Project 0: Logic Gates

E210/B441, Spring 2021

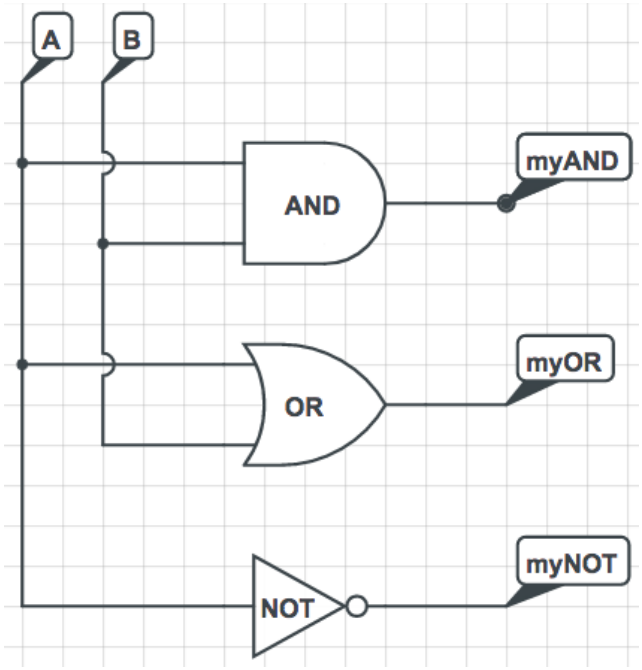
Version 2021.0

Due Date: None

Overview

This tutorial shows the steps in a digital design project using Xilinx Vivado design suite and Digilent Basys 3 FPGA board. You will learn how to use Vivado tools to create a design and implement it on the Basys3's FPGA. This is a starter project with very little hands-on work, but it is a good reference if you ever forget how to start and complete a lab project.

Logic gates are the foundation of all computer systems. They allow for the application of logical processing to be mapped onto physical circuits. In this lab project you will design and implement a digital system that uses three basic logic gates: the AND gate, the OR gate, and the NOT gate. The logic schematic of the digital system is given below.



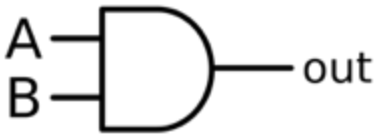
Background

AND Gate

The truth table for a 2-input AND gate is:

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

And is denoted by the following schematic symbol:



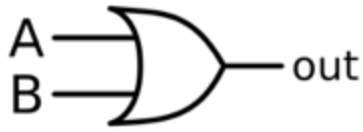
The verilog keyword is: **&**

OR Gate

The truth table for a 2 input OR gate is:

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

And is denoted by the following schematic symbol:



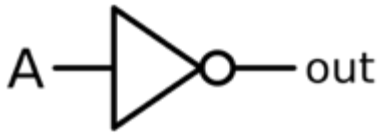
The verilog keyword is: |

NOT Gate

The truth table for a 1-input NOT gate is:

A	Out
0	1
1	0

And is denoted by the following schematic symbol:



The verilog keyword is: ~

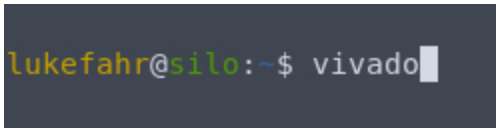
Launching Vivado

To launch Vivado, you will need to be logged into one of the Luddy Linux Machines. These are available in 4111 IF(Luddy).

Remote users are encouraged to follow the tutorial here:

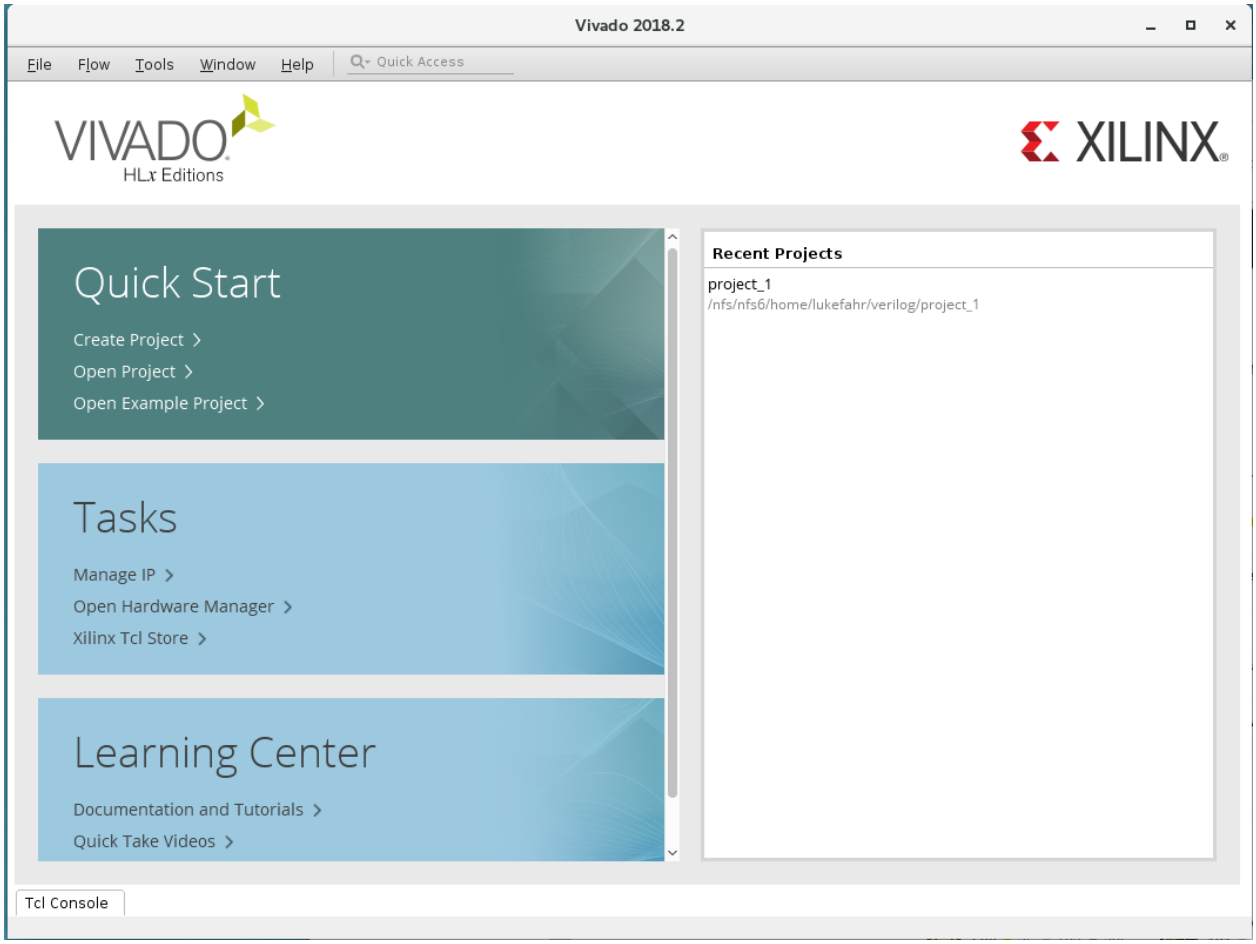
Running Graphical Programs Remotely

To launch Vivado, you should only need to type ‘vivado’ into the command line:

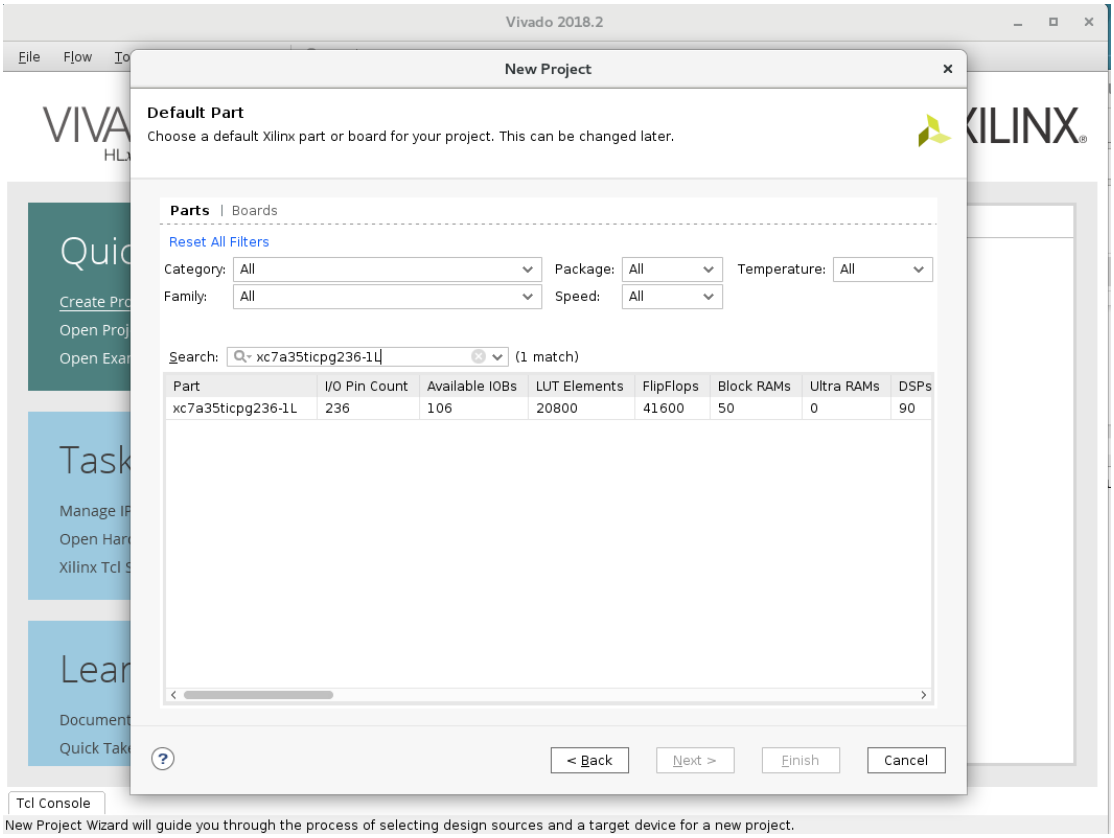


Creating a Vivado Project

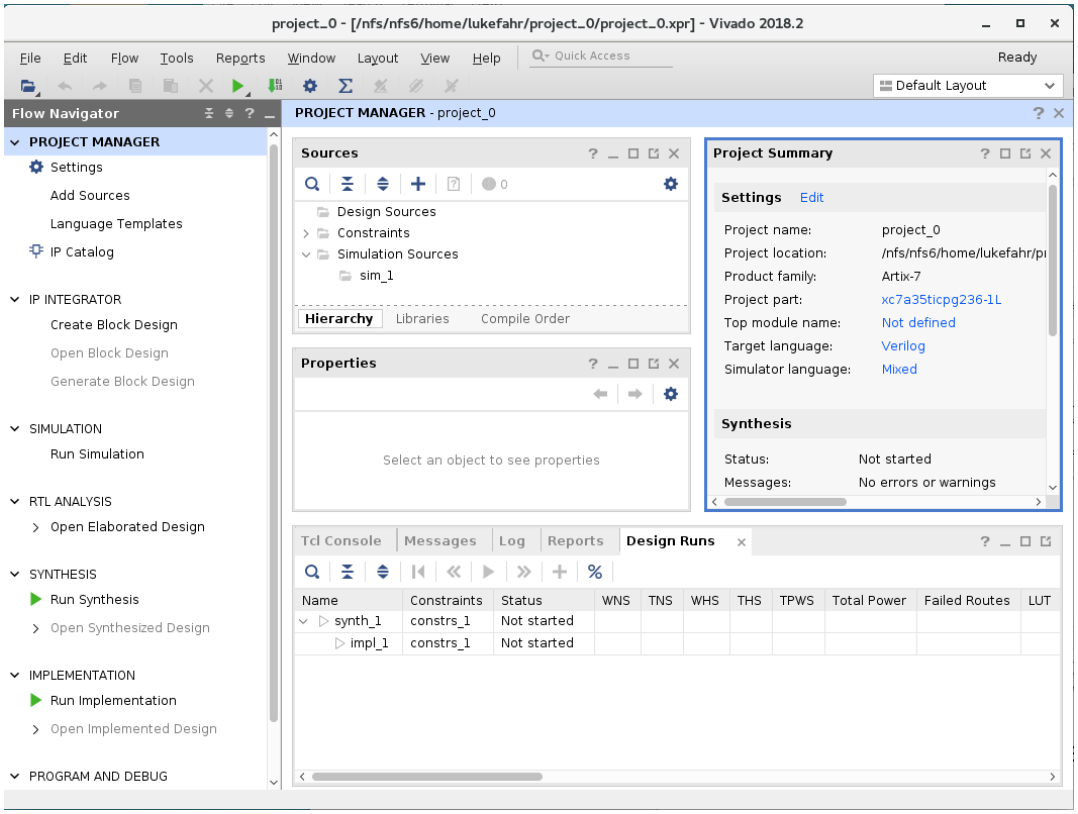
- Start Vivado and create a new project.



- Follow the wizard. Browse and select a folder in which your project will be created.
- Give a name to your project, for example Project0.
- Check “Create project subdirectory.”
- For Project Type, select “RTL Project.”
- You do not need to add any sources at this time.
- You do not need to add any constraints at this time.
- Search for part “*xc7a35ticpg236-1L*.”



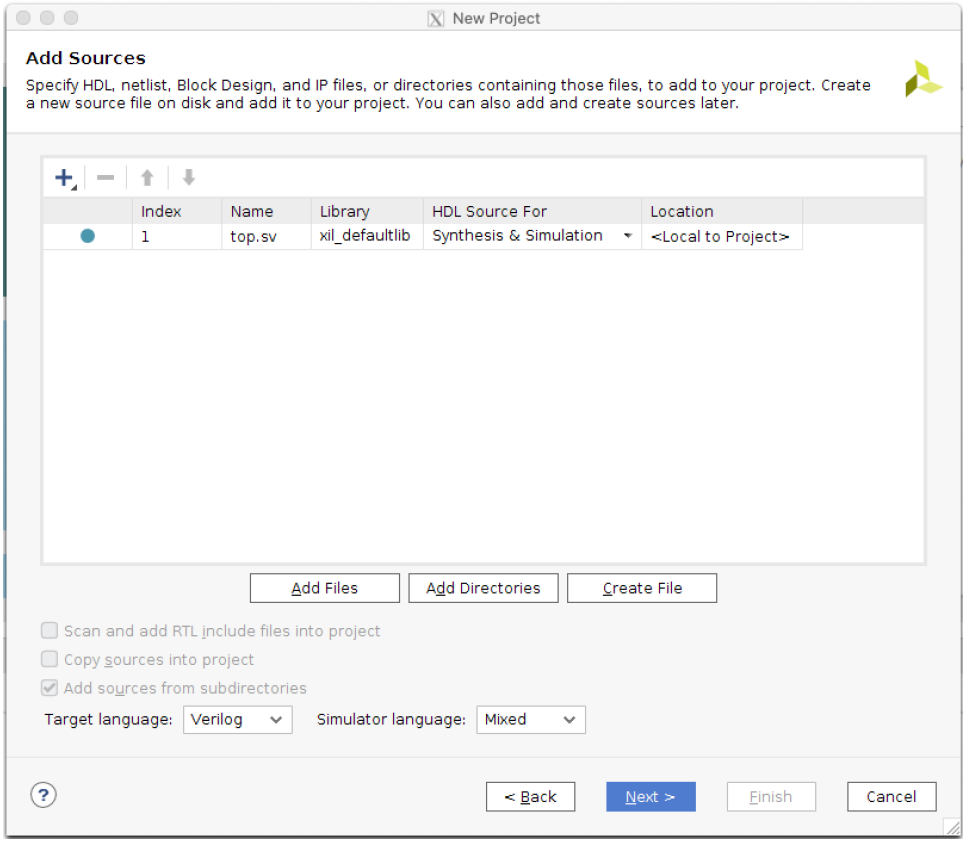
- Now click ‘Finish’
- Your screen should look like this:



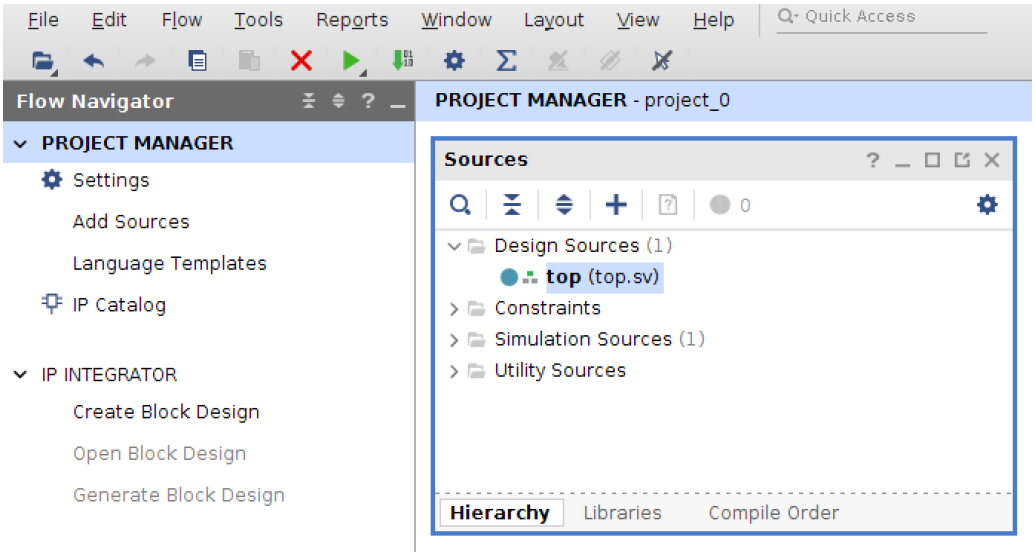
Creating a Design Source File

Now we will add our first SystemVerilog source file

- Right click on “Add Sources” on the left side window.
- Check “Add or create design sources”, then “Next.”
- Click “+” on the left hand menu, and select “Create file.” In the dialog box enter the name of the file, to make it easier, give it the name “**top.sv**” This will be our ‘top’-level ‘verilog’ file.
- **Select ‘System Verilog’ from the ‘File type’ Menu.** The “System” Verilog extension lets us use modern Verilog features. We will discuss this more later in the class. If you forget to do this, you can modify the File type later through the ‘properties’ menu.
- Click “Finish.”
- It should look something like this:



- If you get a “Define Module” prompt, just click “OK”.
- File “top.sv” is now in Design Sources under the Project Manager Window.

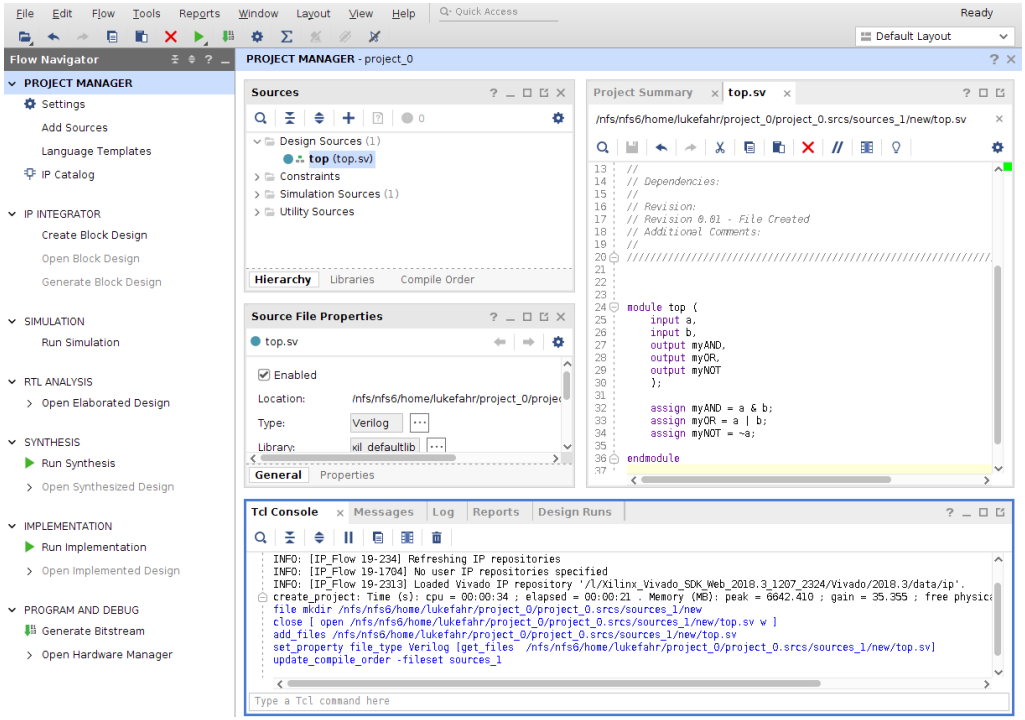


- Double click on “*top.sv*” is the *Design Sources*.
- Enter the following Verilog statements:

```
module top (
      input a,
      input b,
      output myAND,
      output myOR,
      output myNOT
      );

      assign myAND = a & b;
      assign myOR = a | b;
      assign myNOT = ~a;

endmodule
```

- This code has two inputs (`a` and `b`) and three outputs (`myAND`, `myOR`, and `myNOT`). The `assign` statements tell the FPGA how to map the output signals given the input signals. In this case, we are using boolean logic operations (`&`, `|`, and `~`) to compute the logical `and`, `or`, and `not` of the input signals.

- When you are done it should look something like this:



- Save the file (`Ctrl^S`)

Creating a Constraints File

Now we will add our first constraints source file. This tells Vivado how the inputs and outputs of the verilog code map to the real inputs and outputs of the Basys3 board.

Verilog Signal	Basys switch/LED
a	sw0
b	sw1
myAND	led0
myNOT	led1
myOR	led2

- Click on "*Add Sources*."
- Select "*Add or create constraints*"
- Click + on the left hand menu, and select "*Create file*." In the dialog box enter the name of the file, to make it easier, give it a name "***constraints.xdc***"
- Click "*Finish*"
- File "*constraints.xdc*" is now in *Constraints -> constrs_1* under the Project Manager Window.
- Double click on "*constraints.xdc*"
- Enter the following statements:

```
set_property PACKAGE_PIN V17 [get_ports {a}]
    set_property IOSTANDARD LVCMOS33 [get_ports {a}]

set_property PACKAGE_PIN V16 [get_ports {b}]
    set_property IOSTANDARD LVCMOS33 [get_ports {b}]

set_property PACKAGE_PIN U16 [get_ports {myAND}]
    set_property IOSTANDARD LVCMOS33 [get_ports {myAND}]

set_property PACKAGE_PIN E19 [get_ports {myOR}]
    set_property IOSTANDARD LVCMOS33 [get_ports {myOR}]

set_property PACKAGE_PIN U19 [get_ports {myNOT}]
    set_property IOSTANDARD LVCMOS33 [get_ports {myNOT}]
```

- This tells Vivado which physical pins (or `PACKAGE_PIN`) you want your verilog signals to map to. A complete listing of what each pin does will be provided for you.
- Save the file (Ctrl^S)

Later in the semester, we will make use of the "`Basys3_Master.xdc`" constraints file uploaded on the website. It can be found under the `Downloads` quick link.

Create Simulation File (AKA a 'testbench')

We are now going to run a simulation to ensure that our verilog code is correct.

- Right click "Add Sources."
- Select "Add or create simulation sources" hit Next.

- Click + on the left hand menu, and select “Create file.” In the dialog box enter the name of the file, to make it easier give it a name “top_tb.sv”. *Select “System Verilog” from the “File Type” drop-down menu.* This will be our “top” level testbench. The “tb” is shorthand for “test bench”.
- Click “Finish.”
- Again, if you get a Module popup, just click OK.
- File “top_tb.sv” is in Simulation Sources -> sim_1.
- Note that “top.sv” is *also* in Simulation Sources -> sim_1. This is normal.
- Double click on “*top_tb.sv*”.
- Enter the following Verilog statements into top_tb.sv:
- (please copy + paste this code, don’t try to rewrite it)

```
`timescale 1ns/1ps //specify how detailed of a simulation we want to run
module testbench();

      logic a, b; //logic (or ‘reg’) for module inputs
      wire myand, myor, mynot; //wires for module outputs

      //
      // connect our module for testing
      // This is sometimes called a DUT - Device Under Test
      //
      top top0 (
      .a(a),
      .b(b),
      .myAND(myand),
      .myOR(myor),
      .myNOT(mynot)
      );

      initial begin
      //print out these signals whenever anything changes
      $monitor("a:%h b:%h myand:%h myor:%h mynot:%h",
                  a, b, myand, myor, mynot);
```

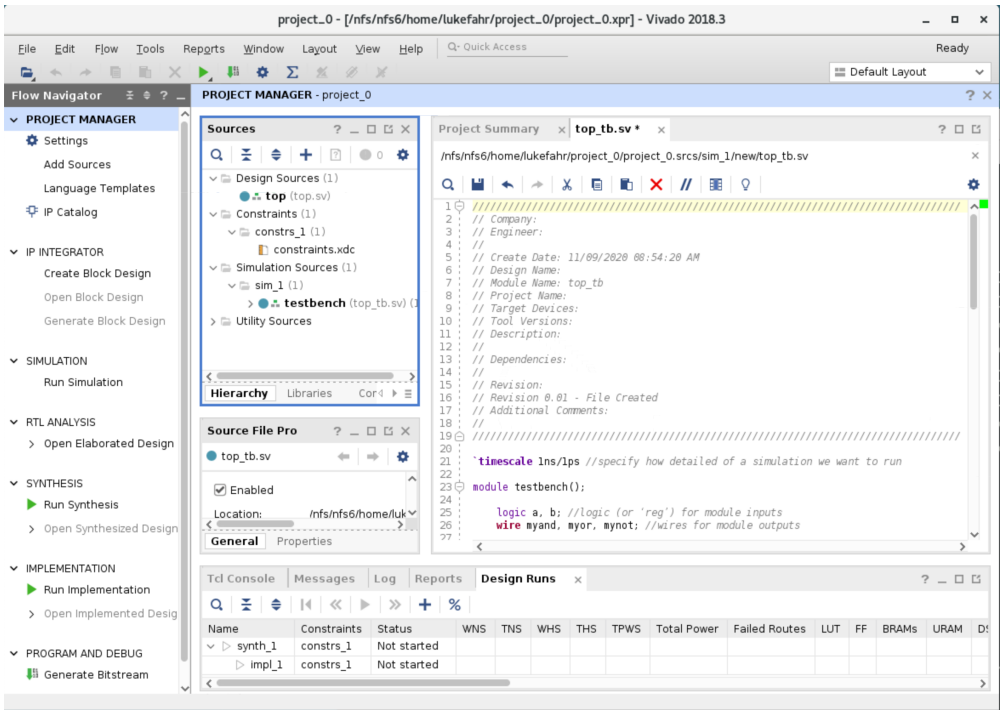
```
#100 // a 100 nanosecond delay
a = 0; b = 0; //set inputs a and b to logical 0
#100 // a 100 nanosecond delay
// check if any of our outputs are incorrect.
// if so, fail the simulation with an error message
assert( myand == 0) else $fatal(1, "myand");
assert( myor  == 0) else $fatal(1, "myor");
assert( mynot == 1) else $fatal(1, "mynot");

#100 // a 100 nanosecond delay
a = 0; b = 1; // now set a=0, b=1, and repeat the test
#100 // a 100 nanosecond delay
assert( myand == 0) else  $fatal(1, "myand");
assert( myor  == 1) else  $fatal(1, "myor");
assert( mynot == 1) else  $fatal(1, "mynot");

#100
a = 1; b = 0; //continue testing input combinations
#100
assert( myand == 0) else $fatal(1, "myand");
assert( myor  == 1) else $fatal(1, "myor");
assert( mynot == 0) else $fatal(1, "mynot");

#100
a = 1; b = 1; //last input combination
#100
assert( myand == 1) else $fatal(1, "myand");
assert( myor  == 1) else $fatal(1, "myor");
assert( mynot == 0) else $fatal(1, "mynot");



#100
```

```
        // now we've tested all possible input combinations to
        // ensure correct output for each
        $display("@@@Passed");
        $finish;
        end
endmodule
```

- Save the file (`Ctrl^S`)

Project Checkpoint

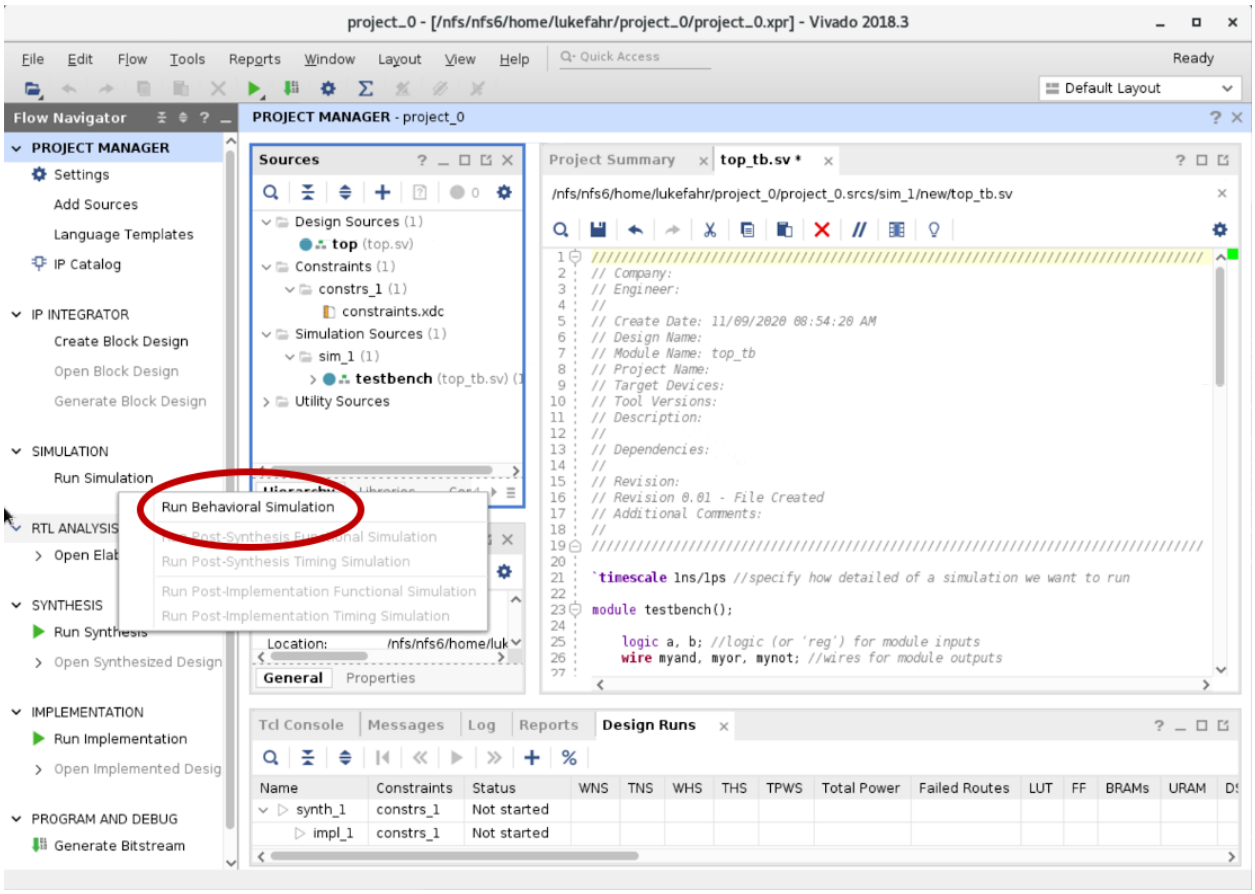
Now we should have all of our source files. Your setup should look like the following:



Running a Simulation

Now we are going to run our simulation, and use the testbench to drive inputs and observe the outputs of our verilog module.

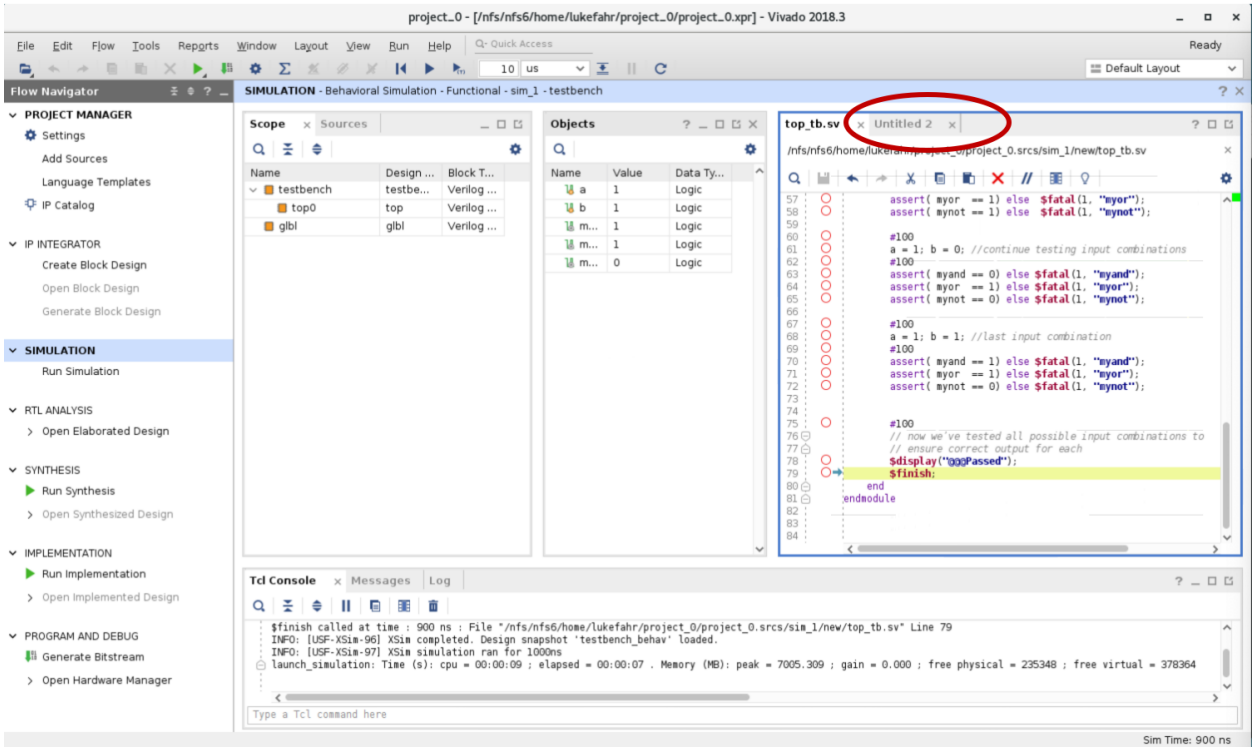
- On the left hand menu select "*SIMULATION*" -> "*Run Simulation*", then "Run Behavioral Simulation."



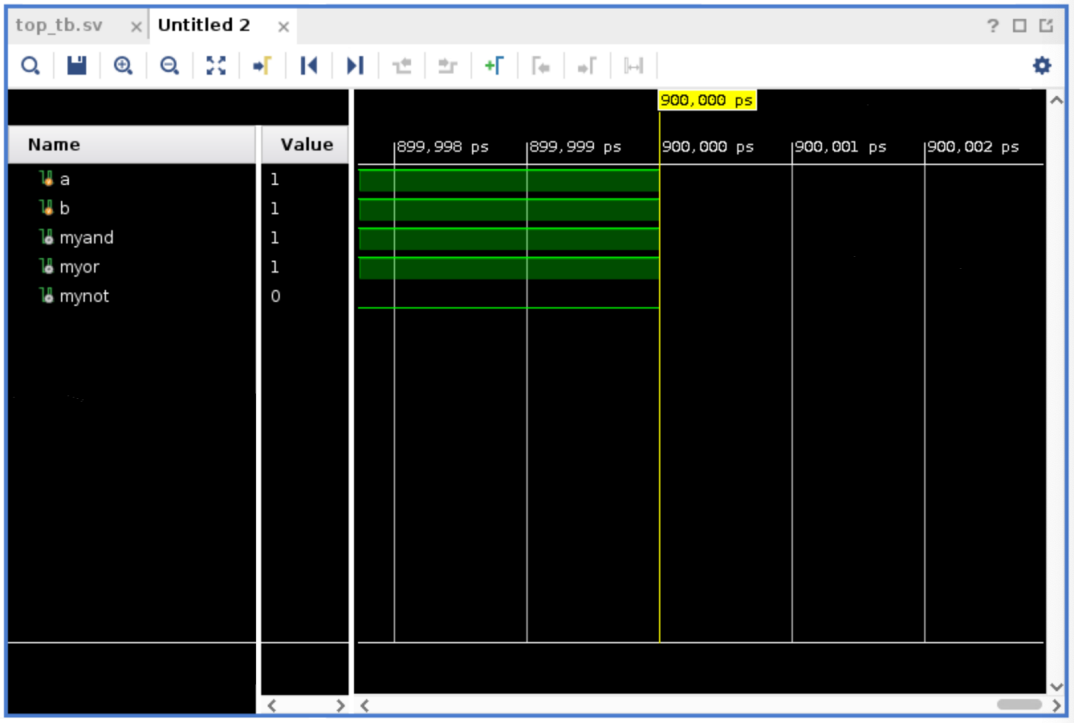
- This will take a few seconds
- When it is finished, you should get a window that looks something like this in the bottom window. Note the `@@@Passed`. This means our simulation, well, passed.

```
source testbench.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want t
#   }
# }
# run 1000ns
a:x b:x myand:x myor:x mynot:x
a:0 b:0 myand:0 myor:0 mynot:1
a:0 b:1 myand:0 myor:1 mynot:1
a:1 b:0 myand:0 myor:1 mynot:0
a:1 b:1 myand:1 myor:1 mynot:0
@@@Passed
$finish called at time : 900 ns : File "/nfs/nfs6/home/lukefahr/project_0/project_0.srcs/sim_1/new/top_tb.sv" Line 79
xsim: Time (s): cpu = 00:00:17 ; elapsed = 00:00:06 . Memory (MB): peak = 7003.309 ; gain = 59.848 ; free physical = 235373
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:30 ; elapsed = 00:00:21 . Memory (MB): peak = 7003.309 ; gain = 68.855 ; free physi
```

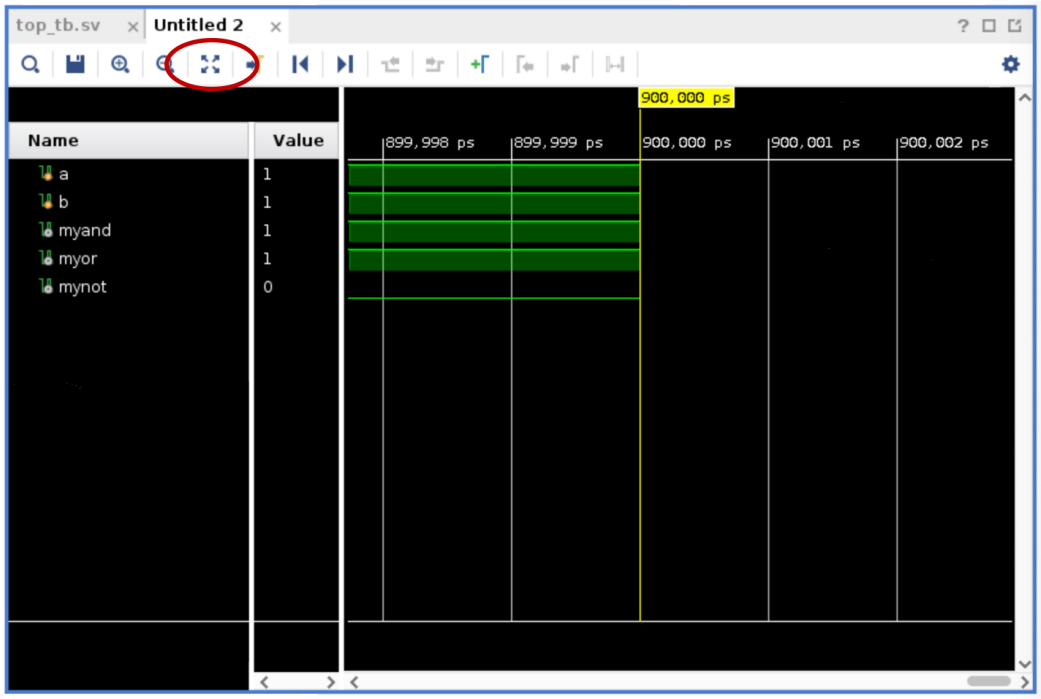
- Now let's try to understand *why* it passed. For that, we need to look at the "Wave Window". Vivado automatically created one for you (circled below). Click on it.



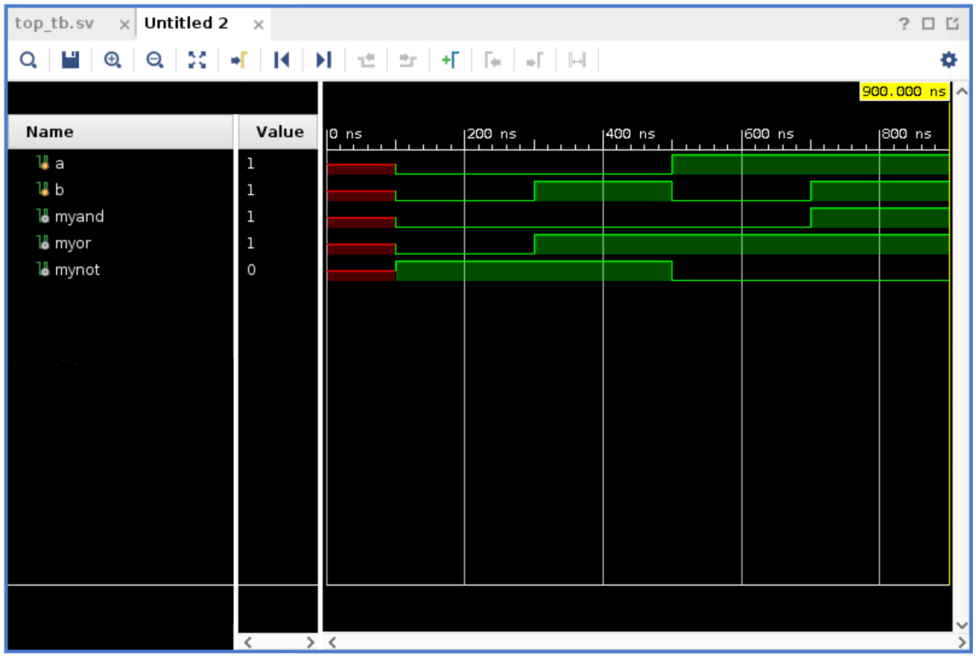
- It should look something like this:



- Now click on the  button:



- Now your wave window should display the entire **waveform**. Waveforms allow us to observe how signals change through time. Time moves from left to right, it starts at 0ns (nanoseconds), and moves to 900ns.



- The simulator looks for an initial block to run the simulations.

```
initial begin
```

- It starts at 0ns, but everything is red. Red lines means that the simulation doesn't know the value for a signal. Before 100ns, we haven't assigned `a` or `b` to anything, so the simulation doesn't have a value for them. Because the simulation doesn't know the value for the inputs, it can't figure out the value for the outputs `myAND`, `myOR`, and `myNOT`.
- At 100ns, everything changes to green. This is because we used

```
#100 // a 100 nanosecond delay
a = 0; b = 0; //set inputs a and b to logical 0
```

  in our simulation file. We told the simulation to set `a` and `b` to 0, so now it knows their values. Once the simulation knew the value for `a` and `b`, it could calculate the value for `myAND`, because we told it

```
assign myAND = a & b;
```

  in our source file.

- FPGAs require a small amount of time to update an output signal given changes in the input. Therefore, we need another `#100` delay before the updates in `a` and `b` show up in the outputs.
- At 200ns, we test the values of myAND, myOR, and myNOT to make sure they are correct. Assert statements test a boolean equation (`myand == 0`) and does nothing if it is true. If it is false, it will terminate the simulation and report an error (`$fatal(1, "myand")`).

```
assert( myand == 0) else $fatal(1, "myand");
assert( myor  == 0) else $fatal(1, "myor");
assert( mynot == 1) else $fatal(1, "mynot");
```

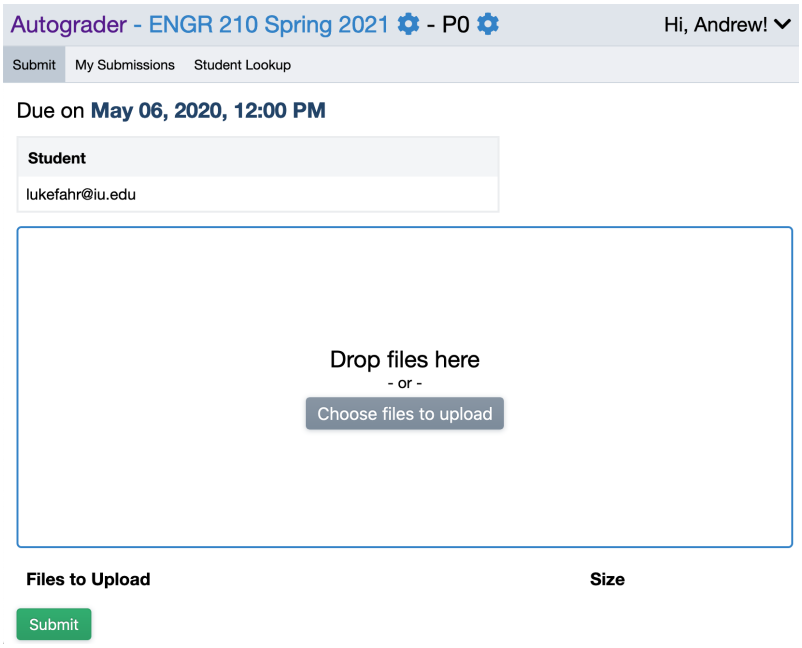
- This process repeats for every possible combination of `a` and `b`.
- Waveform simulations are a great way to visualize and understand what is happening when problems arise. We will use them frequently in this class. The autograder uses a command-line version of these same simulations for its tests.

Autograder Submission

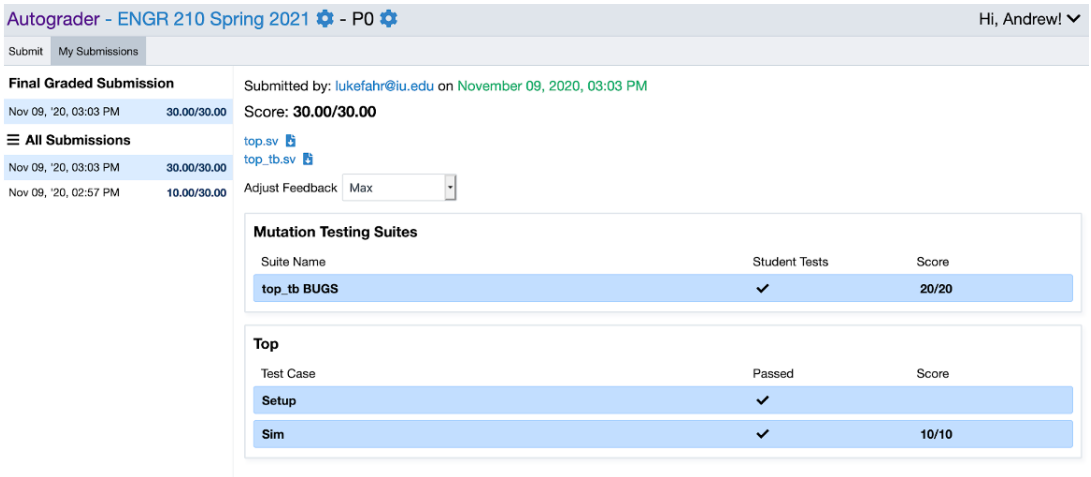
We'll be using an automatic testbench and grading system (Autograder) for the Verilog in this course. It will help you debug your code and give you live feedback on your current score for the project. Although this project is ungraded, we recommend you test out the autograder to make sure it works.

- Using a web-browser, log on to: **https://autograder.sice.indiana.edu**.
- **Log in using your 'email@iu.edu' Google Account.**

- Select the “Engr 210 Sping 2021” class
- Select “P0” from the Projects list
- You should now be at a page that looks something like this:



- Drag (or upload) both your `top.sv` and `top_tb.sv` files into the “submissions” window. These files can often be found under the “project_1.srcs” subfolder within Vivado.
- Click submit.
- You should now be taken to the ‘My Submissions’ window, where the results of your submissions will be shown shortly. It should look something like this:



- This page will display the score for both the modules that will be used for the FPGA (`top.sv`) and also the testbench modules that are used to test the code (`top_tb.sv`).

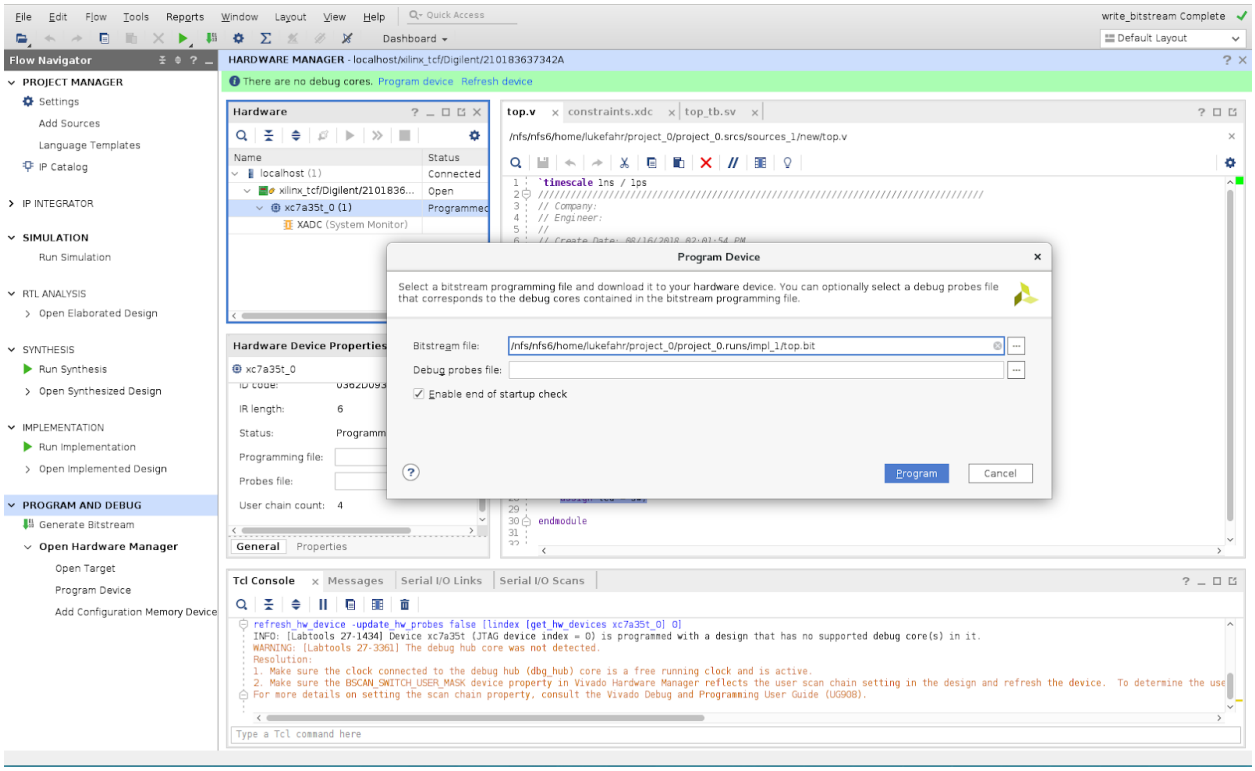
Hardware Synthesis (COVID: SKIP FOR NOW)

- Now that we're (mostly) sure our logic is correct, we can move on to "Synthesis". Synthesis is roughly equivalent to "Compiling" for FPGAs.
- On the left hand menu select "SYNTHESIS" -> "*Run Synthesis*." If prompted where to launch runs, select "Launch runs on local host'. This tells Vivado to run synthesis on your local machine.
- Even for simple designs, synthesis takes a surprisingly long time, usually 1-2 minutes. For large industrial designs, it can take days. This process translates our verilog code into LUTs, or "Look-Up Tables". It can also translate our code into basic logic gates. We'll discuss this later in class.
- After the Synthesis is complete, select "*Run Implementation*". This is also called "Auto Place and Route". This process decides which locations within the FPGA to use for each LUT, and how best to connect then.
- After the Implementation is complete, select "*Generate Bitstream*". This generates a configuration file that is read by the FPGA when it boots up to decide how to configure itself.
- After the Generate Bitstream is complete, select "Open Hardware Manager".

Programming the FPGA (COVID: SKIP FOR NOW)

This process will (finally) program the FPGA.

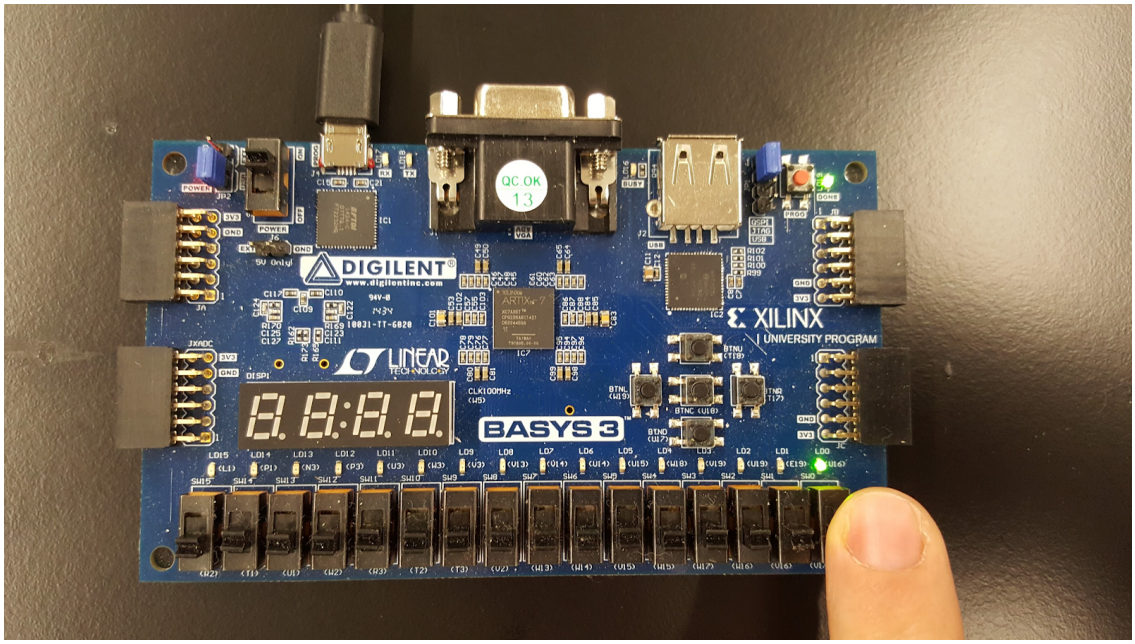
- If you haven't already, make sure your Basys3 board is connected.
- On the right-hand side, select "Programming and Debug" -> "Open Hardware Manager" -> "Open Target" -> "Autoconnect"
- This should automatically detect the Basys3
- Now select "Programming and Debug" -> "Open Hardware Manager" -> "Program Device" -> 'xc7a35t_0'. (Your device might be named differently)
- You will need to select your 'bitstream' file. Xilinx did not make this easy. The bitstream is located at './project_0.runs/impl_1/top.bit' within your project. See the example below:



- With the bitstream selected, hit “program”. This should only take a few seconds.

Testing your FPGA (COVID: SKIP FOR NOW)

You should now be able to test your FPGA. When you flip the right-most switches, the corresponding LEDs should also toggle.



This concludes the Vivado tutorial lab.