

Lab 1: Introduction to Xilinx Vivado and XSim

EE478 Fall 2019

Objective

In this lab, you will become familiar with the development and testing of digital systems implemented for Xilinx FPGAs using the Xilinx Vivado software and the Xilinx Simulator (XSim). You will be given instructions for project setup, design entry, simulation, and synthesis, as well as programming your Zybo development board. For the remaining labs, you will be expected to perform these basic steps, so try to become familiar with the layout of the software.

Deliverables

Lab start: Beginning of lab session during the week of 9/9.

Demo: Working 2-bit unsigned adder in simulation waveform and on Zybo board using four slider switches as input and three LEDs as output.

Demo due: End of lab session during the week of 9/9.

Design Document due:

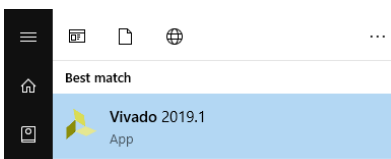
- On UBLearns by 11:59PM on 9/20. *Submit one per team.*

Lab Contents

Objective	1
Deliverables.....	1
Part A: Creating a new project in Xilinx Vivado	1
Part B: Adding new source to the project.....	3
Part C: Creating a testbench	7
Part D: Constraints	9
Part E: Synthesize and implement design and generate programming file	11
Part F: Program Zybo board and test.....	12

Part A: Creating a new project in Xilinx Vivado

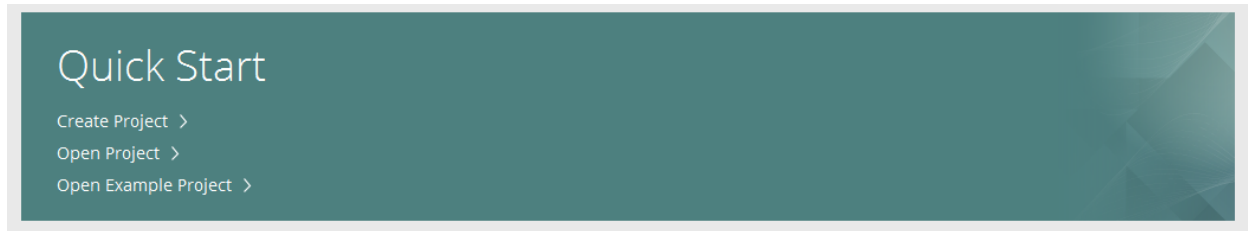
Let's start by opening the Xilinx Vivado software suite, which we will be using for all of our designs this semester. Search Start for Vivado



This tool is an example of an IDE CAD tool (Integrated Development Environment Computer Aided Design). An IDE generally provides a way to enter a hardware or software design along with a way to test it and build it for the target system. Vivado includes the Xilinx Simulator (XSim) for testing our designs, as well as tools for synthesizing our designs so that they can be used to program Xilinx FPGAs.

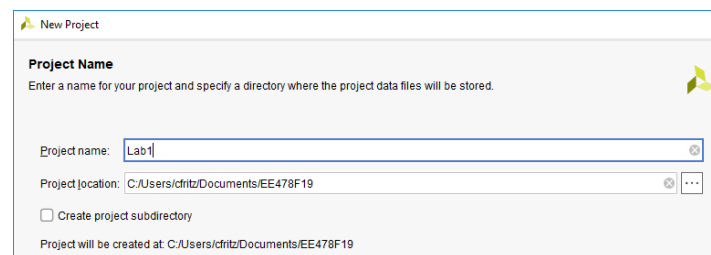
Let's get started right away by creating a new project. Our goal in this lab is to create a 2-bit Ripple Carry Adder and see it work on the Zybo board using slider switches as inputs and LEDs as outputs. You already have seen this code from the lectures, so you should have your notes handy for reference.

Create a new project by pressing Create Project under Quick Start.



Important: Do not assume that your project files and code will remain on the lab computers from week to week. After each lab, be sure to back up your code somewhere, or create the project on a flash drive.

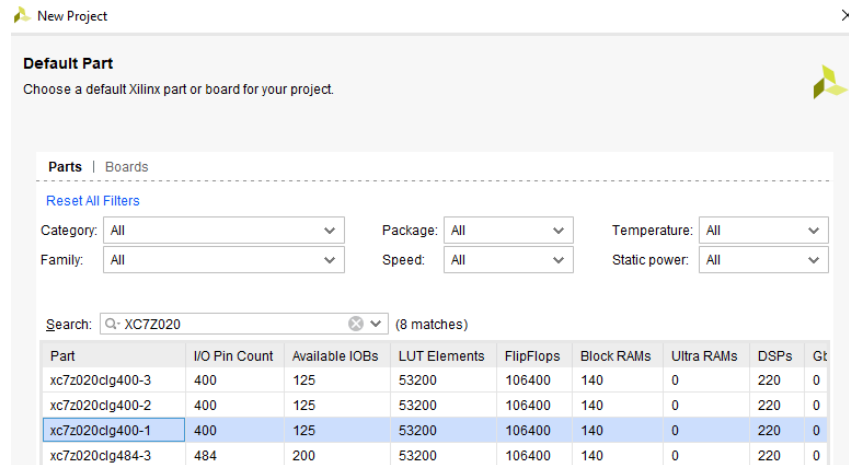
Press Next, name the project Lab1, and select any location to save it. Deselect "Create project subdirectory".



Press Next, leave the "RTL Project" setting alone and press Next again.

Press Next on both the Add Sources and Add Constraints pages (we will create these later).

Press Next, then you will be prompted to select the FPGA device we are targeting. Search for "XC7Z020" and select "xc7z020clg400-1."

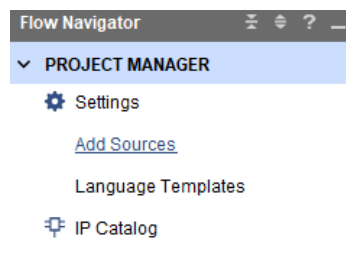


Press Next and then Finish. We now have an empty Vivado project.

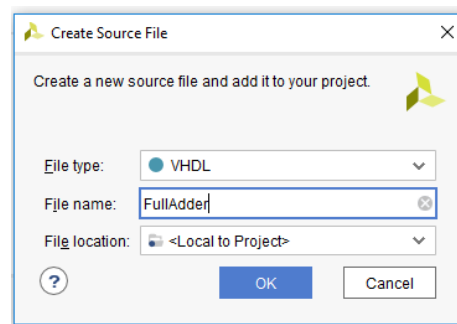
Part B: Adding new source to the project

Let's create a new VHDL module. We know that the 2-bit Ripple Carry Adder relies on Full Adders, so first we will create a Full Adder module.

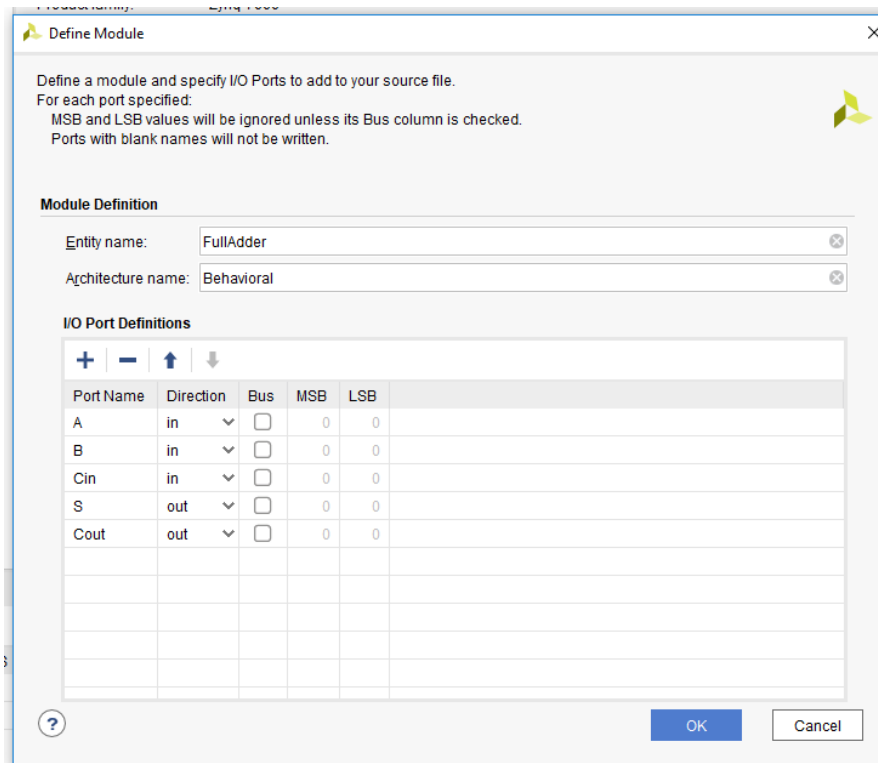
Under Project Manager on the left, press Add Sources.



Leave the "Add or Create Design Sources" button pressed and press Next. Then on the next page, press **Create File** and name the file "FullAdder".



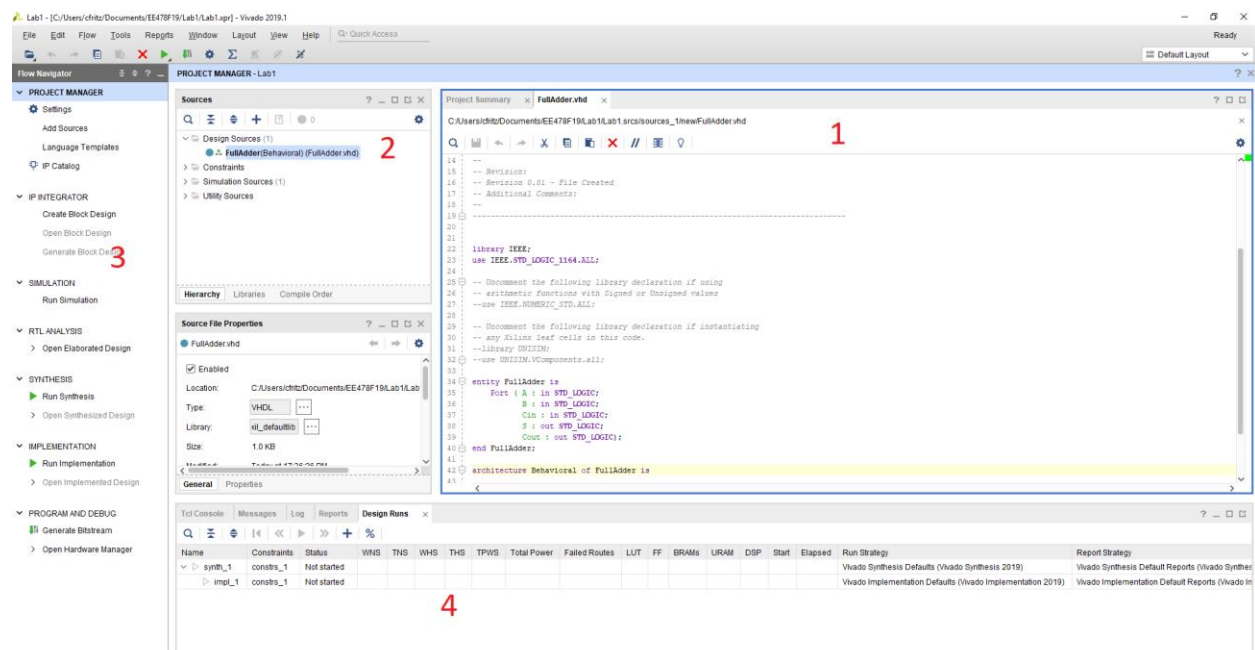
Click OK, then Finish. Then you will be prompted to specify the input and output names. Set the names as follows, and don't forget to change the direction to Out for S and Cout.




Click OK. The tool will generate the template for the VHDL module automatically for you.

Double click FullAdder under DesignSources in the Sources panel.

Let's take a look at the different parts of the Vivado GUI.



1. This is the **text editor** that allows entry of HDL code. Note that the VHDL for the Full Adder has already been started for us.
2. This is the **design hierarchy**. The module with the  symbol is the top level module. This area shows all of the modules in our design, and it shows their hierarchy – a module instantiated by another module is shown below its parent.
3. This is the **flow navigator**. Commands to simulate, synthesize, implement, and generate bitstream are all located here.
4. This is the **console**. Error messages during synthesis are displayed here. Keep an eye on this area, as every time you save your code, syntax errors will be called out here.

Note that the input and output types are **std_logic**. We'll be exploring this type more throughout the semester, but you can think of it as being synonymous with the **bit** type for now. In the lab, we'll always use the **std_logic** type.

Fill in the missing architecture for the Full Adder module. Here's a start:

```

-----
entity FullAdder is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           S : out STD_LOGIC;
           Cout : out STD_LOGIC);
end FullAdder;

architecture Behavioral of FullAdder is

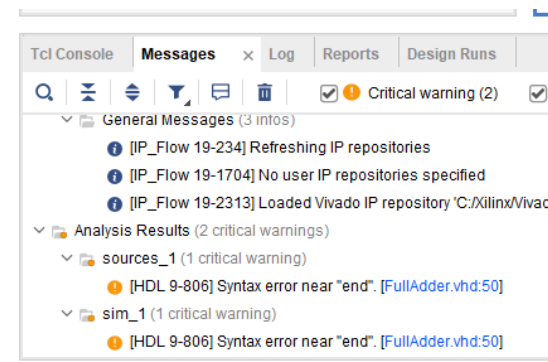
begin

    S <=
    Cout <= |

end Behavioral;

```

Now save your code (ctrl+s). Vivado will automatically analyze your code for syntax and semantic errors when you type and save. Make sure there is no red underlined code, and you can also take a look on the Messages tab – if you have any errors they should show up here.



Now we will create the 2-bit adder module. This module should have 4 inputs called A0,A1 and B0,B1. It should have 3 outputs called S0,S1,S2. Follow the same steps as above (Add Sources->Create File, etc.) using the name Adder2, and your auto generated template should look like this:

```
entity Adder2 is
    Port ( A0 : in STD_LOGIC;
          A1 : in STD_LOGIC;
          B0 : in STD_LOGIC;
          B1 : in STD_LOGIC;
          S0 : out STD_LOGIC;
          S1 : out STD_LOGIC;
          S2 : out STD_LOGIC);
end Adder2;
```

Now let's add the Component declaration for the Full Adder module so that we can instantiate Full Adders in our design. The component declaration for a Full Adder was given in class. Add it under architecture, before the word "begin".

We also need an internal signal X0 for the internal carry. Declare this internal signal under your component declaration.

Lastly, we need to instantiate two full adders. This was given in class as well. Here is an example of the first one:

```
component FullAdder is
    port (
        A, B, Cin    : in std_logic;
        S, Cout      : out std_logic
    );
end component;

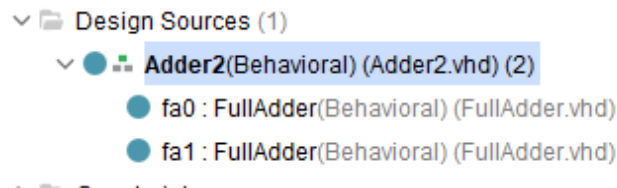
signal X0 : std_logic;

begin

fa0 : FullAdder port map(A => A0, B => B0, Cin => '0', S => S0, Cout => X0);
fa1 :
```

Finish the other adder yourself, and save the code to make sure there are no errors.

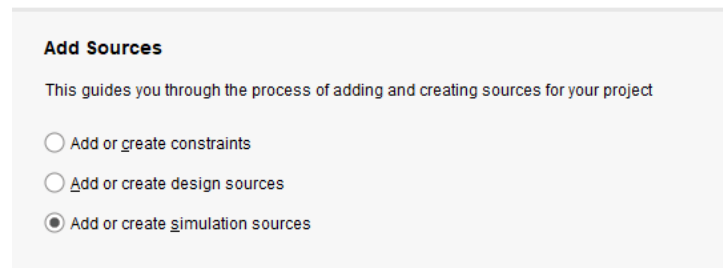
Note that the design hierarchy has changed to:



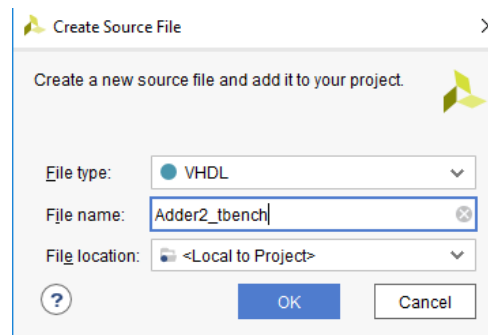
This looks correct – we have created two FullAdder instances in our design.

Part C: Creating a testbench

Now we will simulate our design to make sure it is working as a 2-bit adder. Again, press Add Sources, but this time we will select “Add or Create Simulation Sources.”

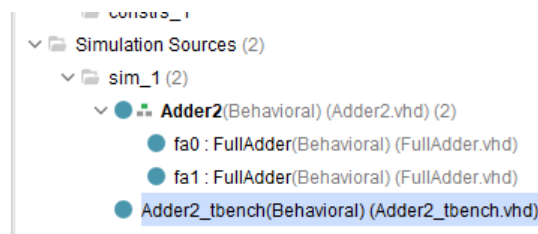


On the next page, press Create File and name the file “Adder2_tbench”.



Press OK and then Finish. We don’t need any inputs or outputs for the testbench, so just press OK on the Define Module window.

Find the testbench file under Simulation Sources and double click it to open:



The testbench will instantiate our Adder2 module as a component, so add a component declaration for Adder2. Also declare internal signals to hook up to all of the inputs and outputs:

```

entity Adder2_tbench is
-- Port ( );
end Adder2_tbench;

architecture Behavioral of Adder2_tbench is

component Adder2 is
    port (A0, A1, B0, B1 : in std_logic;
          S0, S1, S2      : out std_logic
    );
end component;

signal A0, A1, B0, B1, S0, S1, S2 : std_logic;

```

Then save the testbench and make sure there are no syntax errors.

Now we need to instantiate the Adder2 which is our **Unit Under Test (UUT)**:

```

uut : Adder2 port map (
    A0 => A0,
    A1 => A1,
    B0 => B0,
    B1 => B1,
    S0 => S0,
    S1 => S1,
    S2 => S2
);

```

Now we need to add some changes to the inputs so that we can see the adder working properly. Let's add two test cases:

- First, we'll try adding 01 + 01 (1 + 1). We expect to get 010 (2).
- Next, we'll try adding 11 + 11 (3 + 3). We expect to get 110 (6).

We'll add these two cases under a new process called stim_proc, with 100 ns in between the two test cases. Your stimulus process should look like this:

```

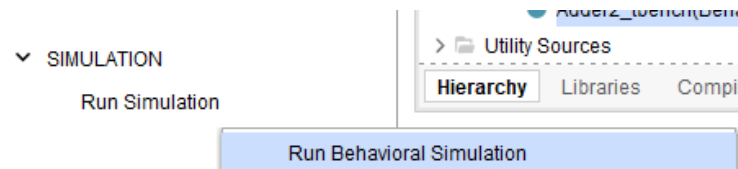
stim_proc : process
begin
    wait for 100 ns;
    A0 <= '0';
    A1 <= '1';
    B0 <= '0';
    B1 <= '1';

    wait for 100 ns;
    A0 <= '1';
    A1 <= '1';
    B0 <= '1';
    B1 <= '1';
    wait;
end process stim_proc;

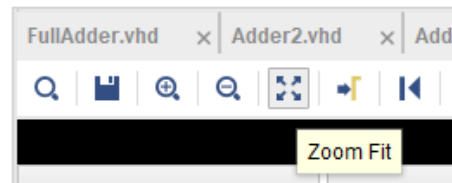
```

Save your code and fix any syntax errors.

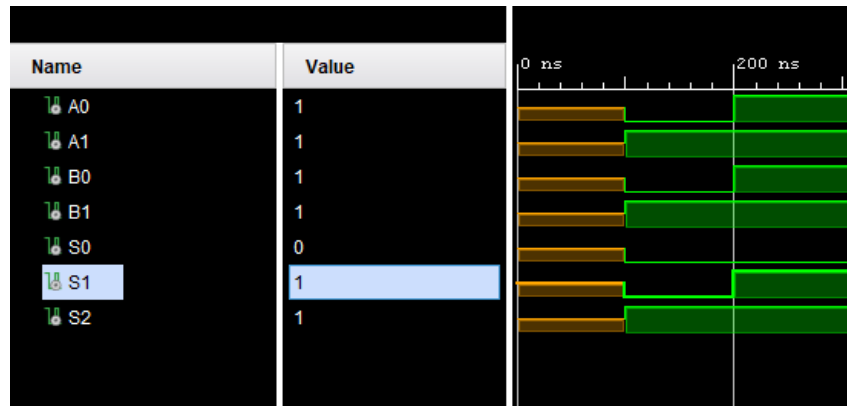
Now let's run the simulation. Press **Run Simulation** and then **Run Behavioral Simulation** under the Flow Navigator:



The simulation automatically runs for 1us, which is long enough for us to see both transitions for our test cases. We'll be getting more comfortable using the simulator throughout the semester. For now, zoom to the full wave display by clicking the Zoom Fit button on the top bar:



You should see the output signals S0, S1, and S2 change to the expected values (001 and 110) at 100 and 200 ns, respectively.



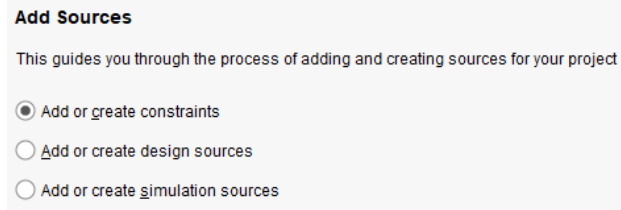
Keep this plot available as a screenshot to show your TA for demo credit.

Part D: Constraints

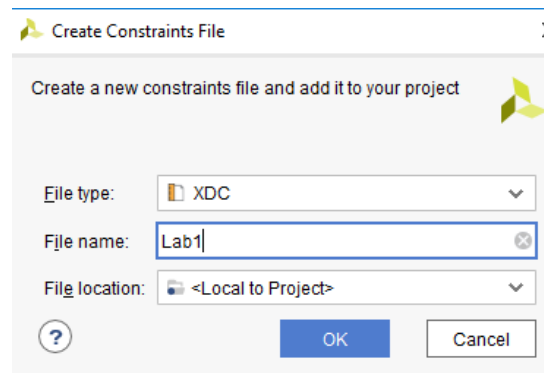
Before we synthesize the design for programming the Zybo board, we need to map the switch inputs and LED outputs to the actual pins the devices are connected to. We do this by adding an XDC Constraints File to the project.

Constraints are commands to Vivado that tell it how to connect our design to the target hardware. In general, we use constraints for *pin mapping* and *timing verification*. In this lab, we'll just do pin mapping.

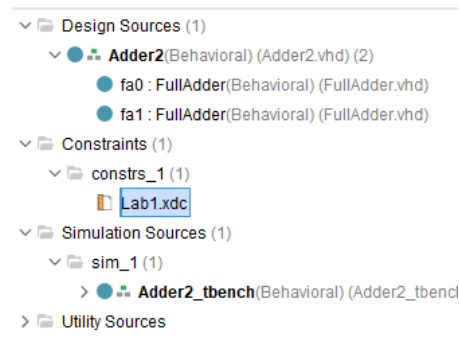
Press Add Sources and then select "Add or Create Constraints."



Press Next, then Create File, and call the constraints file “Lab1”.



Press Finish, then find Lab1.xdc under Constraints in the design hierarchy. Double click to open (you may have to switch to the Sources tab on top)



Now enter the following lines in the blank text file. Note that we are mapping each input and output signal in our design to the pin connected to the switch/LED on the board. You can find the pin numbers on the board next to each LED/Switch. *Fill in the pin number for S0 yourself by looking at the board.* What do you think the IOSTANDARD line means? What other things might you want to be able to configure about the IO pins in the design?

```

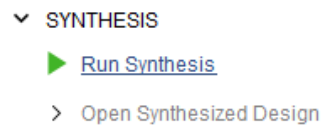
1  set_property PACKAGE_PIN T16 [get_ports {A1}]
2  set_property IOSTANDARD LVCMOS33 [get_ports {A1}]
3  set_property PACKAGE_PIN W13 [get_ports {A0}]
4  set_property IOSTANDARD LVCMOS33 [get_ports {A0}]
5  set_property PACKAGE_PIN P15 [get_ports {B1}]
6  set_property IOSTANDARD LVCMOS33 [get_ports {B1}]
7  set_property PACKAGE_PIN G15 [get_ports {B0}]
8  set_property IOSTANDARD LVCMOS33 [get_ports {B0}]
9  set_property PACKAGE_PIN G14 [get_ports {S2}]
10 set_property IOSTANDARD LVCMOS33 [get_ports {S2}]
11 set_property PACKAGE_PIN M15 [get_ports {S1}]
12 set_property IOSTANDARD LVCMOS33 [get_ports {S1}]
13 set_property PACKAGE_PIN [get_ports {S0}]
14 set_property IOSTANDARD LVCMOS33 [get_ports {S0}]
15

```

Press ctrl+s to save this file.

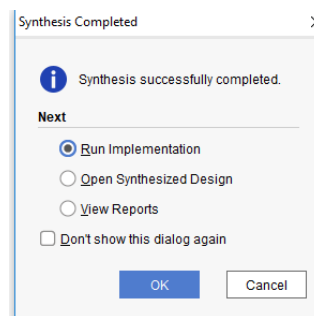
Part E: Synthesize and implement design and generate programming file

Now we are finally ready to synthesize and implement the design and program our board. Click the Run Synthesis button in the Flow Navigator and press OK on the window that opens next.

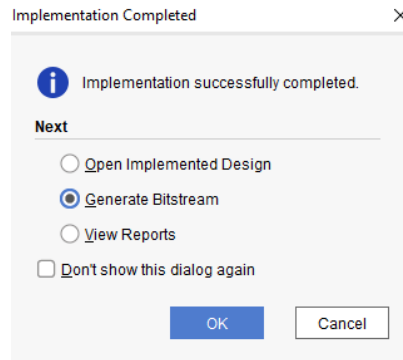


Vivado will Synthesize the design. The current process is indicated in the Design Runs tab on the bottom of the GUI.

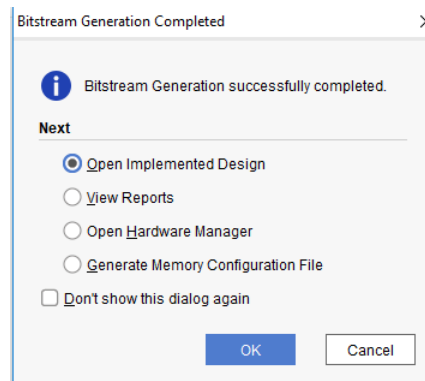
Press OK on the Run Implementation window when it opens and press OK on the next window as well:



Once Implementation completes, select the Generate Bitstream option and press OK, and then OK again on the following window.



Once Bitstream generation is complete, press OK and the implemented design will open.



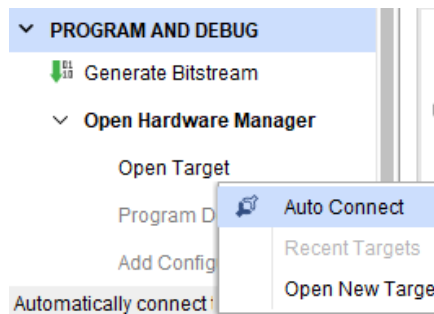
We'll take a closer look at the steps that are taken to generate the programming file (Place and Route, Mapping, etc) in a future lab.

Part F: Program Zybo board and test

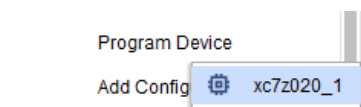
Finally, we will program our design onto the Zybo board and test it. Using your USB Micro cable, plug the board into the PC. You will connect to the "PROG UART" micro USB port on the board. Turn power on to the board with the slider switch above the USB port and note that the board starts performing a self test. *If your board does not turn on, make sure that the blue jumper is on the bottom two pins in the USB configuration.*



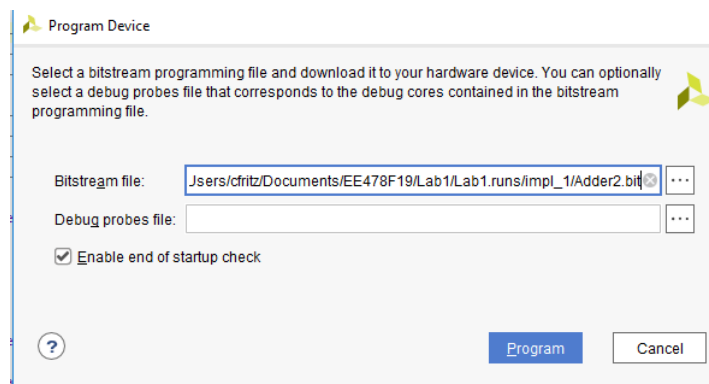
Press “Open Target” under “Open Hardware Manager” under the Flow Navigator, then press “Auto Connect”:



Lastly, press “Program Device” and then “xc7z020_1” under the Flow Navigator:



The .bit file we generated before will automatically be selected so just press Program:



Wait for the programming to complete (a few seconds), and then test out the 2-bit adder. The first two switches are A1:A0 and the last two are B1:B0. Try a few test cases, then call your TA over for demo credit.

Show your working 2-bit adder to your TA as well as a screenshot of your simulation.