) will be elaborated and a logic view of the design is displayed. Notice that some of the switch inputs go through gates before being output to LEDs and the rest go straight through to LEDs as modeled in the file.

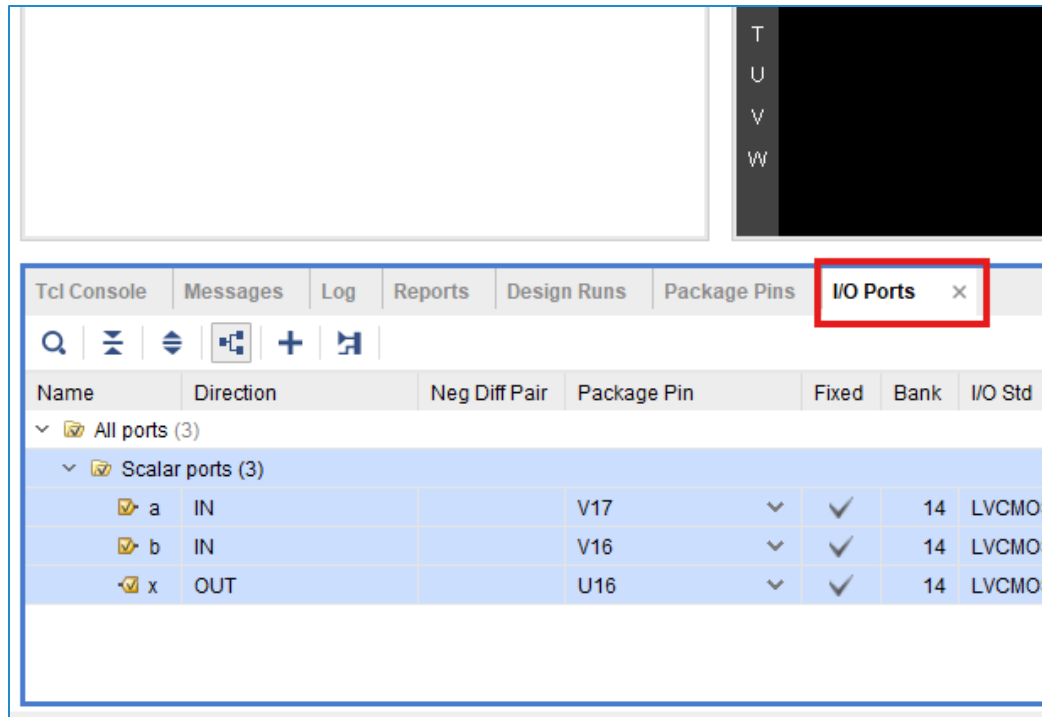




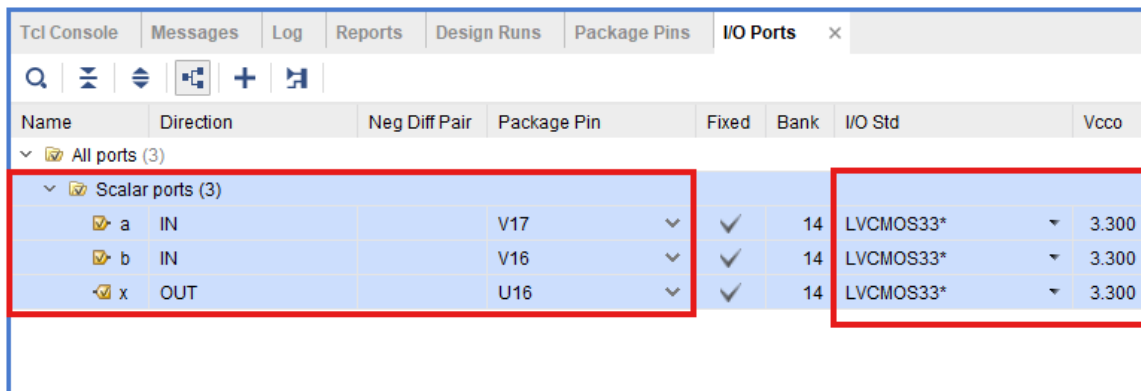
11. Once RTL analysis is performed, another standard layout called the I/O Planning is available. Click on the drop-down button and select the I/O Planning layout.



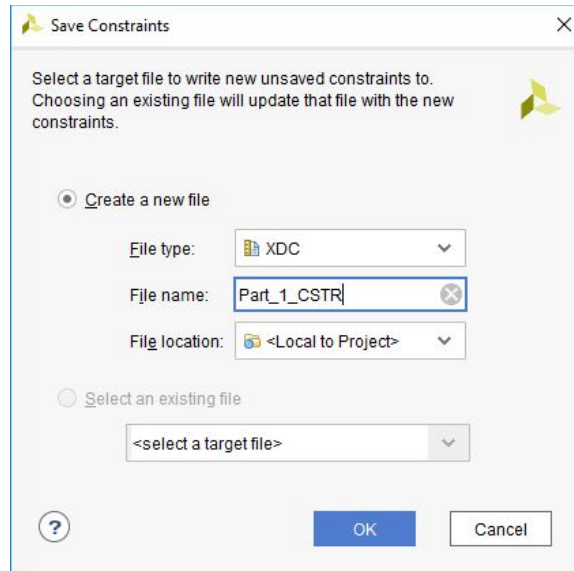
12. Notice that the Package view is displayed in the Auxiliary View area, Device Constraints tab is selected, and I/O ports tab is displayed in the Console View area. Also notice that design ports (led and swt) are listed in the I/O Ports tab with both having multiple I/O standards. Move the mouse cursor over the Package view, highlighting different pins. Notice the pin site number is shown at the bottom of the Vivado GUI, along with the pin type (User IO, GND, VCCO...) and the I/O bank it belongs to.



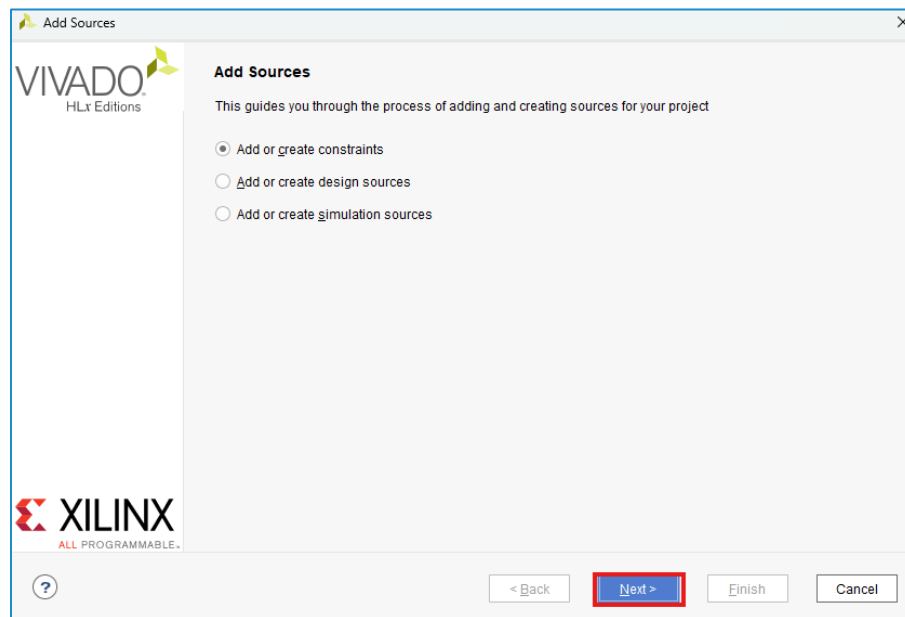
13. The user should simply input the “Site” table next to the input/output to the desired pin. From Appendix D, for BASYS board, SW0 is located on pin V17, SW1 is located on pin V16 and LED0 is located on pin U16. These names can also be found inside the parentheses below the switches and the ones in the right side of the LEDs on the board. The value of I/O Std for all of these switches should be set to “LVC MOS33”.



14. Select File > Save Constraints. Name and save the constraint file.

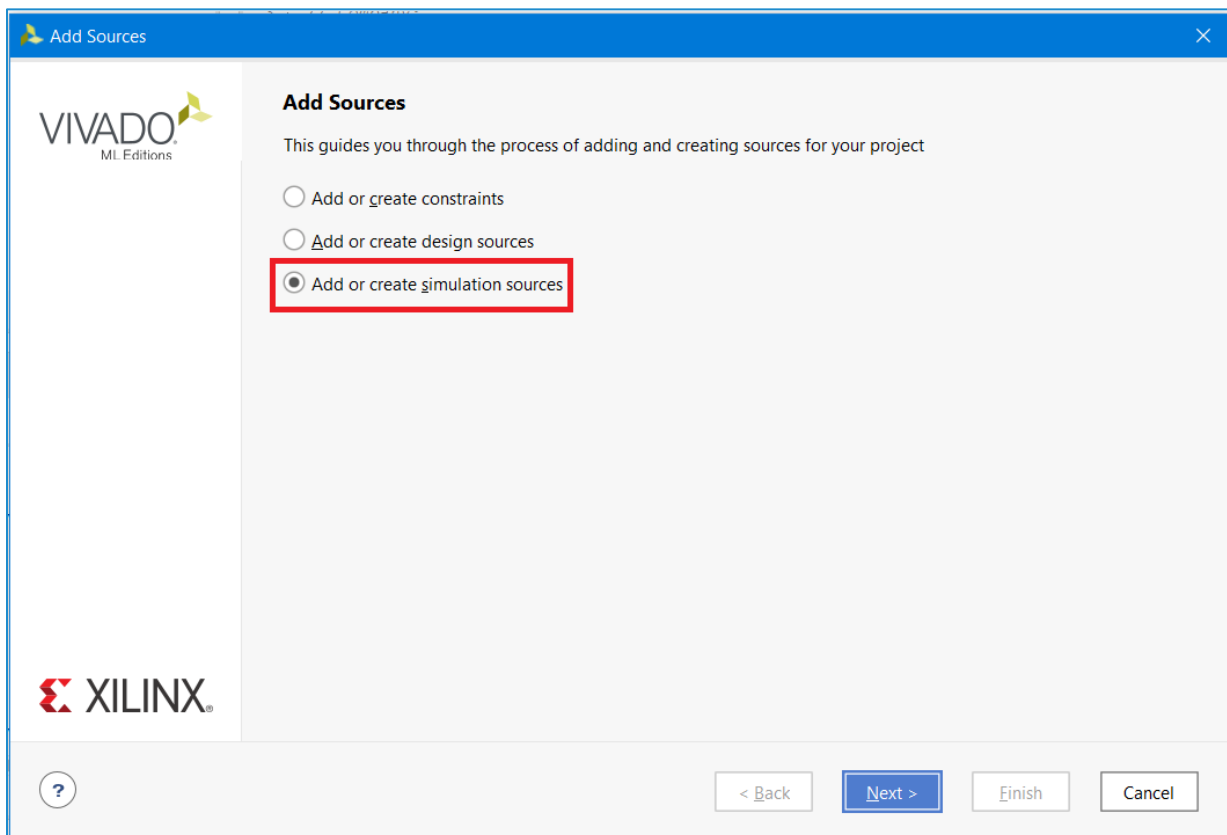


15. Then we should simulate the design using the XSim Simulator. Click Add Sources under the Project Manager tasks of the Flow Navigator pane. Select the Add or Create Simulation Sources option and click Next. We create a new file for simulation, and name it as Part\_1\_Sim.
16. **(OPTIONAL)** Alternatively in part 15, you can create the constraints that will tie your logical outputs to physical components on the Basys3 FPGA via a file. You can create the constraints in the same manner that you create design or simulation files by selecting “Add Sources” on the left navigation bar under Project Manager.

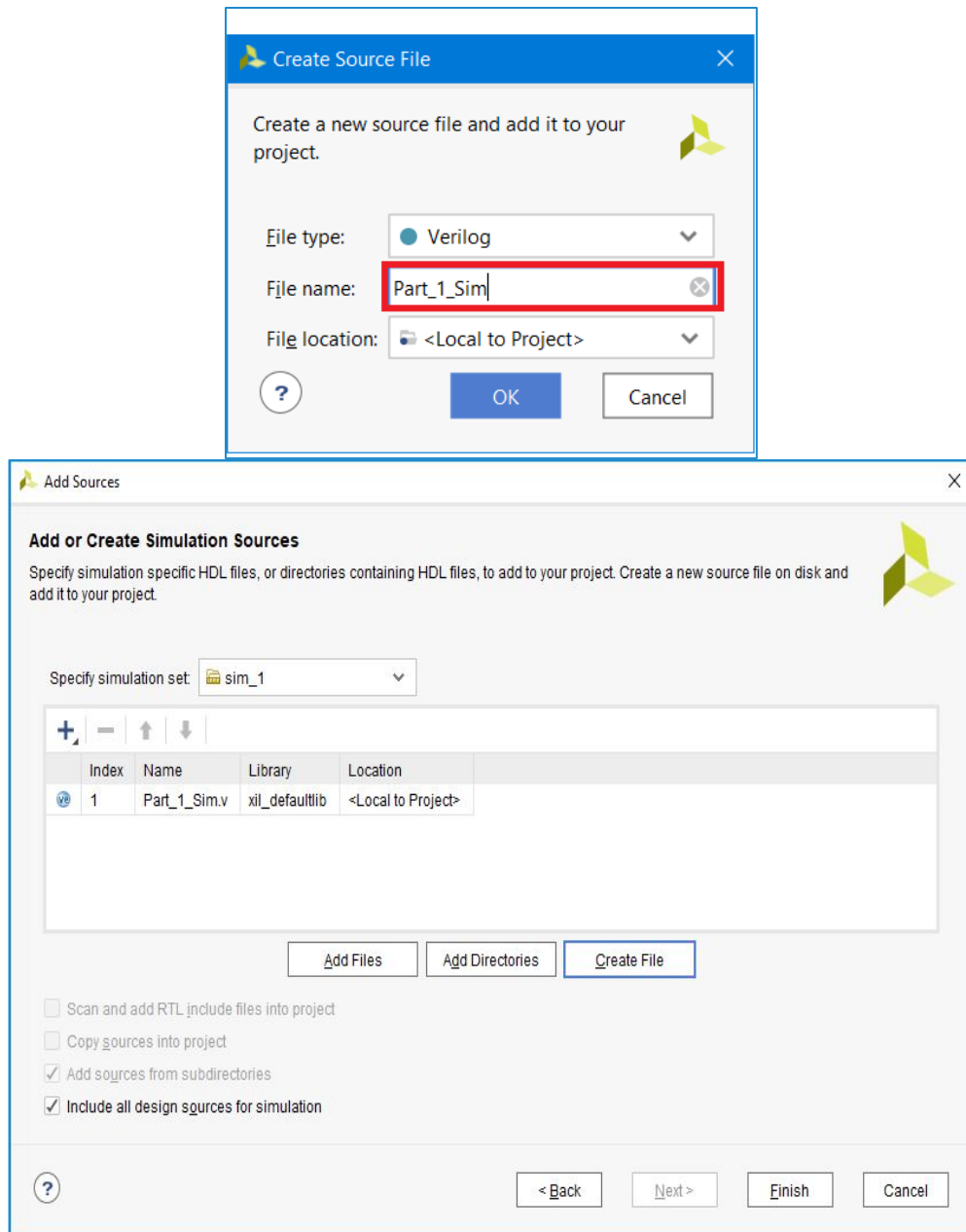


You then can select “Create File” and name it something generic but related to the module like “Module\_1\_CSTR”. You can then copy the line below that are equivalent to the constraint file that was generated from the I/O planning screen. You can access this GitHub repository that has examples of what IOs you can add to any constraint file. <https://github.com/Digilent/digilent-xdc/blob/master/Basys-3-Master.xdc>

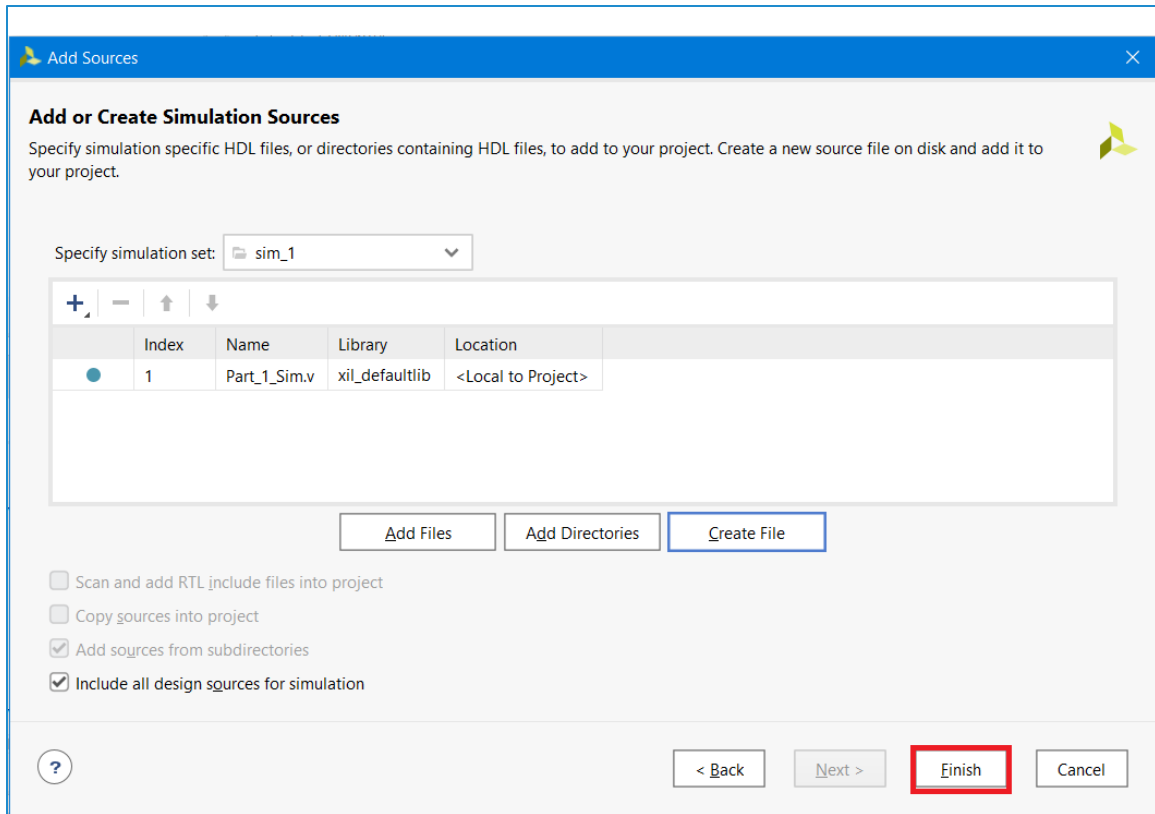
```
1 set_property IOSTANDARD LVCMOS33 [get_ports a]
2 set_property IOSTANDARD LVCMOS33 [get_ports b]
3 set_property IOSTANDARD LVCMOS33 [get_ports x]
4 set_property PACKAGE_PIN V17 [get_ports a]
5 set_property PACKAGE_PIN V16 [get_ports b]
6 set_property PACKAGE_PIN U16 [get_ports x]
7
```







17. For the simulation file, we don't need to set the I/O. Click OK in this step.



18. Double click on the “Part\_1\_Sim.v” in the Sources window to type the following contents into the file:

```

module Part_1_Sim();

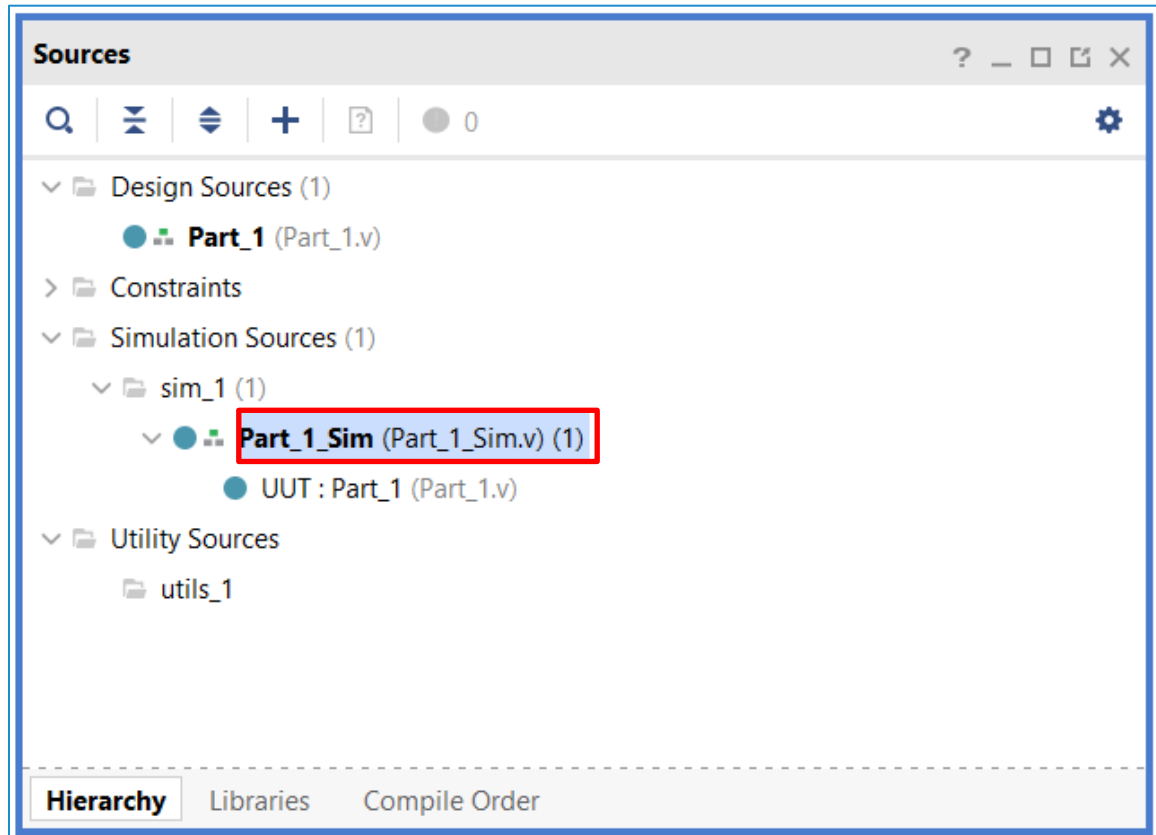
    reg Inp_1, Inp_2;
    wire Outp;

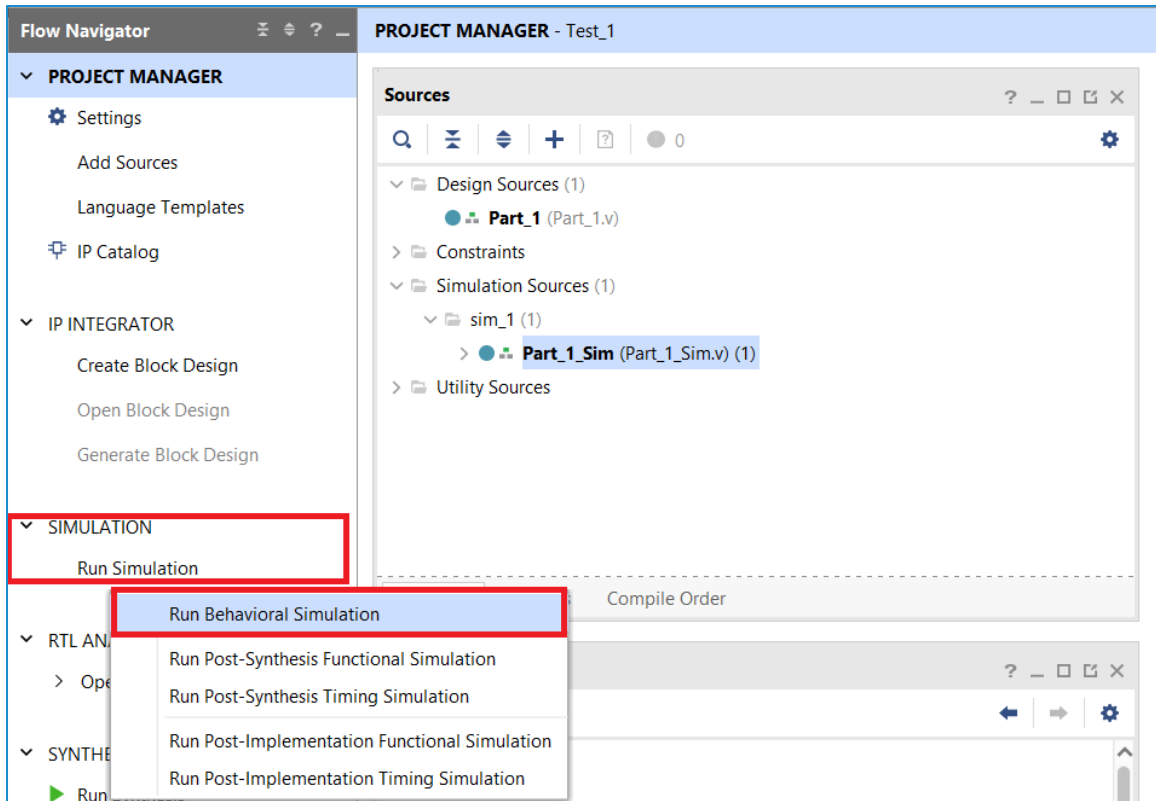
    Part_1 UUT(.Inp_1(Inp_1), .Inp_2(Inp_2),
    .Outp(Outp));

    initial
    begin
        Inp_1 = 0;
        Inp_2 = 0;
        #10
        Inp_1 = 1;
        Inp_2 = 0;
        #10
        Inp_1 = 0;
        Inp_2 = 1;
        #10
        Inp_1 = 1;
        Inp_2 = 1;
    end
  
```

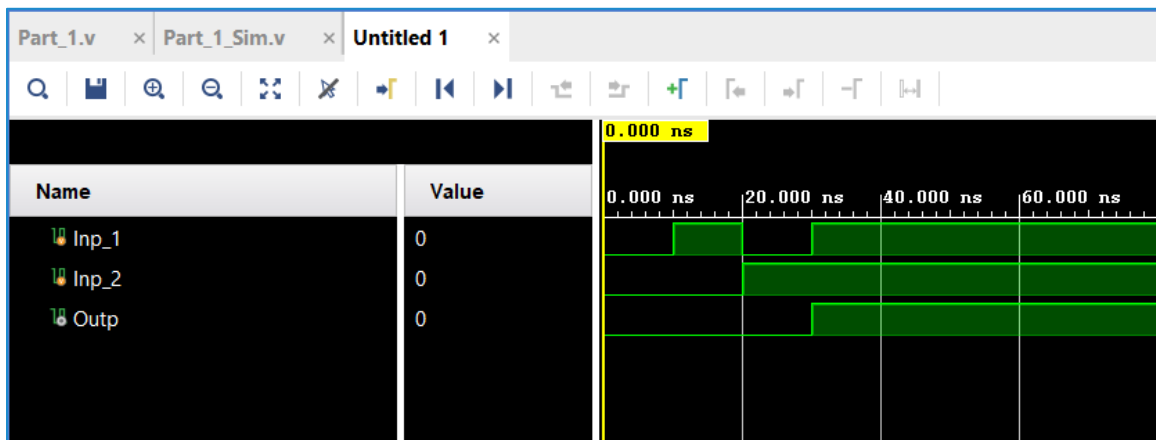
endmodule

*Also, it should be mentioned that the timescale for both the Verilog module and the test-bench module is set to 1 ns/1 ps.*



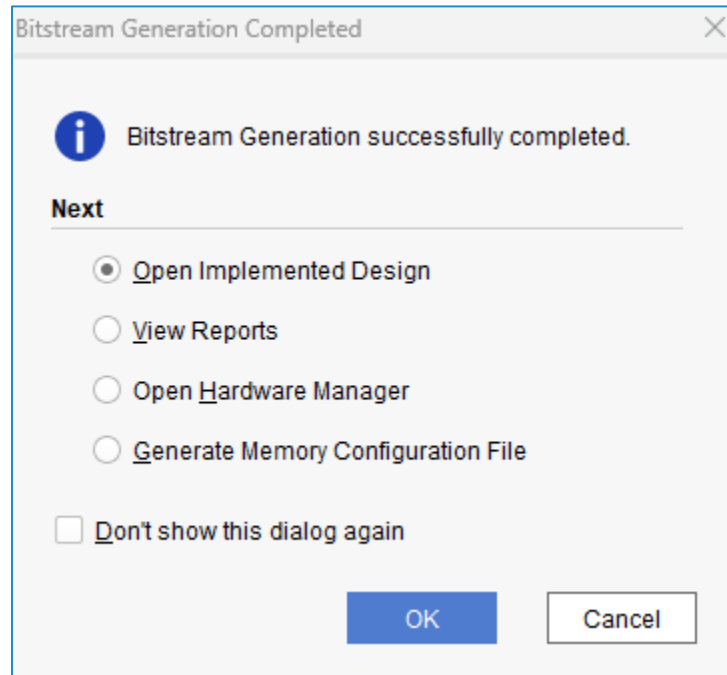


19. Click on Run Simulation > Run Behavioral Simulation under the Project Manager tasks of the Flow Navigator window. The test-bench and source files are compiled and the XSim simulator is run (assuming no errors). Click on the “Zoom Fit” icon to see all the spectrum of simulation.

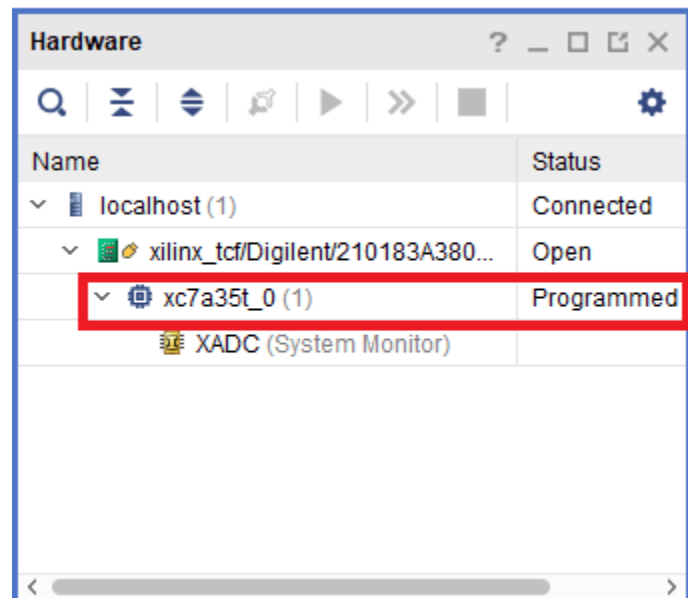
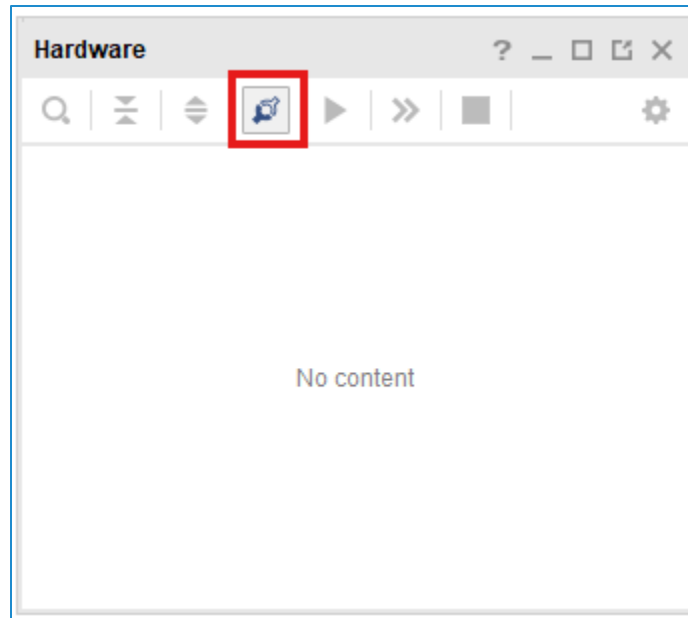


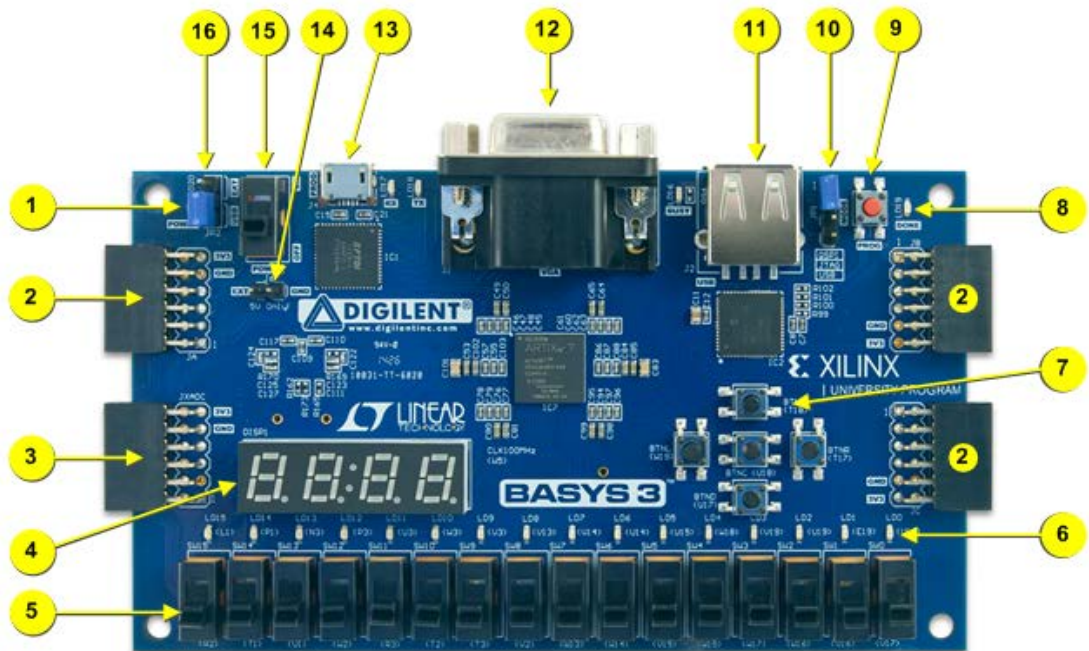
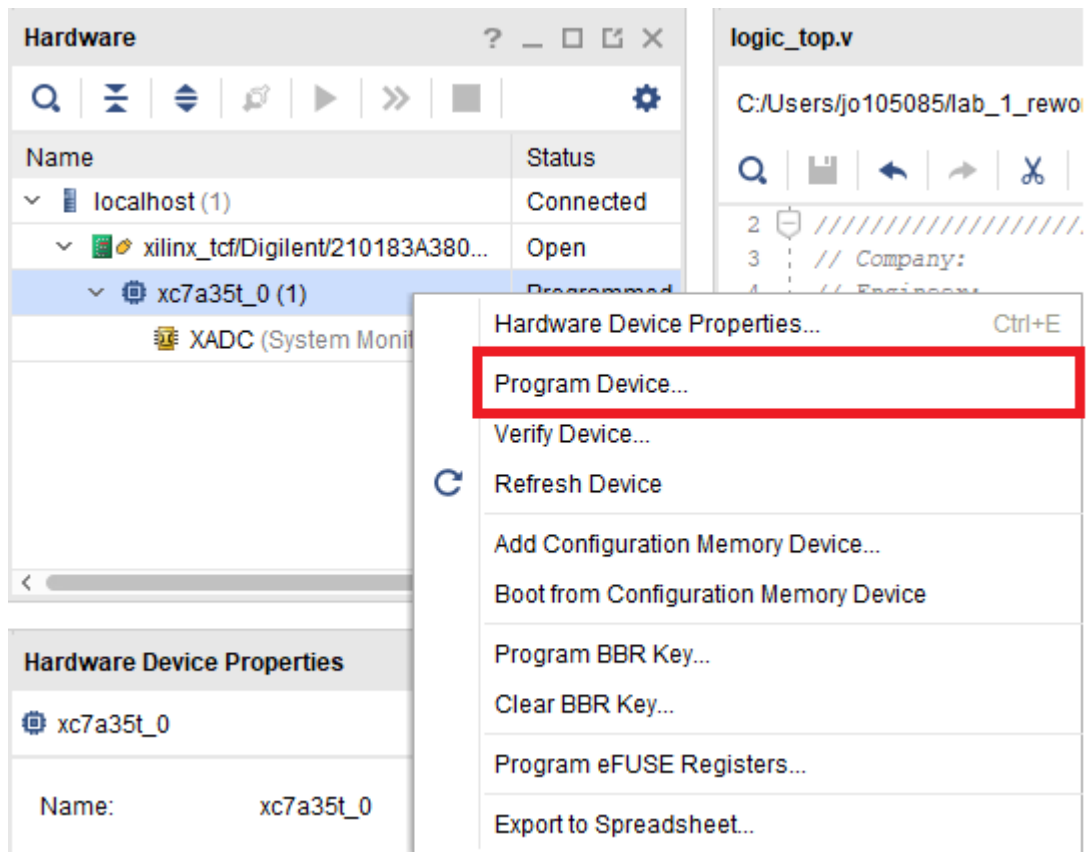
20. In the last part, let’s click on the Generate Bitstream on the left hand menu towards the bottom. Vivado runs through both Run Synthesis and Run Implementation before it

generates the bitstream automatically. This process generates the \*.BIT file needed to program the FPGA. The following window will be opened if the bitstream is generated.



21. Go to “**Flow → Hardware Manager**” in the toolbar. Turn ON your board by pushing up its power switch. Click on “**Auto Connect**” icon in the Hardware Manager window. Click on the board name “**xc7a35t\_0 (1)**”. In the opened “**Hardware Device Properties**” window, make sure the bit file is selected for the “**Programming file**”. Next, right click on the board name and choose “Program Device...”. Once the status of the board goes to “Programmed”, then you can check the design functionality on the board by changing the state of switches.





Callout	Component Description	Callout	Component Description
1	Powergood LED	9	FPGA configuration reset button
2	Pmod connector(s)	10	Programming mode jumper
3	Analog signal Pmod connector (XADC)	11	USB host connector
4	Four digit 7-segement display	12	VGA connector
5	Slide switches (16)	13	Shared UART/JTAG USB port
6	LEDs(16)	14	External power connector
7	Pushbuttons (5)	15	Power Switch
8	FPGA programming done LED	16	Power Select Jumper

22. Check the operation for the two-input AND gate and fill out the following table. Remember we have selected switch SW0 for input “Inp\_1”, switch SW1 for input “Inp\_2” and led LED0 for the output “Outp”. Toggle the switches for the states shown in the table below and fill in the output by observing LED0. This table should confirm the truth table for a two-input AND gate.

SW0	SW1	LED0
0	0	
0	1	
1	0	
1	1	

## Verilog Basics:

**Regs:** The reg components are units that have the responsibility of holding the status of any data in the form of bits. It can change throughout the architecture so when you are working with regs you must consider that you can change it (i.e. assign it in two places at the same time) during its processing time otherwise you’ll get what’s called a “multi-driven net” error. There will be an example of a multi-driven net error when we discuss always blocks.

**Syntax:**  
`reg inp_1;`

**Wires:** The wire components are used to pass data along to different regs or modules throughout the hardware architecture. These elements do not store any data but can pass it along, think of them as an actual wire that you would connect components to on a breadboard, they act the same way but are directional based on how you define its behavior.

**Syntax:**



```
wire w1;
```

**Instantiations:** Instantiation is the process of creating instances of other modules within another module allowing an engineer to reuse a hardware design multiple time. This process is crucial for hardware design as it allows the reuse of modules to function as gates, flip-flops, and more complex functionalities. In the testbench provided in the “**Design Constraints**” we instantiated our lab\_1\_top and named that block UUT (Unit Under Test) and we can use this principle within the design space as well. This is most similar to a function call in programming languages but again we must think that these are physical hardware blocks that are physically connected.

**Syntax:**

```
module_name test_name(.module_input(my_input), ...);
```

**Example:**

```
module AND_gate(  
    input a, b  
    output c)  
  
    assign c = a & b;  
  
endmodule  
  
module top_module(  
    input a, b, c, d,  
    output x, y, z  
)  
  
    AND_gate and1(.a(a), .b(b), .c(x))  
    AND_gate and2 (.a(c), .b(d), .c(y))  
    AND_gate and3 (.a(x), .b(y), .c(z))  
  
endmodule
```

**Initial Blocks and Begin/End Statements:** The initial block is used to initialize or set the values of regs such as to test a variety of inputs or force a desired signal. The begin/end statements are used as a blocking statement that tell where a group of RTL code will begin on a certain condition and where it will end. The main idea of using the initial blocks is to give a reference point of how certain inputs will behave instead of having no initial value.

**Syntax:**  
`initial begin`  
`Inp_1 = 0;`  
`Inp_2 = 1;`  
`end`

**Vectors:** A vector in Verilog is a reg or wire that has a size larger than a single bit. This allows for more complex and creative ways to manipulate digital information within the confines of a hardware circuit. We can then take a wire and turn it into cluster of wires (bus), which is wires bundled together to pass larger data around. We can say that a given reg or wire that is a vector has a bit-width of whatever the designated vector size is.

**Syntax:**  
`//bit-width of 3`  
`reg[2:0] Inp_1`  
`wire[2:0] w1`

**Concatenation:** Within the scope of Verilog, a concatenation is when you can combine wires to populate or form a vector. This technique is used to collect information and store it within a reg or use it for condition handling for logic selection. For instance, if we had a vector that is 3-bits in length and we had 3 inputs that were 1-bit in length, we could combine the three inputs and populate our vector with the single-bit inputs.

**Syntax:**  
`//bit-width of 3`  
`// A, B, and C are 1-bit inputs`  
`wire[2:0] out_1 = {A, B, C}`  
`// out_1[2] = A, out_1[1] = B, out_1[0] = C`

## **Part 2. Implementation of the NAND, OR, XOR, and NOT GATES using Xilinx's Vivado**

1. Repeat the steps in part one of this experiment, but this time for a two-input NAND gate (NAND2). Open a New Project, do not try to include a new schematic file in the previous project.
2. Repeat the steps in part one of this experiment, but this time for a two-input OR gate (OR2). Open a New Project, do not try to include a new schematic file in the previous project.
3. Repeat the steps in part one of this experiment, but this time for a two-input XOR gate (XOR2). Open a New Project, do not try to include a new schematic file in the previous project.

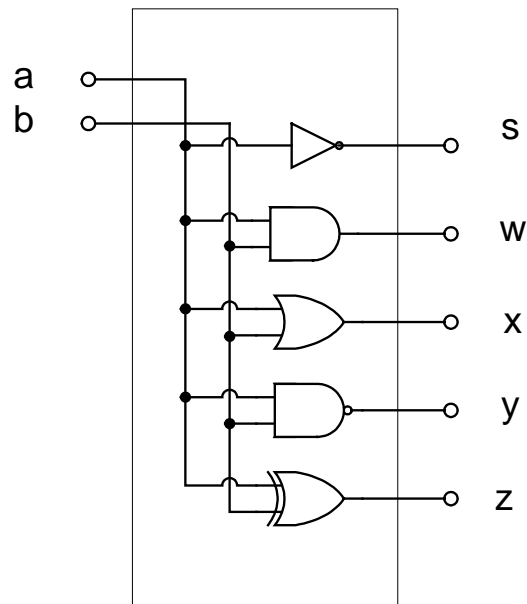
- Repeat the steps in part one of this experiment, but this time for an inverter gate (NOT). Open a New Project, do not try to include a new schematic file in the previous project. Use this truth table for the inverter circuit.

SW0	LED0
0	
1	

### Part 3. Implementation of a two-input five-output logic circuit.

- Implement the two-input and five-output logic circuit using the described steps. Simulate the five-outputs for all possible inputs. Upload the finished design into the BASYS board and verify its functionality using the following table.

A	B	LED0 (S)	LED1(W)	LED2(X)	LED3(Y)	LED4(Z)
0	0					
1	0					
0	1					
1	1					



**Important:** It is recommended to follow the procedures in this laboratory manual for writing the report for this experiment.

- Summarize in your own words the steps required to complete every part of this experiment. Include the screen-shots (e.g. alt + print-screen to place a screen into

the clipboard) of the process in your report to show the steps taken to complete this experiment.

2. Include and discuss the simulated results.
3. Give the truth table.
4. Follow the explanations in the Introduction section of this manual for details of how to write a well-structured report.
5. Look at the announcements in the course webpage for more information regarding this lab and what items to position in your report.

---

## EXPERIMENT #2

### Introduction to Testbenches

---

#### Part 1) Brief Explanation of Testbenches

Testbenches are a key component in the realm of digital design that allow you to verify that your hardware design is working as intended. This can be performed by writing a Verilog testbench file that will instantiate the hardware module you are wishing to test. Then you will need to set the **inputs** of the module to any value that would adequately test the module by monitoring the behavior of the **outputs** via waveforms in Vivado.

From the previous lab, we will be using the **initial** block to initialize the values of the inputs of the module. Note, that all input signals will need to be declared **regs** and output signals **wires** because the values of the inputs will need to change, and the outputs will only be read from.

For example, we if wanted to verify the output of an AND gate module that was designing then our testbench would look like the following:

```
module and_tb();  
  
    reg a, b;  
  
    wire y;  
  
    and UUT(.a(a), .b(b), .y(y));  
  
    initial begin  
        a = 1'b0;  
        b = 1'b0;  
  
        #10; // Delay by 10ns  
  
        a = 1'b1;  
        b = 1'b0;  
  
        #10;  
  
        a = 1'b0;  
        b = 1'b1;  
  
        #10;  
  
        a = 1'b1;  
        b = 1'b1;  
  
    end  
endmodule
```

```

        #10;
    end
endmodule

```

It's easy to see that typing out all of the permutations of the inputs of a and b is not a scalable approach. In any case of cycling through permutations we can utilize a for-loop and concatenation to go through any number of permutations. Below is the source to do so:

```

module and_tb();
parameter NUM_INP = 2;
reg a, b;
reg[NUM_INP-1:0] cnt;
wire y;
integer i;
and UUT(.a(a), .b(b), .y(y));
initial begin
    cnt = 0;
    for(i = 0; i < 2**NUM_INP; i = i + 1)begin
        a = cnt[1];
        b = cnt[0];
        cnt = cnt + 1;
        #10;
    end
end
endmodule

```

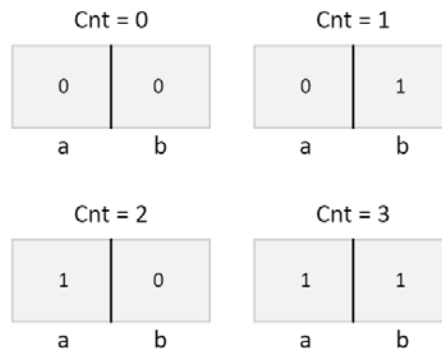


Figure 1: Exhaustive Testbench Logic

The figure above shows that as **cnt** increases, its bits are concatenated into **a** and **b** iterate through all the possible values of them. This can be applied to more variables, but you will need to add them into the for loop to concatenate the proper value and increase the NUM\_INP parameter to the number of inputs you have. All things considered, this type of testbench code is good when you want to check all permutations of a given systems which can be seen as *exhaustive testing*. Depending on the situation, you can have a *targeted testbench* which is when you have a specific value you want to test rather than all possible values. We want to not just write a testbench, but we want to write a **good testbench** so analyze the problem at hand and what you are being tasked with and design the testbench accordingly.

Some small details to pay attention to would be how the module is defined and the # symbols. Regarding the module creation, there are no signals tied to it as we are not creating any hardware for the testbench to use. The # **symbol** is used to introduce delay, in nanoseconds (ns), into the testing environment which allows for a larger amount of control for the given test and better organization of the generated waveform when changing the values of signals. Typically, you always want to put a delay when you are about to change a signal so that we can better analyze the behavior of hardware modules as signals change.

## Part 2) Simple Boolean Network Testbench

During this section of the lab, you will need to create the simple RTL for the digital logic circuit below. In addition, you will need to create a test bench file that will test *every permutation* of the inputs to give the correct outputs.

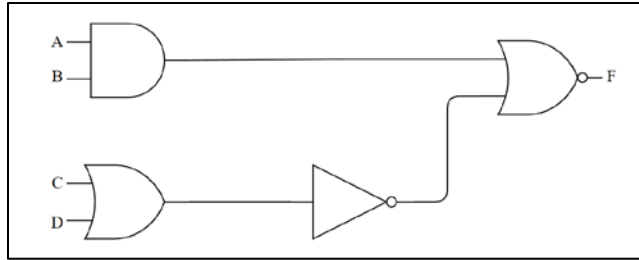


Figure 2: Simple Digital Logic Circuit

**Deliverables:**

- RTL code
- Test bench code
- Boolean equivalent equation of circuit
- Truth table
- Screenshot of Vivado waveform

**Part 3) Complex Boolean Network Testbench**

During this section of the lab, you will need to create the complex RTL for the digital logic circuit below. In addition, you will need to create a test bench file that will test *every permutation* of the inputs to give the correct outputs.

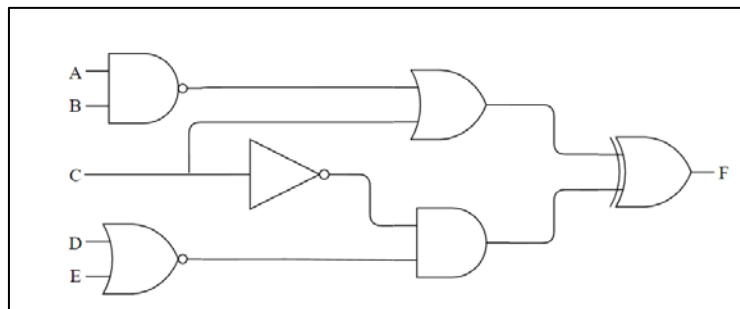


Figure 3: Complex Digital Logic Circuit

**Deliverables:**

- RTL code
- Boolean equivalent equation of circuit
- Test bench code
- Truth table
- Screenshot of Vivado waveform



#### Part 4) Application: Circuit Debugging

You are an entry level hardware verification engineer at a semiconductor firm and a design engineer has requested that you create a test bench for their design. They have warned you that there are *possible bugs* within the RTL code that they have created. You will be provided a table that notates the expected outputs correlated with certain inputs (truth table) and you must write a test bench file based on the design that is provided. The overall task is to identify the bug in the code, fix it, and rerun the simulation to verify that the waveform matches the expected outputs of the given truth table. Once the bug is fixed, draw out the digital logic diagram either by hand or using a website to make the circuit.

#### Helpful Boolean Simplification Laws for Part 4:

$$A'B' + AB = (A \oplus B)'$$

$$A'B + AB' = A \oplus B$$

**Note:** W1, W2, and W3 are guaranteed to be true

#### Truth Table of Expected I/O:

Inputs			Outputs	
<u>A</u>	<u>B</u>	<u>C</u>	<u>X</u>	<u>Y</u>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

### Provided RTL code from design engineer:

```
module logic_top(  
    input A, B, C,  
    output X, Y  
);  
  
    wire w1, w2, w3;  
  
    assign w1 = A ^ B;  
    assign w2 = C & w1;  
    assign w3 = A & B;  
    assign X = C | w1;  
    assign Y = w2 & w3;  
  
endmodule
```

### Deliverables:

- Updated RTL code
- Briefly explain the bug that was found
- Test bench code
- Picture of Vivado waveform **before** debugging
- Picture of Vivado waveform **after** debugging
- Drawing of logic diagram of updated RTL code

### Overall Lab Deliverables Q&A:

1. What type of test bench would you write if you wanted to test all possible permutations?
2. What type of test bench would you write if you wanted to test a specific input?
3. When writing a test bench, what Verilog data type should inputs and outputs be for modules you are testing?
4. What is the purpose of putting a delay into a test bench? Explain your reasoning.
5. Is the test bench a piece of hardware generated by Vivado? Explain your reasoning.
6. Complete all listed deliverables for each section.

**Extra Credit:**

1. What is the intended behavior of the module in Part 4?

---

## EXPERIMENT #3

### Boolean Simplification & Digital Circuit Area Cost

---

#### Part 1) Boolean Simplification Application and Laws

Boolean simplification is a critical theory that allows hardware engineers and researchers to reduce the amount of physical hardware needed to perform any specific hardware task. It's important to be aware of when building your design as it builds the mental foundations on how to **build an optimal design for the fewest number of resources available**. Most RTL design tools, like Vivado, handle a large amount of Boolean simplification for us.

If we can reduce the number of gates for the same purpose, then we can save the total number of transistors on a chip allowing us to include features that can improve the overall speed of a CPU or GPU by modifying the hardware architecture. This concept is known as *designing for area*, to be specific, each logic gate that we design in RTL will eventually be laid out onto a computer chip as a transistor.

For your reference there will be a table below that will review the basic Boolean laws that will be used throughout this lab:

Law Name	AND Laws	OR Laws
Identity Law	$1A = A$	$0 + A = A$
Null Law	$0A = 0$	$1 + A = 1$
Idempotent Law	$AA = A$	$A + A = A$
Inverse Law	$AA' = 0$	$A + A' = 1$
Negation Law	$(A')' = A$	-----
Commutative law	$AB = BA$	$A + B = B + A$
Associative Law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive Law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption Law	$A(A + B) = A$	$A + AB = A$
DeMorgan's Law	$(AB)' = A' + B'$	$(A + B)' = A'B'$
XOR Law	$A \otimes B = A'B + AB'$	-----
XNOR Law	$(A \otimes B)' = AB + A'B'$	-----

The two general forms of a Boolean equation that can be interpreted are Sum of Products (SOP) and Products of Sums (POS). SOP is a way to represent a Boolean equation in the form of ORs (sums) of ANDs (products), whereas POS is in the form of ANDs (products) of ORs (sums). For example, the equation  $AB + A'B$  is an SOP, and the equation  $(A+B)(A'+B)$  is in POS form.

The outputs of a truth table dictate if the inputs that drive the given output should be in SOP or POS form. Specifically, if the output is a high signal (logical 1), the inputs that caused the high signal should be written in SOP form. Alternatively, if the output is a low signal (logical 0), the inputs that caused the low signal should be written in POS form. To properly create a term in SOP, you must AND all the input signals based on their value that drove the logical high output. If the input signal is high, represent it as normal, but if it is low then invert the signal. Lastly, you must OR all SOP terms generated based on high output signals. To properly create a term in POS, you must OR all the input signals based on their value that drove the logical low output. If the input is low, represent it as normal, but if it is high then invert the signal. Lastly, you must AND all POS terms generated based on the low output signals.

Inputs		Outputs
A	B	F
0 (A')	0 (B')	1 (SOP term 0)
0 (A)	1 (B')	0 (POS term 0)
1 (A)	0 (B')	1 (SOP term 1)
1 (A')	1 (B')	0 (POS term 1)

POS Form:  $(Term_0)(Term_1) \rightarrow (A + B')(A' + B')$

SOP Form:  $Term_0 + Term_1 \rightarrow A'B' + AB'$

**Answer the following questions:**

1. Simplify the following Boolean equations:

$$EQ1: AB'C + A'BC' + A'B'C$$

$$EQ2: ABC + AB'C + A'BC' + A'B'C$$

2. Give the POS and SOP Representation of this truth table:

Inputs			Outputs
A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

### Deliverables for Part 1)

- Question 1) solution with steps
- POS and SOP form for Question 2) truth table

### Part 2) 3-input Boolean Circuit Simplification using Laws

In this section of the lab you will be provided with a truth table that you will need to simplify. You must use Boolean simplification laws to solve this problem and name each law that you use. Once you have simplified the Boolean equation, create the Verilog RTL representation of the Boolean circuit both before and after simplification. In a single waveform, show that the outputs of both circuits are equivalent. Lastly, provide the schematics of both circuits and note how many logic gates you saved with the optimized design.

Inputs			Outputs
A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

### Deliverables for Part 2)

- Verilog RTL code
- Screenshot of waveform
- Screenshot of schematic before & after simplification
- Simplified Boolean equation with steps

### Part 3) 4-input Boolean Circuit Simplification using Laws

In this section of the lab you will be provided with a truth table that you will need to simplify. You must use Boolean simplification laws to solve this problem and name each law that you use. Once you have simplified the Boolean equation, create the Verilog RTL representation of the Boolean circuit both before and after simplification. In a single waveform, show that the outputs of both circuits are equivalent. Lastly, provide the schematics of both circuits and note how many logic gates you saved with the optimized design.

Inputs				Outputs
A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

### Deliverables for Part 3)

- Verilog RTL code
- Screenshot of waveform
- Screenshot of schematic before & after simplification
- Simplified Boolean equation with Steps

### Part 4) Boolean Simplification using K-Maps (Get equation -> Make circuit)

In this section of the lab you will be using a Karnaugh-map (K-Map) to find the simplified Boolean equation from the truth table below in both SOP and POS form. Once you have simplified the Boolean equation create the Verilog RTL representation of the Boolean circuit for POS and SOP. In a single waveform, show that the outputs of both circuits are equivalent. Lastly, provide the schematics of both circuits and note how many logic gates you saved with the optimized design.

Inputs				Outputs
A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0



#### **Deliverables for Part 4)**

- Verilog RTL code
- Screenshot of waveform
- Screenshot of schematic for SOP simplification
- Screenshot of schematic for POS simplification
- Simplified Boolean equation for POS and SOP
- K-Map used to simplify the circuit

#### **Part 5) Application: Reduce Transistor Count for a Digital System**

You are a physical design engineer tasked with reducing the amount transistors used for a part of a CPU design. The hardware engineer was responsible for designing a 4-input system. They are struggling to simplify the design based on the following requirements:

**4-inputs: A, B, C, D**

**1-Based Outputs for Input Numbers:**

3, 7, 8, 9, 10, 13

**0-Based Outputs for Input Numbers:**

1, 2, 5, 6, 11, 15

**Don't Care Outputs for Inputs Numbers:**

4, 12, 14, 0

**Ex:**

Input 0)  $A = 0, B = 0, C = 0,$  and  $D = 0$

Input 1)  $A = 0, B = 0, C = 0,$  and  $D = 1$

Input 2)  $A = 0, B = 0, C = 1,$  and  $D = 0$

Input 3)  $A = 0, B = 0, C = 1,$  and  $D = 1$

Input 4)  $A = 0, B = 1, C = 0,$  and  $D = 0$

Input 5)  $A = 0, B = 1, C = 0,$  and  $D = 1$

Input 6)  $A = 0, B = 1, C = 1,$  and  $D = 0$

Input 7)  $A = 0, B = 1, C = 1,$  and  $D = 1$

...

Above shows the value of each input, A, B, C, or D, and what input number it represents. The Don't Cares within a digital system represent an output that isn't relevant to the overall functionality of a Boolean expression. Within a K-Map a Don't Care can be written as a "X" and you can utilize them for SOP and POS for simplification. Based on your knowledge of Boolean simplification, generate the POS and SOP simplified versions of the expected outputs and determine which form produces the least number of gates after simplification. Write the Verilog code of the simplified Boolean system for each form while providing the waveforms that prove that they are equivalent to each other and the original design. It is recommended that you use a K-Map for this problem.

### **Deliverables for Part 5)**

- Verilog RTL code
- Screenshot of waveform
- Screenshot of schematic for SOP simplification
- Screenshot of schematic for POS simplification
- Simplified Boolean equation for POS and SOP
- K-Map used to simplify the circuit

### **Overall Deliverables and Q&A**

- What form of Boolean equations is an ANDs of ORs?
- What form of Boolean equations is an ORs of ANDs?
- How many logic gates comprise a 3 variable term?
- What does a logic gate represent within the physical realm of computers?
- Complete all deliverables for each section of this lab.

---

## EXPERIMENT #4

### Hardware Applications of Binary Adders

---

#### Part 1) Hardware Adders: The Half/Full Adder

Binary addition is a fundamental process that every computer needs to be able to complete to handle any simple task. This lab will teach you how to build the hardware for binary addition. A hardware adder is complete in two steps: collecting the sum from the two bits and then generating a carry value to be used by the following addition of bits. Getting the sum and carry of a 1-bit addition and not considering a carry-in from the previous addition is called a Half-Adder (HA). Taking a carry from the previous addition of bits and using it to generate the sum, we consider it a Full-Adder (FA). A single FA represents an addition for 1-bit, meaning **you will need  $N$  Full-Adders for any  $N$ -bit addition**. Below is the logic gate representation of a Half and Full Adder.

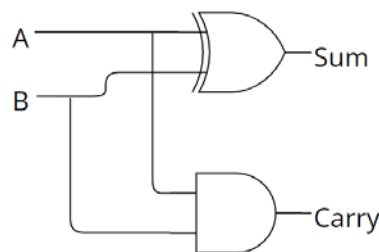


Figure 1: Half Adder

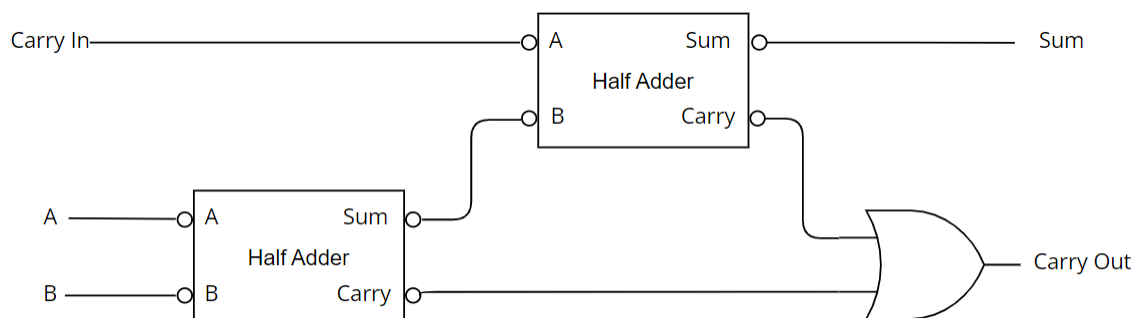


Figure 2: 1-bit Full Adder

Your goal for this section is to create the half and fuller adder circuits in Verilog RTL to solidify the foundations for hardware-based binary addition for a single bit. Once the RTL is written, write a test bench to verify that your circuit is correct.

### Deliverables for Part 1)

- Verilog RTL of Half Adder
- Verilog RTL of Full Adder
- Screenshot of 1-bit Adder waveform of  $1'b1 + 1'b1$  and  $1'b1 + 1'b0$

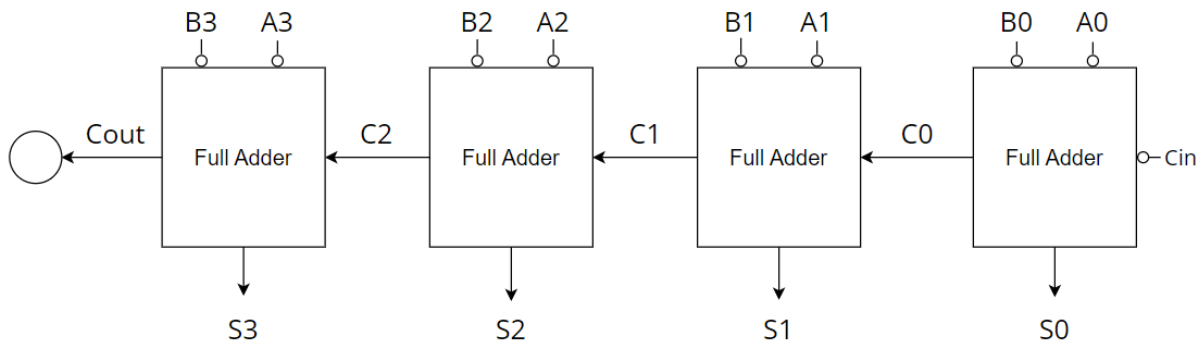
### Part 2) 8-bit The Ripple Carry Adder (RCA)

Now that you know how to create a 1-bit FA, we can connect them into a chain to create an 8-bit adder. This idea of daisy chaining adders is known as the Ripple Carry Adder (RCA). The carry-out of an FA will feed into the carry-in of the next FA, the number of these adders connected will determine the bit width of the addition. For this section of the lab, you will need to create an 8-bit RCA based on the adders you made in Part 1.

There is a useful block in Verilog that you can take advantage of known as a *generate block* where you can utilize a for-loop to generate multiple instantiations of a given hardware module. We will be doing several instantiations for the 8-bit RCA, so it is highly recommended that you use the generate block.

```
genvar i;
generate
    // n is the number of blocks youd like to make
    for(i = 0; i < n; i = i + 1)begin
        Module_name inst_name(input/output ports);
    end
endgenerate
```

**Figure 3: Syntax Example of Generate Block**



**Figure 4: 4-bit RCA Example**

### **Deliverables for Part 2)**

- Verilog RTL of 8-bit RCA
- Screenshot of waveform of 8-bit RCA: 8'b11001011 + 8'b10101010
- Provide the binary and decimal value of the addition

### **Part 3) 8-bit The Carry-Look-Ahead Adder (CLA)**

Although the RCA provides a simple way to add binary numbers, it produces significant propagation delay. The delay is caused by the carry-in relying on the previous addition to process through each full adder before finishing the addition. Thus, the larger the bit width of the RCA (32-bit or 64-bit), the longer it will take to finish the addition. A solution to this problem is the Carry-Look-Ahead Adder (CLA), which processes all carry bits in *parallel* instead of waiting on the previous FA.

The CLA will need two new variables: carry-generate ( $G$ ) and carry-propagate ( $P$ ) per FA. The equations below will show how the  $G$  and  $P$  variables are logically defined:

$$P = A \otimes B$$

$$G = AB$$

The sum and carry-out of the FA are altered to use the new  $P$  and  $G$  variables with their definitions be defined below:

$$S = P \otimes Cin$$

$$Cout = G + PCin$$

Each FA will use this new logic to generate the carry-bits at the same time. Below is an example of this if there was a 4-bit CLA:

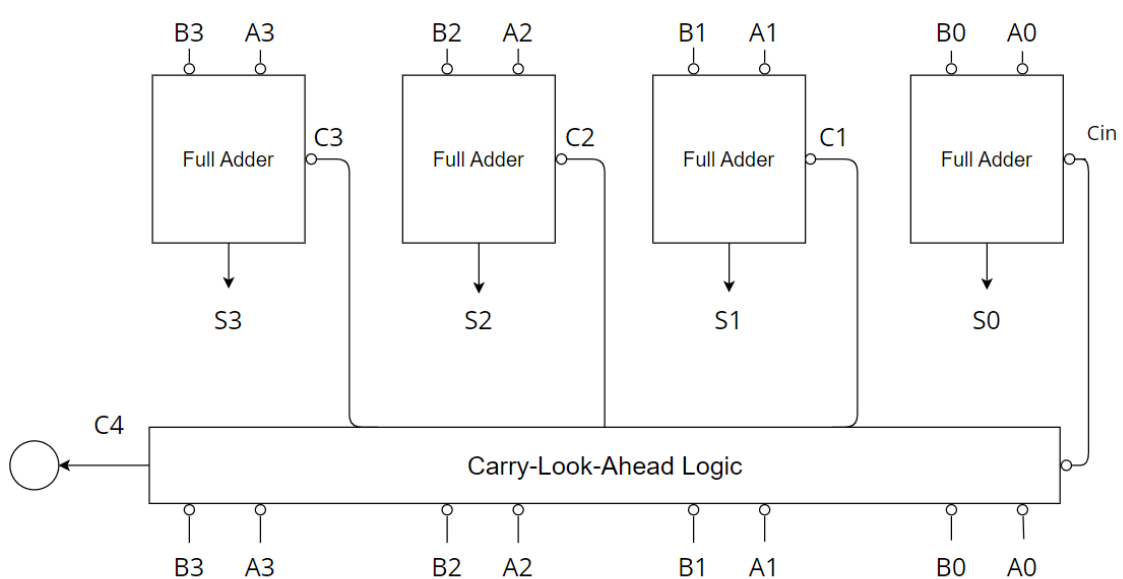
$$C_1 = G_0 + P_0 C_{in}$$

$$C_2 = G_1 + P_1 C_1$$

$$C_3 = G_2 + P_2 C_2$$

$$C_4 = G_3 + P_3 C_3$$

The trend of analyzing tradeoffs between **timing, area, performance, and power** is an idea that computer engineers make daily. The complexity of the carry-bits as the number of FAs increases (size of the data) is an important consideration when deciding to go with the CLA design. The CLA provides better performance in the form of speed but drastically increases the number of gates and power consumption versus the RCA. In this section create an 8-bit CLA in Verilog RTL and verify it by writing a testbench for the design. Use generate-blocks for the carry-look-ahead adder instead of making redundant module instantiations.



**Figure 5: 4-bit CLA Diagram**

### Deliverables for Part 3)

- Verilog RTL of 8-bit CLA
- Screenshot of waveform of 8-bit CLA: 8'b11001011 + 8'b10101010
- Verify that the RCA and CLA produce the same output

### Part 4) Application: Mode Select Between 32-bit Adder and Subtractor

You are an entry-level hardware design engineer for a startup company that designs flexible solutions for hardware devices. The contract that your team is working on requires an adder/subtractor module that is *ultra-low power and minimal in area consumption*. The senior engineer has determined that, with clever applications of a toggleable signal, you can create a subtractor from the internals of an adder by adding combinational XOR gates to the inputs of the FAs. **The "mode" signal determines if the adder circuit will perform addition or subtraction by being set to 0 or 1.** When adding, we perform a standard add operation with the inputs of each bit (A and B) and our carry-in set to the mode signal. When subtracting, we need to XOR one of the signals coming into the FAs (A or B) with the mode signal and have the initial carry-in set to mode. Lastly, the data width for addition and subtraction will be 32 bits for significant computations.

#### Helpful Tips:

1. Choose one input bit signal to be XOR'ed and keep it consistent with the other FAs
2. The Mode signal can replace the carry-in signal in the top module
3. No need to change internals of standard FAs or HAs
4. Decided on which type of adder is best for this situation

### Deliverables for Part 4)

- Verilog RTL of 32-bit Add/Sub Module
- Truth table relationship between "Mode" signal and chosen input bit signal
- Screenshot of waveform of an addition mode and subtraction mode
- 3-4 sentences of the described design

### **Overall Deliverables and Q&A**

- How many gates are required to make a 1-bit FAs?
- What are the four tradeoffs that computer hardware engineers need to consider when designing any digital system?
- What are the pros and cons of an RCA?
- What are the pros and cons of a CLA?
- How many gates were used in the CLA versus the RCA in Part 2 and 3?
- Complete all deliverables for each section of this lab.



---

## EXPERIMENT #5

### Multiplexers & Decoders

---

#### Part 1) Introduction to Multiplexers & Decoders and RTL sensitivity lists

In hardware engineering, multiplexers (mux/muxes) and decoders control the flow of binary information. A mux can take in multiple input signals and forward the selected signal to a single output. There will be  $N$  number of select signals that will determine what the output signal will select from  $2^N$  inputs. The general styles that muxes come in are 2-1, 4-1, 8-1, and 16-1 (inputs-output). A decoder takes in multiple input signals to configure a list of outputs, where the number of inputs  $N$  yields  $2^N$  outputs. The general styles that decoders come in are 1-2, 2-4, 3-8, and 4-16 (inputs-outputs). Additionally, you can perform the inverse function for both muxes and decoders. The alternative inverse designs for muxes and decoders are demuxes and encoders respectively. Figures 1 and 2 show the standard component layout of muxes and decoders.

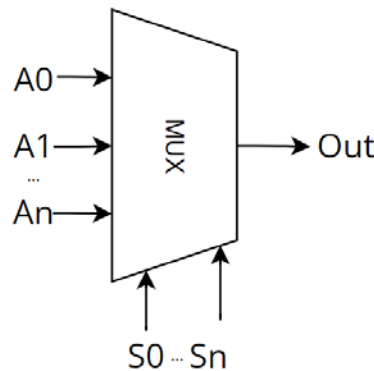


Figure 1: N-1 Multiplexer (N select lines and  $2^N$  inputs)

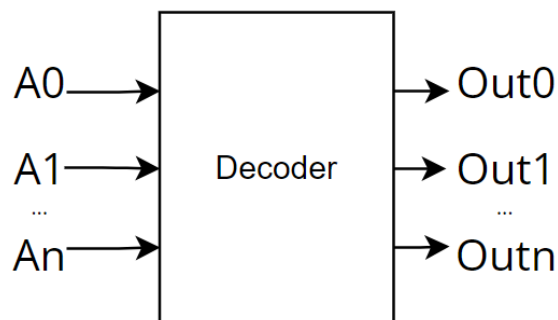


Figure 2: N- $2^N$  Multiplexer (N inputs and  $2^N$  outputs)

**Examples of 2-1 Mux | A and B are inputs where S is 1-bit select and F is an output:**

Inputs			Outputs
A (Index 0)	B (Index 1)	S (Index Select)	F
0	0	0	$F = A = 0$
0	1	0	$F = A = 0$
1	0	1	$F = B = 0$
1	1	1	$F = B = 1$

**Examples of 2-4 Decoder | A and B are inputs and F is an output:**

Inputs		Outputs
A (Index 0)	B (Index 1)	F[3:0]
0	0	$F0 = A'B'$
0	1	$F1 = A'B$
1	0	$F2 = AB'$
1	1	$F3 = AB$

This lab will use the Verilog blocks known as an “always block” and a “case statement block”. An “always block” is a functional block with Verilog that performs operations whenever a given signal is “high” or “low”. Whenever a signal approaches 1 (high), it is a positive edge (posedge) and if it approaches 0 or “low” it is a negative edge (negedge). Always-blocks are essential for digital logic design as they interact with signals such as CPU clocks, read/write signals for memory, and many more. Additionally, we can create an Always Block that will trigger when *any* signal changes, allowing for *combinational logic* behavior rather than *sequential logic*. The signals that drive an always block is known as a sensitivity list.

**Syntax:**

```
always@(sensitivity list)
always@(posedge my_signal) // High
always@(negedge my_signal) // Low
always@(*) // Anyc
```

**Example of Combinational Always Block:**

```
always(*)begin
    if(a == 1 || b == 1)
        y = a & b;
end

always(a or b)begin
    y = a & b;
end
```

### Example of Sequential Always Block:

```
// Trigger on rising edge of clock cycle
// Non-blocking (<=) assignments are used for
sequential circuits

always(posedge clk)begin
    // Store the value of d into q
    q <= d;
end
```

Case statements are similar if not identical to how they are used within other programming languages such as C/C++, Java, Python, etc. There will be a signal in which the case statement will drive its election off of and perform the specific digital logic based on the case signal. The given case signal can be a vector or single-bit signal. Case statements must be written in **initial blocks and always blocks**.

```
Syntax:
case(my_signal)
    0: //case 0
    1: //case 1
    ...
default: //default case

Endcase
```

### Example of case statements W/ parameters:

```
parameter ADD = 0, SUB = 1, MUL = 2, DIV = 3

always@(*)begin
    case(math_op)
        ADD: y = a + b;
        SUB: y = a - b;
        MUL: y = a * b;
        DIV: y = a / b;
        default: y = y;
    endcase
end
```

**Please answer the following questions:**

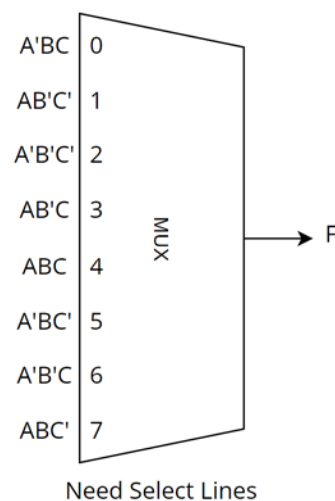
- 1) Build the mux based on the truth table above with Verilog RTL and generate a testbench to verify functionality. Use always(\*) and case statements.
- 2) Build the decoder based on the truth table above with Verilog RTL and generate a testbench to verify functionality. Use always(\*) and case statements.

**Deliverables for Part 1)**

- Verilog RTL of mux
- Verilog RTL of decoder
- Screenshot of waveform that matches the above truth tables of mux and decoder

**Part 2) Simple 8-to-1 Mux**

In this section of the lab, you need to create an 8-1 mux and upload the bitstream to the FPGA like you did in lab 1. There will be three variables that make 8 Boolean expressions input to the mux, and the number of selection lines must be determined. Once the inputs and outputs are determined and the RTL is written, you will need to write a testbench to verify that the mux is correct. Finally, you need to assign the inputs and outputs to switches and LEDs on the BASYS3 FPGA board and upload the bitstream. Figure 3 shows the 8-1 mux and its I/O.



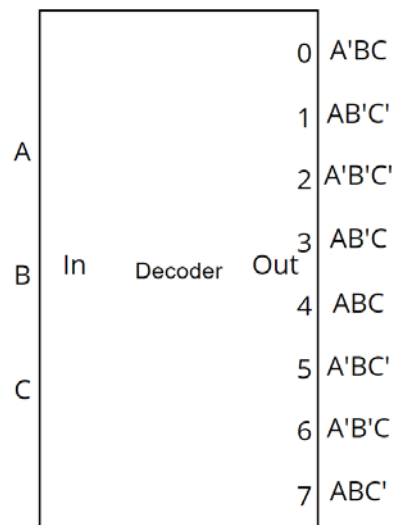
**Figure 3: 8-1 Mux | Needs Select Lines**

## Deliverables for Part 2)

- Verilog RTL of 8-1 mux
- Table of I/O used on FPGA
- Screenshot of 8-1 mux waveform
- Pictures of FPGA board

## Part 3) Simple 3-to-8 Decoder

In this section of the lab, you need to create 3-8 and upload the bitstream to the FPGA like you did in lab 1. There will be three variables that will be decoded in the **8-bit register output**. Each output port will be set to the respective it-index of the register where 0 is the least-significant-bit (LSB) and 7 is most-significant-bit (MSB) (**Ex:  $\text{reg\_bit}[0] = A'BC$**  ). Once the inputs and outputs are determined and the RTL is written, you will need to write a testbench to verify that the decoder is correct. Finally, you need to assign the inputs and outputs to switches and LEDs on the BASYS3 FPGA board and upload the bitstream. Figure 4 shows the 3-8 decoder and its I/O.



**Figure 4: 3-8 Decoder I/O**

### Deliverables for Part 3)

- Verilog RTL of 3-8 decoder
- Table of I/O used on FPGA
- Screenshot of 3-8 mux waveform
- Pictures of FPGA board

### Part 4) Application 4-bit Binary-to-7-Segment Display on BASYS3

You are an FPGA design engineer, and you are tasked with implementing a 7-segment display on a BASYS3 FPGA board. The RTL for getting the fresh rate of the 7-segment display is finished but they need you to design the binary to 7-segment decoder. The binary number will enter the decoder, and the decimal number will be printed on the rightmost 7-segment display. The binary number will be generated by toggling four switches on the FPGA. Since 4-bits are being used for the binary number, there is potential to go beyond the number 9. Any binary number input that is greater than 9, must display the decimal number 0. In addition, you will need to find the pins on the BASYS3 FPGA board that connect to each of the segments on the 7-segment display. Table 1 will provide the values needed to display the numbers 9-0 on the 7-segment display. Some RTL code will be provided to generate 7-segment display refresh rate and enabling signals. Please use the BASYS3 documentation to find the pins and learn about the 7-segment display. <https://digilent.com/reference/programmable-logic/basys-3/reference-manual>

4-bit Input (A,B,C,D)	Decimal 7-Segment Number	7-Segment Output Register Value
0000	0	0000001
0001	1	1001111
0010	2	0010010
0011	3	0000110
0100	4	1001100
0101	5	0100100
0110	6	0100000
0111	7	0001111
1000	8	0000000
1001	9	0000100

**Table 3: 7-Segment Display I/O Values**

### **Summary of tasks:**

- Create a new decoder module that will print binary-to-7-segement using 4-bit input counter
- Display 4-bit binary numbers 9-0, otherwise display 0
- Write a testbench to verify the decoder is correct
- Connect BASYS3 FPGA pins to 7-segment display
- Generate bitstream

### **Deliverables for Part 4)**

- Verilog RTL of 7-segment display decoder
- Table of I/O used on FPGA
- Screenshot of 7-segment display waveform
- Pictures of FPGA board working for binary numbers 10-0

### **Overall Deliverables and Q&A**

- What hardware component is referred to as many-to-one?
- How many inputs would there be for a decoder if there were 32 outputs? Show work.
- What are the names of the inverse designs of mux and decoder respectively?
- What is the difference between a mux and a decoder and when would you use it? Explain
- What are the anode and cathode used for on the 7-segment display?
- What is the refresh period and frequency of the 7-segment display?
- Complete all deliverables for each section of this lab.

## Provided RTL:

```
module seven_segment_top(

    input A, B, C, D,
    input wire clk,
    input wire reset,
    output reg [3:0] segEnable,
    output wire [6:0] outSeg
);

    reg [16:0] clockCounter;
    reg [6:0] onesSegment, tensSegment, hunsSegment, thousSegment;
    wire slowClk;

    // Divides 100 MHz Clock to 0.66Hz clock
    always@(posedge clk or posedge reset)
    begin

        // If reset, set counter to 0
        if(reset)
            clockCounter <= 0;

        // Each clock cycle increment counter
        else
            clockCounter <= clockCounter + 1;

    end

    assign slowClk = clockCounter[16:16];

    // Switch to each 4 segment each slow clock cycle
    always@(posedge slowClk or posedge reset)
    begin

        // If reset, set state machine to 0
        if(reset)
            segEnable <= 4'b1111;

        // State machine that switches between ones and tens place
        case(segEnable)

            4'b1110 : segEnable <= 4'b1101;
            4'b1101 : segEnable <= 4'b1011;
            4'b1011 : segEnable <= 4'b0111;
            4'b0111 : segEnable <= 4'b1110;
            default : segEnable <= 4'b1110;
        endcase
    end

    // TODO : Make print_seg decoder module where the segments are a 7-bit output
    // Decoder will be inside always@(posedge slowClk or posedge reset) block
    // Reset is asynchronous, set segment to 7'b0000001 when reset is 1.
    print_seg onesDisplay(A, B, C, D, slowClk, reset, onesSegment);
    print_seg tensDisplay(A, B, C, D, slowClk, reset, tensSegment);
    print_seg onesDisplay(A, B, C, D, slowClk, reset, hunsSegment);
    print_seg tensDisplay(A, B, C, D, slowClk, reset, thousSegment);

    assign outSeg = (segEnable == 4'b1110)? onesSegment :

        (segEnable == 4'b1101)? tensSegment :

        (segEnable == 4'b1011)? hunsSegment : thousSegemnt;

endmodule
```



---

## EXPERIMENT #6

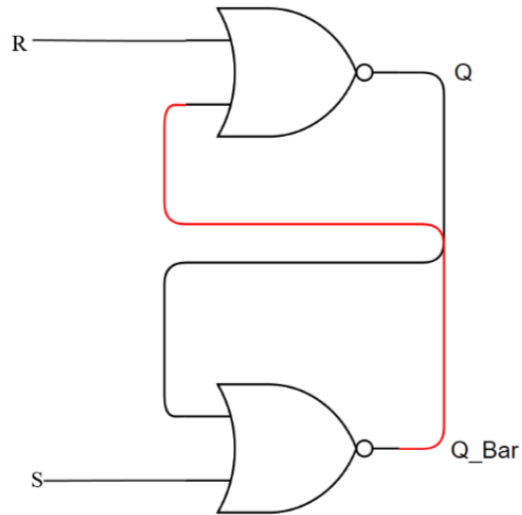
### Latches & Flip-Flops

---

#### Part 1) Introduction to Latches & Flip-Flops

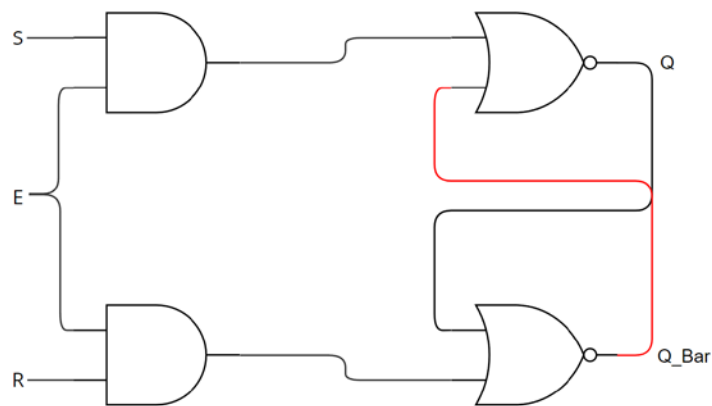
Latches and flip-flops are hardware components that allow a computer to remember or store states or single bits of data. This will enable us to use this data for other purposes later or store relevant outputs from combinational or sequential logic blocks. This concept allows computer engineers to build out the fundamental structures of how computer architectures can execute instructions that run our programs. The difference between a latch and a flip-flop is when they are triggered to store information. Latches are *level-triggered*, meaning the latch will only store data whenever the signal enabling the flow of data is high. Flip flops are *edge-triggered*, meaning that the signal allowing the storage of data is a positive or negative edge of a given clock signal. Both latches and flip flops are sequentially designed within the realm of digital electronics so you will need to use the always-blocks learned in lab 5.

An SR latch, also known as a Set-Reset latch, is a basic bistable memory device used in digital electronics to store a single bit of information. It has two inputs, labeled Set (S) and Reset (R), and two outputs, typically denoted as Q and Q-bar, where Q is the main output and Q-bar is its inverse. The SR latch operates by setting the output Q to high (1) when the S input is activated (high) and resetting Q to low (0) when the R input is activated. If both inputs are low, the latch maintains its current state, holding the previous value of Q. However, if both S and R are high simultaneously, it creates an invalid condition where the outputs can become unstable or unpredictable, which is why this input combination is usually avoided in practical designs. SR latches can be either gated or non-gated, with gated versions including an enable signal to control when the inputs affect the output. The SR latch is one of the simplest and most fundamental building blocks in digital memory circuits, serving as the basis for more complex storage devices like flip-flops. Below will show non-gated and gated SR latch designs alongside their truth table.



S	R	Q	Q-Bar
0	0	Latch prev. value	Latch prev. value
0	1	0	1
1	0	1	0
1	1	x	x

**Figure 1 & Table 1: Non-gated SR Latch & SR Latch Truth Table**



E	S	R	Q	Q-Bar
1	0	0	Latch prev. value	Latch prev. value
1	0	1	0	1
1	1	0	1	0
1	1	1	x	x

**Figure 2 & Table 2: Gated SR Latch & SR Latch Truth Table (E = 1)**

E	S	R	Q	Q-Bar
0	0	0	Latch prev. value	Latch prev. value
0	0	1	Latch prev. value	Latch prev. value
0	1	0	Latch prev. value	Latch prev. value
0	1	1	Latch prev. value	Latch prev. value

**Table 3: Gated SR Latch Truth Table (E = 0)**

**Please answer the following questions:**

- 1) Build the non-gated SR-latch based on the truth table above with Verilog RTL and generate a testbench to verify functionality. Upload to BASYS3, set Q to an LED and S/R to switches to simulate the SR-latch.
- 2) Build the gated SR-latch based on the truth table above with Verilog RTL and generate a testbench to verify functionality. Upload to BASYS3, set Q to an LED and S/R to switches to simulate the SR-latch.

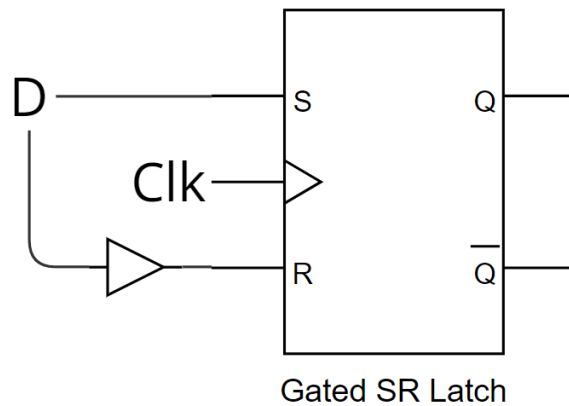
**Deliverables for Part 1)**

- Verilog RTL of gated & non-gated SR latch
- Pictures of FPGA board
- Screenshot of waveform that matches the above truth tables of SR-latches

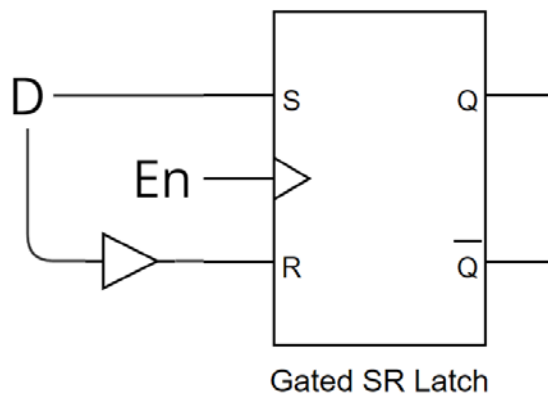
**Part 2) D Flip-Flop and D Latch**

A D latch and a D flip-flop both store a single bit of data, with key differences in how they operate. The D latch has a Data (D) input and an Enable (En) signal; when the enable is active, the latch is "transparent," allowing the D input to pass to the output (Q). When the enable is inactive, the output holds the last value. In contrast, a D flip-flop has a Data (D) input and a Clock (Clk) input. It captures the value of D only at the clock edge (rising or falling), storing this value at the output (Q) until the next clock edge, regardless of changes in D between clock pulses. Both devices also provide an inverted output Q-bar representing the complement of the stored value.

In this part of the lab, you will need to create a D latch and a D flip-flop from an SR latch and monitor its behavior. Below will be a high-level schematic of the D flip-flop and D latch and how it can be constructed using an SR latch. Lastly, you will need to demonstrate the capability of the D latch and D flip-flop on the BASYS3 by setting the Q to an LED, D to a switch, an enable signal for the D latch to a switch, and the clock signal to the 100MHz crystal oscillator clock on the BASYS3 FPGA. Be sure to compare how the D flip-flop and D latch differ in how they can store data (turn on/off the LED).



**Figure 3: D Flip-Flop from a Gated SR-Latch**



**Figure 4: D Latch from a Gated SR-Latch**

### How to Write Testbenches for Sequential Clock Circuits

Testing sequential circuits with clock and reset signals involves several key steps to ensure the circuit functions correctly over time. First, the clock signal must be provided to synchronize the sequential elements, such as flip-flops or latches, enabling them to transition between states. The test begins by applying a reset signal to initialize the circuit to a known state, typically setting all outputs or internal states to a defined value. After releasing the reset, various input patterns are applied in sync with the clock to observe the circuit's behavior over multiple clock cycles. The circuit's outputs are then monitored to verify that they match the expected outputs for the given inputs. It's crucial to check that the circuit correctly handles edge cases, such as changes in inputs just before or after a clock edge, and that it reliably returns to a known state upon receiving a reset signal. Testing under different clock frequencies and reset conditions also helps ensure robust performance across all operational scenarios.

### Provided Testbench RTL for D-Flip Flop:

```
module d_flip_flop_tb();

    reg iD, iclk, rst;

    wire oQ, oQ_bar;

    d_flip_flop
    UUT1(.iD(iD), .iclk(iclk), .rst(rst), .oQ(oQ), .oQ_bar(oQ_bar));

    integer i;

    initial begin
        iclk = 1;
        rst = 1;
        iD = 0;
        #10;
        rst = 0;
        #10;
        iD = 1;
        #30;
        iD = 0;
        #40;
        iD = 1;
        #40;
        $finish;
    end

    always #10 iclk = ~iclk;

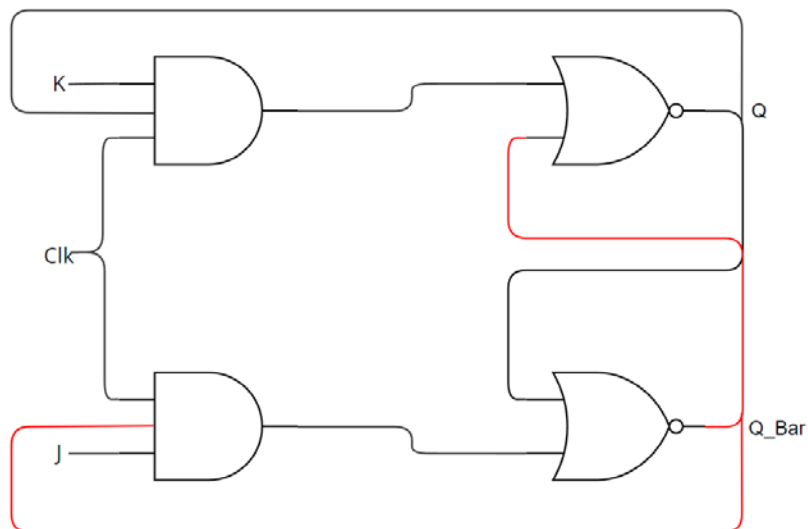
endmodule
```

## Deliverables for Part 2)

- Verilog RTL of D flip-flop and D latch
- Pictures of FPGA board for D flip-flop
- 2-3 Sentence of test plan
- Waveform based on test plan

## Part 3) JK Flip-Flop

A JK flip-flop is a versatile digital storage element that extends the functionality of a basic flip-flop by incorporating two inputs, J and K, along with a Clock (Clk) input. The JK flip-flop's behavior is determined by the combination of the J and K inputs at each clock edge. When both J and K are low, the flip-flop holds its previous state. If J is high and K is low, the output (Q) is set to high. Conversely, if J is low and K is high, the output is reset to low. The most distinctive feature of the JK flip-flop is its toggle mode: when both J and K are high, the flip-flop toggles its output state with each clock pulse, switching between 0 and 1. This makes the JK flip-flop useful in counters and other applications requiring complex state changes. The flip-flop also provides an inverted output Q-bar, which is the complement of the Q output. Below will show the schematic and truth table of a JK flip flop.



J	K	Clk	Q	Q-Bar
0	0	posedge	Latch prev. value	Latch prev. value
0	1	posedge	0	1
1	0	posedge	1	0
1	1	posedge	Toggle (0 ->1 or 1->0)	toggle (0 ->1 or 1->0)

**Figure 5 & Table 3: JK Flip Flop Schematic Truth Table**

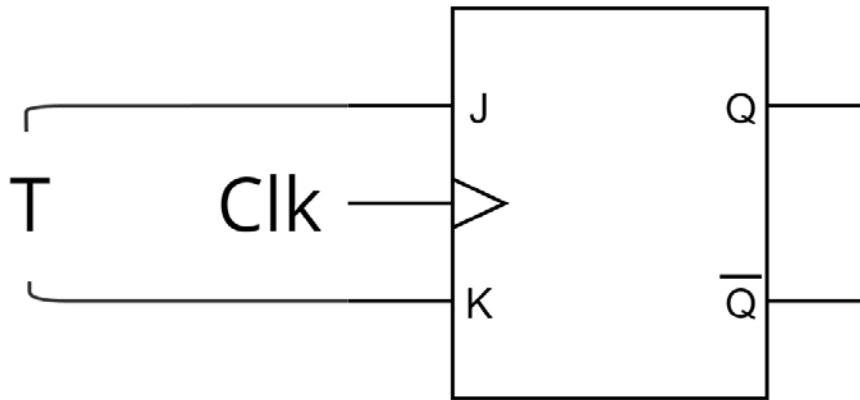
In this section of the lab, you will build the JK flip flop in Verilog RTL, verify the functionality with a waveform, and upload the RTL to the BASYS3 FPGA to show the JK flip flop works physically. You will need to set Q to an LED, J and K to switches, and the Clk to the 100MHz clock oscillator on the FPGA.

### **Deliverables for Part 3)**

- Verilog RTL of JK flip-flop
- Pictures of FPGA board for JK flip-flop
- 2-3 Sentence of test plan
- Waveform based on test plan

### **Part 4) T Flip-Flop**

A T flip-flop, or toggle flip-flop, is a digital memory device used to store and toggle a single bit of data. It has a T (Toggle) input and a Clock (Clk) input. The T flip-flop changes its output state (Q) on each clock pulse when the T input is high (1). If T is low (0), the output remains unchanged, holding its previous state. Essentially, when T is active, the flip-flop "toggles" its output from 0 to 1 or from 1 to 0 with each clock cycle. This toggling behavior makes T flip-flops useful in counters, where a sequence of clock pulses can be counted by repeatedly toggling the output, and in frequency division, where the output frequency is half that of the input clock. The T flip-flop also provides an inverted output Q-bar, which is the complement of the current state. Below will show the schematic of the T flip-flop.



**Figure 6: T Flip Flop Schematic**

In this section of the lab, you will build the T flip flop in Verilog RTL, verify the functionality with a waveform, and upload the RTL to the BASYS3 FPGA to show the T flip flop works physically. You will need to set Q to an LED, T to a switch, and the Clk to the 100MHz clock oscillator on the FPGA.

#### **Deliverables for Part 4)**

- Verilog RTL of T flip-flop
- Pictures of FPGA board for T flip-flop
- 2-3 Sentence of test plan
- Waveform based on test plan

#### **Part 5) Applications: 8-bit Dual Edge-Triggered Flip-Flop**

You are a hardware design engineer intern working on GPU acceleration for high throughput designs. Your manager has asked you to create an 8-bit register using flip-flops that can store data on either the positive or negative edge of the clock. A register is a link of flip-flops that store a single bit, so if you want to have an N-bit register you need N number of flip-flops. To verify this, write a test plan and create the waveforms to ensure the dual edge triggering works as intended. Lastly, upload this Verilog program to a BASYS3 FPGA, using 8 LEDs to visualize each bit being stored.



### **Deliverables for Part 5)**

- Verilog RTL of 8-bit dual edge-triggered flip flop
- Pictures of FPGA board for 8-bit dual edge-triggered flip flop
- 4-5 Sentence of test plan
- Waveform based on test plan

### **Overall Deliverables and Q&A**

- What is the difference between a latch and a flip flop?
- What types of flip flops are capable of “toggling”? Explain their differences.
- How many flip flops would you need if you wanted to make a 32-bit register? What type of flip flop would you use?
- Explain in your own words why there is an invalid state within the SR-latch.
- Complete all deliverables for each section of this lab.

---

## EXPERIMENT #7

### Hardware Memory Elements

---

#### Part 1) Introduction to Hardware Memory Elements

ROM (Read-Only Memory), RAM (Random-Access Memory), and registers are critical components in digital hardware, each serving unique functions that contribute to the overall performance and efficiency of a system. ROM provides permanent storage for firmware and essential instructions that the system needs to boot and operate, ensuring stability and consistency. RAM, on the other hand, offers temporary storage that allows the processor to quickly access and manipulate data while executing programs, directly influencing the speed and responsiveness of the system. Registers, located within the CPU, are the fastest form of memory, holding the most critical data and instructions currently being used by the processor. These components work together to enable efficient processing, with ROM providing long-term storage, RAM enabling flexible data handling, and registers ensuring rapid execution of tasks. It should be noted that that flop-flops are the foundations of registers

In this lab, you will learn how to build each of these hardware elements and learn simpler ways to design flip-flops that don't derive from latches. Specifically for this section, you will need to **create an 8-bit register in two ways**: a D flip flop derived from a SR-latch and another using the “posedge” characteristic for clock signal within the parameter list of an always-block. Below is an example of the Verilog segment that lists how to build out the D flip flop using the posedge of the clock within the always-block. Note that you can use this technique with other flip flops such as the T and JK flip flops. Figure 1 shows an example of what the internal of a parallel input parallel output (PIPO) 4-bit register looks like with all the D flip flops used.

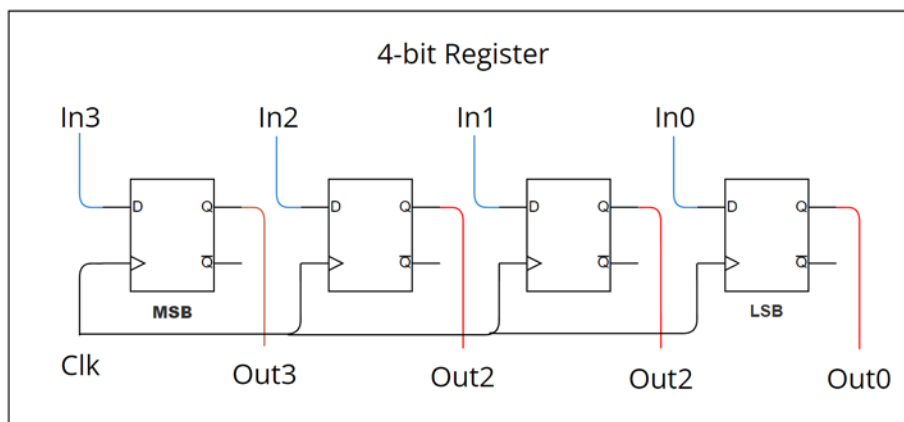


Figure 1: 4-bit PIPO Register Example

```

// q collects the value of d at the raising
edge of the clk signal with a synchronous reset to
set the value of q
always @(posedge clk) begin
    if(reset)
        q <= 1'b0;
    else
        q <= d;
end

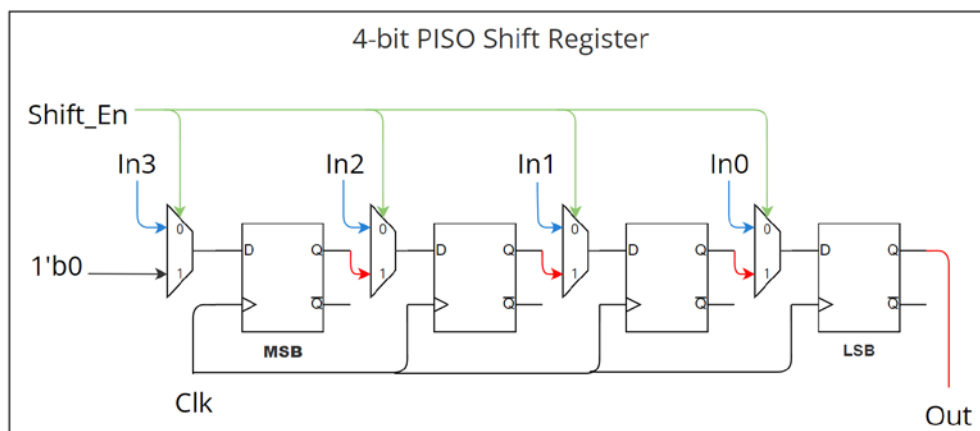
```

### Deliverables for Part 1)

- Verilog RTL of 4-bit register using SR-latch based D flip flops
- Verilog RTL of 4-bit register using posedge always-block D flip flops
- Screenshot of both waveforms for 4-bit register variations (must match values)

### Part 2) Parallel Input Serial Output Shift Register

Shift registers are specialized digital circuits used to store and manipulate data by shifting bits in a specific direction, either left or right, on each clock pulse. They are composed of a series of flip-flops connected in sequence, where each flip-flop holds a single bit of data. Shift registers are commonly used in applications like data conversion between serial and parallel formats, data storage, and creating time delays in digital systems. By shifting data through the register, shift registers can efficiently handle sequential data processing tasks. In figure 2 you will see an example of a 4-bit parallel in serial out shift register.



**Figure 2: 4-bit PISO Shift Register**

In this section of the lab, you will need to create a 4-bit PISO shift register. This type of shift register should output a single bit per clock cycle of the input that was given in parallel. Additionally, you will need to make sure that each D flip-flop must have a positive synchronous reset signal.

#### 4-bit PSIO Shift Register Example:

Input	Clock Cycle	Output
1101	1	1
x	2	0
x	3	1
x	4	1

#### Constraints for Part 2)

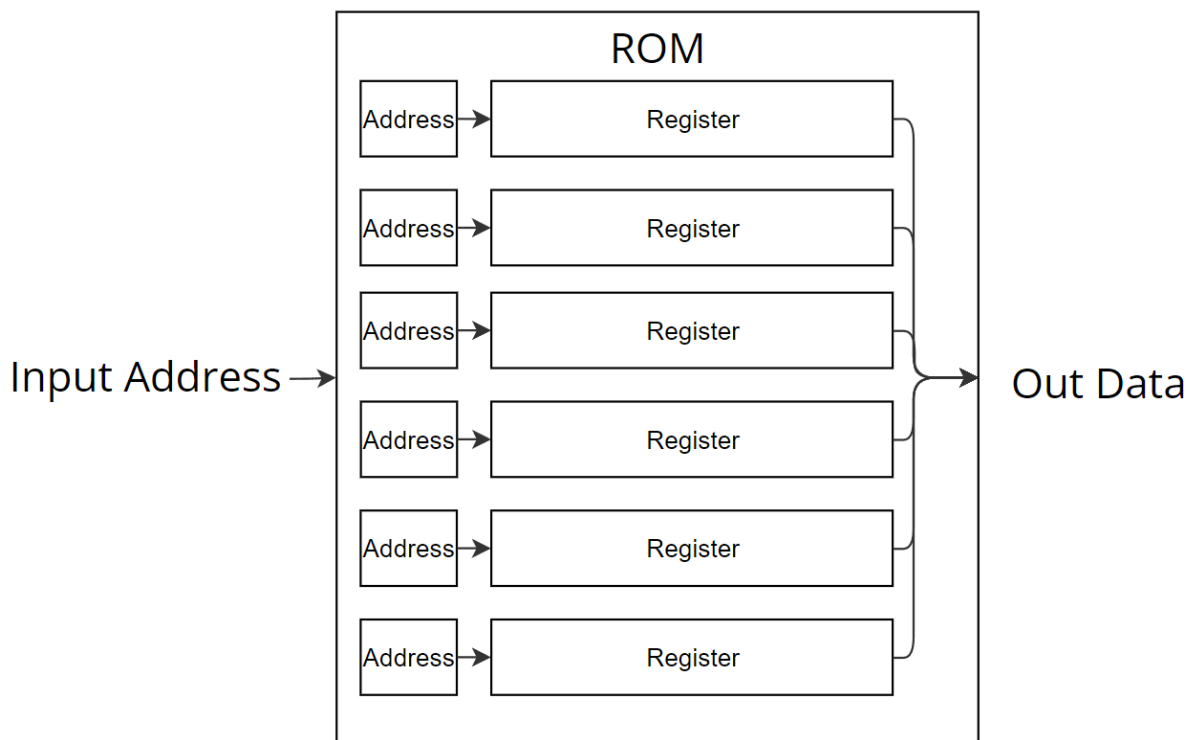
- Must use D flip-flop modules to make the shift register
- Create module for the muxes between shift registers

#### Deliverables for Part 2)

- Verilog RTL of 4-bit PISO shift register
- Waveform screenshot of 4-bit PISO shift register based on example
- Screenshot of 4-bit PISO schematic

### Part 3) 8x8-bit Register Read-Only-Memory Module

ROM (Read-Only Memory) is a type of non-volatile memory used in digital systems to store data that does not change during the system's operation. Unlike RAM, which can be written to and read from dynamically, ROM is typically written once during manufacturing and then only read during the device's operation. This makes ROM ideal for storing firmware, which includes the fundamental code required for a device to boot up and operate. ROM retains its contents even when the power is turned off, ensuring that the system has access to essential instructions immediately upon startup. The data stored in ROM is accessed by supplying an *address* to the memory, which then *outputs the corresponding data*, making it a crucial component in embedded systems, microcontrollers, and other applications where reliable, permanent storage is needed. Figure 3 shows the internal workings of ROM and how providing an input address will output the data held within the respective register.



**Figure 3: Simple Internal of ROM**

Based on figure 3 we can see that ROM acts similarly to a decoder holding valuable information in the form of registers that are typically hardcoded. In Verilog there is a useful way to make vectors that allow us to create simple ROM/RAM devices, below will show how you can create a vector.

```
reg [DATA_SIZE-1:0] ROM [0:2**ADDR_WIDTH-1];
```

This will create a vector that has  $2^{AddrWidth}$  number of registers that are DATA\_SIZE bits in width. The leftmost bracket from the name of the reg “ROM” will order the registers from 0 to DATA\_SIZE-1 (little endian) but if you reverse the order (DATA\_SIZE-1 to 0) then you will represent the register in big-endian. This way we don’t need to create the D flip flop modules to define registers but the cost of limited flexibility with the data.

In this section of the lab, you will need to create a ROM module that has the following constraints: 8 total registers, each register being 8-bits in width, and must be created using vectors. Use your knowledge of how decoders are made and write a Verilog code that meets the design metrics. Below will be a table of BASYS3 peripherals and register values needed to complete this section.

Registers	Value
Reg 0	0x32
Reg 1	0x84
Reg 2	0x21
Reg 3	0x53
Reg 4	0x83
Reg 5	0x98
Reg 6	0x72
Reg 7	0x11

**Table 1: Register Values**

BASYS3 Port (Switches)	Register Address (Input)
V17	Address Bit 0
V16	Address Bit 1
W16	Address Bit 2

**Table 2: Register Address Inputs via BASYS3 Switches**

BASYS3 Port (LEDs)	Register Data (Output)
U16	Reg 0
E19	Reg 1
U19	Reg 2
V19	Reg 3
W18	Reg 4
U15	Reg 5
U14	Reg 6
V14	Reg 7

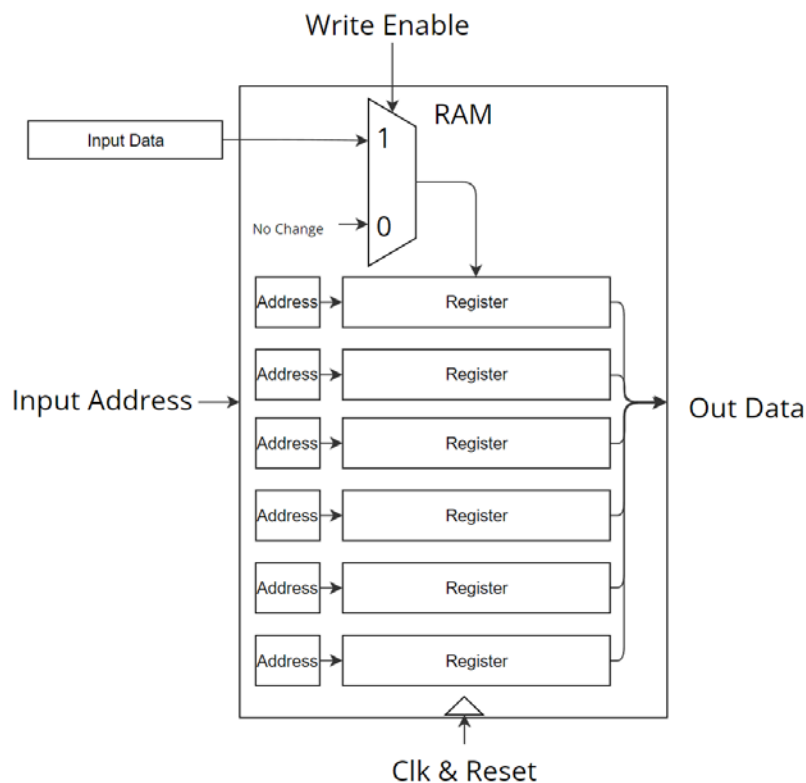
**Table 3: Register Data Output via BASYS3 LEDs**

### **Deliverables for Part 3)**

- Verilog RTL of ROM
- Waveform screenshot of ROM working with stated metrics
- Pictures of FPGA working for each register within ROM

### **Part 4) 8x4-bit Register Random-Access-Memory Module**

RAM (Random-Access Memory) is a type of volatile memory used in digital systems to store data and instructions that the CPU needs to access quickly while performing tasks. Unlike ROM, RAM can be read from and written to, allowing it to store temporary data that changes frequently during a system's operation. RAM is called "random-access" because the processor can directly access any memory location without having to go through other locations sequentially. This allows for rapid retrieval and modification of data, which is essential for multitasking and running complex applications. However, RAM is volatile, meaning it loses all stored information when the power is turned off. RAM plays a crucial role in determining the speed and performance of a system, as it provides the working space the CPU needs to execute programs and manage data efficiently.



**Figure 4: Simple Internals of RAM Module**

Based on the diagram shown, the main difference between ROM and RAM is writing to the registers. You will need to create a RAM module with the following constraints: 8-bit register that are 4-bits in size, you must use a vector to create the structure of RAM, change register data based on a positive clock edge, and have a synchronous reset. Use your knowledge of how decoders are made and write a Verilog code that meets the design metrics. Below will be a table of BASYS3 peripherals and register values needed to complete this section.

BASYS3 Port (Switches)	Register Address (Input)
T3	Write EN

**Table 6: Write Enable Port for BASYS3**

BASYS3 Port (Switches)	Register Address (Input)
U16	Input Data Bit 0
E19	Input Data Bit 1
U19	Input Data Bit 2
V19	Input Data Bit 3

**Table 6: Register Address Input Port for BASYS3**

BASYS3 Port (Switches)	Register Address (Input)
V17	Address Bit 0
V16	Address Bit 1
W16	Address Bit 2

**Table 7: Register Address Inputs via BASYS3 Switches**

BASYS3 Port (LEDs)	Register Data (Output)
W18	Reg 4
U15	Reg 5
U14	Reg 6
V14	Reg 7

**Table 8: Register Data Output via BASYS3 LEDs**

*Populate register 0 through 3 with the input data bits 0 through 3:*

Registers	Value
Reg 0	0x3
Reg 1	0xc
Reg 2	0xa
Reg 3	0x4

**Table 9: Register Data with Input Data**



#### **Deliverables for Part 4)**

- Verilog RTL of RAM
- Waveform screenshot of RAM working with stated metrics
- Pictures of FPGA working for each register within RAM

#### **Part 5) Application: Heterogenous Memory Architecture**

You are a memory hardware engineer who has been tasked with setting up a ROM and RAM heterogeneous memory architecture. The ROM and RAM will be vector-based with 4x4-bit registers with both of them needing a positive edge trigger and positive synchronous reset. The contents of the ROM will be the address pointer for the RAM, and you must populate the RAM with your input data. Your boss would like you to test this hardware on a BASYS3 FPGA and provide port connections. Based on the defined metrics and tables below, create the heterogeneous memory architecture.

<b>BASYS3 Port (Switches)</b>	<b>Register Address (Input)</b>
T3	Write EN

**Table 10: RAM Write Enable Port for BASYS3**

<b>BASYS3 Port (Switches)</b>	<b>Register Address (Input)</b>
U16	Input Data Bit 0
E19	Input Data Bit 1
U19	Input Data Bit 2
V19	Input Data Bit 3

**Table 11: RAM Register Address Input Port for BASYS3**

<b>BASYS3 Port (Switches)</b>	<b>Register Address (Input)</b>
V17	Address Bit 0
V16	Address Bit 1

**Table 12: ROM Register Address Inputs via BASYS3 Switches**

BASYS3 Port (LEDs)	Register Data (Output)
W18	Reg 0
U15	Reg 1
U14	Reg 2
V14	Reg 3

**Table 13: RAM Register Data Output via BASYS3 LEDs**

*Populate RAM register 0 through 3 with the input data bits 0 through 3:*

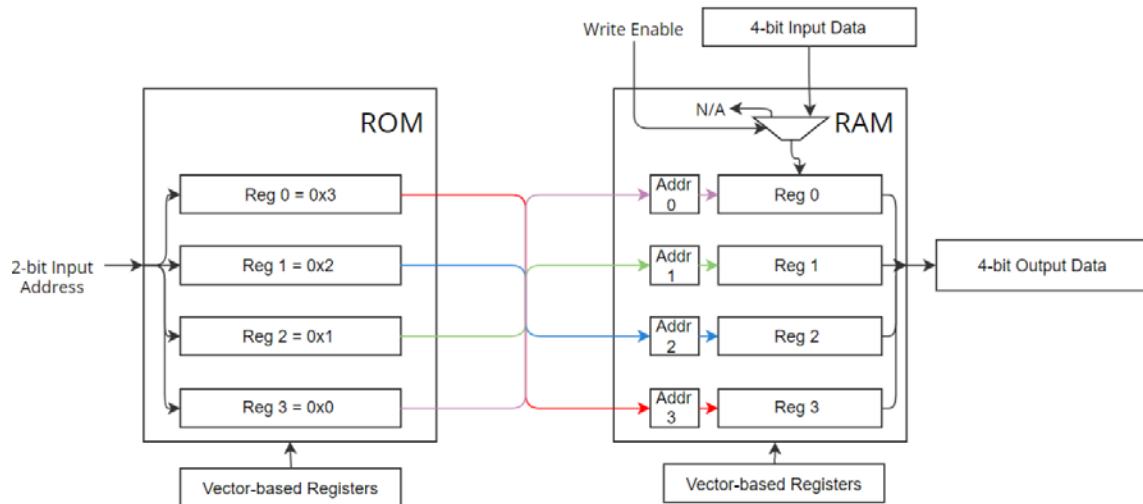
Registers	Value
Reg 0	0x3
Reg 1	0xc
Reg 2	0xa
Reg 3	0x4

**Table 14: RAM Register Data with 4-bit Input Data**

*Populate ROM register 0 through 3 with the input data bits 0 through 3:*

Registers	Value
Reg 0	0x3
Reg 1	0x2
Reg 2	0x1
Reg 3	0x0

**Table 15: ROM Register Data with Input Data**



**Figure 5: Heterogenous Memory Architecture Diagram**

### Deliverables for Part 5)

- Verilog RTL of RAM module
- Verilog RTL of ROM module
- Verilog RTL of Top-level heterogenous memory
- Waveform screenshot of memory architecture working with stated metrics
- Pictures of FPGA working for each register within memory architecture

### Overall Deliverables and Q&A

- What is the difference between a register and a shift register?
- What two types of data conversions does a shift register perform?
- How many clock cycles would it take to serially output data from a 32-bit shift register?
- What is the difference between ROM and RAM?
- Why would you want to use flip-flop registers over vector-based registers?
- How many registers should I expect to have if the address size of RAM is 8-bits?
- Complete all deliverables for each section of this lab.

---

## EXPERIMENT #8

### Finite State Machines

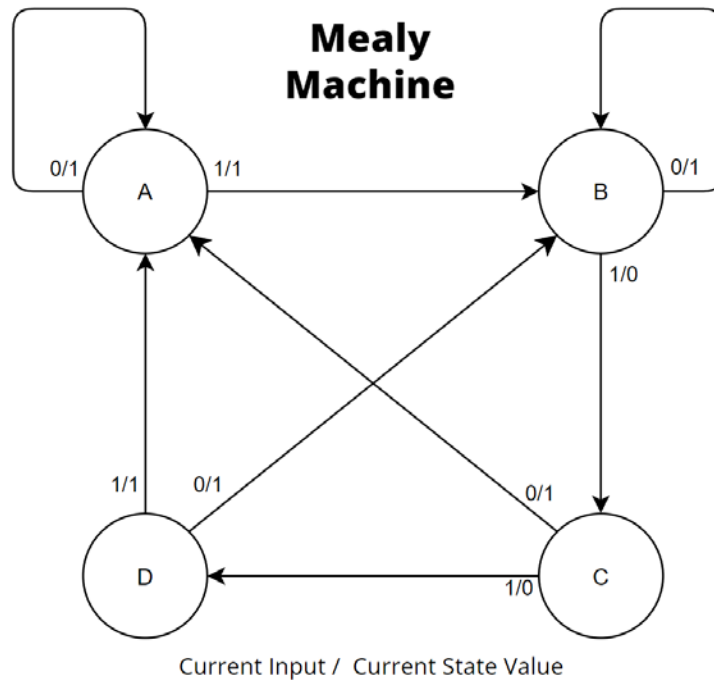
---

#### Part 1) Introduction to Finite State Machines

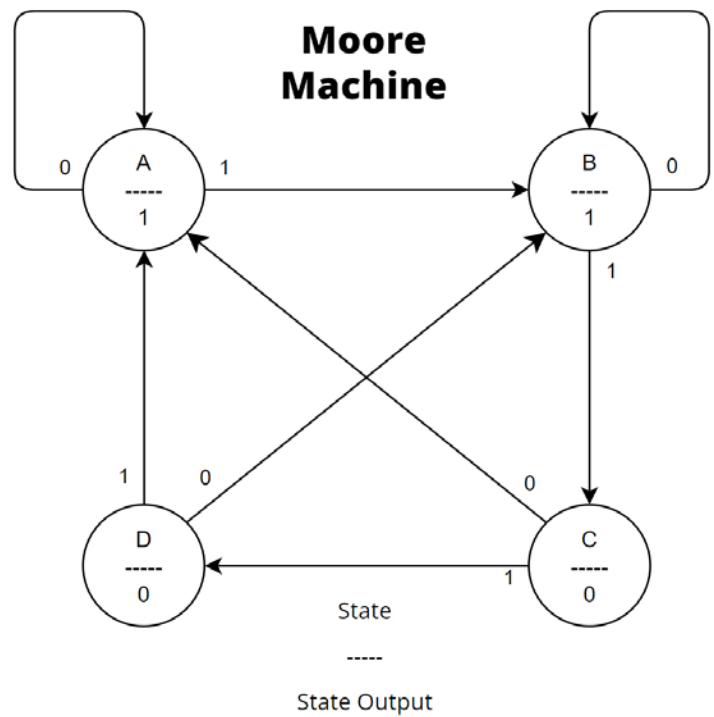
A Finite State Machine (FSM) is a computational model used to design systems that can be in one of a finite number of states at any given time. It operates by transitioning between these states based on input signals and predefined rules. In hardware design, FSMs are crucial because they provide a structured way to model and implement control logic, which is fundamental to the operation of digital circuits. FSMs are used in a wide range of applications, from simple devices like traffic lights to complex systems such as processors and communication protocols. By clearly defining states and transitions, FSMs help ensure predictable and reliable behavior in hardware, making them essential for the design of efficient and effective digital systems.

Mealy and Moore machines are two types of finite state machines (FSMs) that differ in how they generate outputs. In a Mealy machine, the output is determined by both the current state and the current input. This means that the output can change immediately in response to an input, leading to potentially faster response times since the output can vary within a single state. On the other hand, a Moore machine generates output solely based on the current state, independent of the input. As a result, the output changes when the machine transitions to a new state, leading to more predictable and stable outputs, albeit sometimes with a slower response to inputs. The choice between Mealy and Moore machines depends on the specific needs of the system being designed. Mealy machines are often more efficient in terms of state transitions, and Moore machines offer simpler and more predictable design characteristics. FSMs need to have a state diagram and state table that describe their characteristics. Below Table 1 will show the state tables for a Mealy and Moore machine that has four states being A, B, C, and D. Figures 1 and 2 will show what a Mealy and Moore machine is based on Tables 1 and 2.

Input Value	Current State	Next State	Mealy Output	Moore Output
0	A	A	1	1
0	B	B	1	1
0	C	A	1	0
0	D	B	1	0
1	A	B	1	1
1	B	C	0	1
1	C	D	0	0
1	D	A	1	0



**Figure 1: Mealy Machine | State Value = State Output**



**Figure 2: Moore Machine Equivalent of Mealy Machine**

In this section of the lab, you will create the Moore and Mealy FSMs from Figures 1 and 2. Keep in mind that the output of the Mealy machine will be *asserted during the state transitions* and the Moore machine will only assert the output during every clock cycle. To will need two state registers for any FSM that will need to be  $\log_2 N$  in size where  $N$  is the number of states. For this example, we will need to state registers to be 2 bits in size. Below will be sudo-RTL on how you can setup an FSM.

```
reg[size-1:0] state, next_state;

parameter STATE_0 = 0, STATE_1 = 1, STATE_2 = 2;

// Mealy output is decided within the states
always@(*)begin
    case(state)
        STATE_0: next_state = (transition logic) ? STATE1 : STATE 2;
        STATE_1: ...
        STATE_2: ...
        default: next_state = STATE_0;
    endcase
End

// Async. positive reset
always@(posedge clk or posedge reset)begin
    if(reset)begin
        state <= 0;
        next_state <= -0;
    end

    // Moore output is decided during each clock based on state
    else
        state <= next_state;
end
```

### **Deliverables for Part 1)**

- Verilog RTL of Moore machine.
- Verilog RTL of Mealy machine
- Create testbench using this input sequence below with a 10ns delay after each delivered bit:
  - 0 1 0 1 1 0 1 0 1 0 0 1
- Screenshot of Waveform of Moore and Mealy machine output.

### **Part 2) Bit Sequence Detector with Mealy Machine from PISO Shift Register**

In this section of the lab will need to create a Mealy FSM that can take in a bit input sequence in serial and determine if the sequence “1101” has been detected. You will need to create this FSM with only four states with an asynchronous negative reset with a positive clock edge pulse to change states. Once the reset is triggered, you will disregard all the previous read bits in the sequence and start fresh. Once this is tested on a waveform, you will need to use an 8-bit that will get the parallel information from switches on the BASYS3 FPGA. Use the rightmost switch to act the LSB of the 8-bit PISO shift register until you use 8 switches to make up the register's data. Make sure to test the overall design with both the Mealy FSM and PISO shift register. Toggle the rightmost LED on the FPGA to show that the 1101 sequence exists within the shift register. Lastly, you will need to set a switch for the reset signal, any open switch on the FPGA will do.

### **Deliverables for Part 2)**

- Verilog RTL of Mealy machine.
- Screenshot of waveform of Mealy machine working with PISO shift register using the asynchronous reset.
- Create testbench using this input sequence from PISO below with a 10ns delay after each delivered bit:
  - Test 1: 0 1 0 1 1 0 1 0
  - Test 2: 1 1 0 1 0 0 0 0 Reset 1 0 1 0 1 0 1 0
- Photo of FPGA on Mealy FSM with PISO shift register.

### **Part 3) Application: 4-bit Multiple and Divider using a Moore FSM**

As an Arithmetic Hardware Engineer, you've been tasked with designing a multiplier and a divider using Moore finite state machines (FSMs). The goal is to implement two separate digital

circuits: one that performs multiplication and another that performs division of unsigned binary numbers.

## Requirements:

### 1. Multiplier FSM:

- **Input:** Two 4-bit unsigned binary numbers (multiplicand and multiplier).
- **Output:** A 8-bit product.
- **Operation:** The FSM should use sequential addition (shift and add method) to calculate the product.
- **States:** Define states for initialization, partial product calculation, shifting, and completion.
- **Output Logic:** The output depends solely on the state, as per the Moore machine paradigm.
- **Control Logic:** Use clock cycles to manage state transitions. Include a mechanism for detecting when the multiplication is complete (e.g., based on a counter or when the multiplier becomes zero).

### 2. Divider FSM:

- **Input:** Two 4-bit unsigned binary numbers (dividend and divisor).
- **Output:** 4-bit quotient and 4-bit remainder.
- **Operation:** The FSM should implement a restoring division algorithm.
- **States:** Define states for initialization, shifting, subtraction, checking the sign of the remainder, and completion.
- **Output Logic:** Similar to the multiplier, the output depends solely on the current state.
- **Control Logic:** State transitions should be based on clock cycles and the current value of the remainder.

## Design Process:

- **State Diagrams:** Create detailed state diagrams for both the multiplier and divider FSMs. Ensure each state is clearly defined with corresponding transitions based on input conditions.
- **State Encoding:** Choose an appropriate encoding scheme for the states (e.g., binary encoding) to minimize hardware complexity.
- **Verilog Implementation:** Write the Verilog code to implement both the multiplier and divider FSMs.



- **Testbench:** Develop a comprehensive testbench that verifies the functionality of both circuits for a wide range of inputs, including edge cases like multiplying or dividing by zero.

You will need to upload this RTL to the BASYS3 FPGA for a basic demonstration for your engineering team. The first 4 rightmost switches will act as the first number and the next 4 switches will act as the second number. The leftmost switches will act as a toggle switch to change between multiplication and division. There will need to be an asynchronous positive reset switch that can be set to any open switch on the FPGA.

### **Deliverables for Part 3)**

- Detailed state diagrams and explanations of each state.
- Verilog code for the multiplier and divider FSMs.
- A testbench with simulation results showing correct operation.
- Photo of FPGA working with the same test using for the testbench simulation
- A summary of the design process and any challenges faced during implementation.

### **Overall Deliverables and Q&A**

- What is the difference between a Moore and Mealy machine? Explain the pros and cons.
- What size would your state registers need to be if you had 16 states?
- Where do you set the outputs of a Mealy machine?
- Where do you set the outputs of a Moore machine?
- In your own words, do you think FSMs can replace other digital circuits? Explain your reasoning.
- Complete all deliverables for each section of this lab.