

ECE 2700 Lab 1

Due at the end of your registered lab session (100 points)

1 Objectives

- Part 1: Linux exercises.
 - Learn basic terminal command skills in Linux environment.
 - Learn scripting for task automation in Linux.
- Part 2: Become familiar with Xilinx Vivado.
 - Introduce programming in Verilog.
 - Perform testbench simulation to verify design function.
 - Deploy simple designs on the FPGA board.

2 Pre-Lab Preparation

- Before doing these exercises, follow Section 3.1 to create your own user account.
- Complete all sections listed under “Outline” of the following Linux tutorial:
<https://ryanstutorials.net/linuxtutorial>. You may skip the “Vi Text Editor” section.
- Purchase a Basys3 Board from the ECE store prior to your lab session. You should also read this document in its entirety beforehand.

3 Part 1: Linux Exercises

To pass Part 1 of this lab assignment, you must carry out the following activities using only Terminal commands. After completing each section, show your work to the TA by running the `history` command in the terminal. The `history` command will display all of the commands you have typed into the terminal, so that the TA can verify your work.

3.1 Creating a private user account

It is recommended that you create a personal user account. To do so, simply open a Terminal and use this command, and replace “[username]” with a name of your choice:

```
[ecestudent@centos7 ~]$> sudo useradd [username]
```

It will prompt for your password. Use the password for the default account that you started with. Then create a new password for your new user:

```
[ecestudent@centos7 ~]$> sudo passwd [username]
```

You will not need to enter the administrator password again. It will just ask you to create a new password for the new account. For the last step, make your new user an administrator using this command:

```
[ecestudent@centos7 ~]$> sudo usermod -aG wheel [username]
```

Once this is done, switch users to your new account by typing this command:

```
[ecestudent@centos7 ~]$> su [username]
```

In order to access FPGA and other devices, your new user needs to be added to some special system groups. The easiest way to do this is to launch the Arduino application. It will prompt for your password and automatically add you to the required system groups. Then you can quit the Arduino application.

3.2 Basic File Manipulation

1. Directory navigation: Using the `mkdir` and `cd` commands, create a directory tree that looks like this:

```
~/linux_tutorial
~/linux_tutorial/dir1
~/linux_tutorial/dir1/subdir1
~/linux_tutorial/dir1/subdir2
~/linux_tutorial/dir2
```

Then verify your directory tree by running the command

```
[ecestudent@centos7 ~]$> find -type d
```

Once you have completed this part, show your result to the TA.

2. Terminal text editing: Change to directory `subdir1`, and run the command `pwd` to verify that you are in the right directory. Then create a file called “`Readme.txt`” using the `cat` command.

```
[ecestudent@centos7 ~]$> cat > Readme.txt
```

The file's contents should be as follows: ECE students are Linux gurus. Remember that when you are done typing your text in `cat`, you end the file by pressing `Enter`, then `Ctrl-D`.

3. Copying, renaming and deleting: Using the `cp` command, make a copy of `Readme.txt` called `Copy.txt`.

```
[ecestudent@centos7 ~]$> cp Readme.txt Copy.txt
```

Then use the `mv` command to rename `Copy.txt` to `Useless.txt`:

```
[ecestudent@centos7 ~]$> mv Copy.txt Useless.txt
```

Finally, delete the copy by using the remove command:

```
[ecestudent@centos7 ~]$> rm Useless.txt
```

4. Environment variables. Many Linux programs use *environment variables* to control their configuration. Make an environment variable via the following commands::

```
[ecestudent@centos7 ~]$> x=15  
[ecestudent@centos7 ~]$> echo $x
```

Do not use any spaces around the = sign in the `x=15` statement. The `echo` command prints the environmental variable's value onto the terminal. The dollar-sign is used to indicate that `x` is a variable. Try running these commands to see the difference between `x` and `$x`:

```
[ecestudent@centos7 ~]$> echo x  
[ecestudent@centos7 ~]$> echo $x
```

3.3 Environment Variables and Basic Programming

1. Exporting variables: When you make an environment variable, it only exists in your current shell. Other programs outside your shell (such as programs you launch from the Applications menu) will not be able to see this variable. To set a variable so that it can be seen by all your programs, use the `export` command:

```
[ecestudent@centos7 ~]$> export x=15
```

all programs that you launch until you logout. The variable will not exist any more after you log out.

2. The `$PATH` and `$HOME` variables: Some variables are set by the system every time you login. These include the `$HOME` variable, which says where your user's directory is located, and the `$PATH` variable, which tells the system where to look for executable files. Try these commands:

```
[ecestudent@centos7 ~]$> echo $HOME  
[ecestudent@centos7 ~]$> echo $PATH
```

so on. It is often helpful to include the current working directory in the `PATH` listing, so that the system knows to look in your current directory for executable programs. To add this to your path, use this command:

```
[ecestudent@centos7 ~]$> export PATH=.: $PATH
```

This command simply adds the characters `“.:”` to the front of `PATH`, so that the system will always look in the `“.”` directory first when searching for commands.

3. Launching a GUI: Go to directory `subdir2`, and create a file called `hello.cpp` using the command `gedit hello.cpp &`. The “&” tells the program to run in the *background*, so that you can continue using the terminal while the editor runs. Enter the following text exactly as shown:

```
// hello.cpp
// A C++ program written for the ECE Linux Tutorial

#include <iostream>
using namespace std;

int main() {
    cout << "Hello_world_from_the_ECE_Design_Automation_Lab!\n";
    return 0;
}
```

4. Using the GNU compiler: Now, while still in `subdir2`, compile the `hello.cpp` program using the GNU g++ compiler. To do this, use this command

```
[ecestudent@centos7 ~]$> g++ hello.cpp -o hello
```

Now run your program by typing this command:

```
[ecestudent@centos7 ~]$> ./hello
```

2), then you should be able to run the program by simply typing

```
[ecestudent@centos7 ~]$> hello
```

Please try this and verify that it works.

5. Creating permanent environment variables: You will sometimes need to create environment variables and settings that are permanent – i.e. they need to be set every time you use the terminal. Permanent changes can be made by editing a file called `.bashrc` in your home directory. To permanently add “.” to your `$PATH` variable, edit your `.bashrc` file using this command:

```
[ecestudent@centos7 ~]$> cp ~/.bashrc ~/.bashrc_backup
[ecestudent@centos7 ~]$> nano ~/.bashrc
```

in case you make some mistake. Add this line to the end of the file:

```
[ecestudent@centos7 ~]$> export PATH=.: $PATH
```

To save the change, type `Ctrl-O`. To exit the editor, type `Ctrl-X`. The `$PATH` variable will now include “.” every time you open a terminal. To test your new settings, type `bash` and press return. This relaunches the bash shell within your terminal. If you did everything right, you should be able to run your program by typing `hello` from within `subdir2`. If something goes horribly wrong, and

your bash shell fails to relaunch, try typing **Ctrl-C** or **Ctrl-D** to cancel the new shell. This should take you back to your safe working shell, where you can restore your `.bashrc` file by typing:

```
[ecestudent@centos7 ~]$> cp ~/.bashrc\_backup ~/.bashrc
```

3.4 Manipulating Inputs and Outputs, and Searching

1. Output redirection: Now change directories to `dir1`. Perform a recursive directory listing using `ls`, and direct the output to a file called `listing.txt`. Then view the `listing.txt` file using the `more` command. Your command sequence should look like this:

```
[ecestudent@centos7 ~]$> ls -R > listing.txt  
[ecestudent@centos7 ~]$> more listing.txt
```

2. Pipes: Now do the listing again, only use the *pipe* operator to send the listing directly to the `more` command, without needing to create a file:

```
[ecestudent@centos7 ~]$> ls -R | more
```

3. Using `grep`: You can use the `grep` command to search for text within a file. Change to `subdir2` and run the following commands:

```
[ecestudent@centos7 ~]$> grep -n "Hello" hello.cpp  
[ecestudent@centos7 ~]$> grep -l "Hello" hello.cpp  
[ecestudent@centos7 ~]$> grep -i "heLl0" hello.cpp  
[ecestudent@centos7 ~]$> grep -in "hello" hello.cpp
```

Note the affect of the options `-i`, `-l` and `-n`. Also note that they can be used in combination.

4. Using `find`: You can search your entire directory tree using the `find` command. Change to the `linux_tutorial` directory, and run these commands:

```
[ecestudent@centos7 ~]$> find -type d  
[ecestudent@centos7 ~]$> find -type f  
[ecestudent@centos7 ~]$> find -name hello.cpp
```

The “`-type d`” option searches for directory names. The “`-type f`” option searches for regular files. The “`-name hello.cpp`” option searches for a specific filename.

5. Using `grep` and `find` together: The Bash shell allows various ways to use output from one command as input to another. For example, the `grep` syntax is

```
[ecestudent@centos7 ~]$> grep <pattern> <filename>
```

For the <filename> option, you can insert a list of files returned by the `find` command. This is done by using backward-ticks (‘) to surround the find command, like this:

```
[ecestudent@centos7 ~]$> grep -li "hello" 'find -type f'
```

Run this command from the `linux_tutorial` directory, and it should print out a list of files that contain the text “hello”. The `grep` options are “i”, which specifies a case-insensitive search, and “l”, which requests that command print out names of files that contain the word “hello”.

6. Man: Many commands, like `grep` and `find`, have a large number of options and configurations. To find out more about a command, you can type `man <command>` in the terminal (“man” is short for “manual”). Try studying the `grep` options using this command:

```
[ecestudent@centos7 ~]$> man grep
```

3.5 Scripts and Archives

1. Scripts: You can automate a sequence of commands using a *script* file. Change to the directory `dir2`. In this directory, use a text editor to create a file called `hello.sh`, with the following contents:

```
#!/bin/bash

echo "Hello World from the ECE Design Automation Lab!"

# Change to your tutorial's root directory:
cd ..

# List all the files you've created:
for x in `find -type d`
do
    echo $x "is a directory"
done

for x in `find -type f`
do
    echo $x "is a regular file"
done
```

Then, to run your script, first change the file’s permissions to allow the user (you) to execute the script:

```
[ecestudent@centos7 ~]$> chmod u+x hello.sh
```

With the permissions properly changed, you may run the script with the command

```
[ecestudent@centos7 ~]$> ./hello.sh
```

You should see that it prints out the hello message, and gives you a customized listing of your directory tree.

2. Archives: You can create an archive of your directories and files by using the `tar` command. Try running the following commands:

```
[ecestudent@centos7 ~]$> cd ~
[ecestudent@centos7 ~]$> tar -cvf linux_tutorial.tar linux_tutorial
[ecestudent@centos7 ~]$> ls
```

The `tar -c` command creates an archive called a “tarball”. The name of the tarball is specified using the `-f` option. The `-v` option tells `tar` to print verbose messages while it works (you may want to omit this when archiving large directory trees). You can *extract* the tarball using the `tar -x` command:

```
[ecestudent@centos7 ~]$> mv linux_tutorial old_tree
[ecestudent@centos7 ~]$> tar -x -v -f linux_tutorial.tar
```

This creates an exact replica of the files stored in `linux_tutorial.tar`. The extracted files are now placed in a directory called `linux_tutorial`, hence restoring the entire file tree. You can use `tar` archives to create backups of your work, to exchange your files with other people, and to transfer files between different computers.

3. Compressed archives: Because `tar` archives can be quite large, they are usually *compressed* using the `gzip` program, which reduces the file’s size. The best way to make a compressed archive is to add the `-z` option after the `tar` command:

```
[ecestudent@centos7 ~]$> tar -cvzf linux_tutorial.tgz linux_tutorial/
```

Notice that the file is now called “`linux_tutorial.tgz`” instead of “`linux_tutorial.tar`”. The `tgz` file extension indicates that the archive is compressed. You may also see files with names like “`file.tar.gz`”. This is the same as “`file.tgz`”. To expand a compressed archive, simply add the `-z` option to the `tar -x` command:

```
[ecestudent@centos7 ~]$> mv linux_tutorial old_tar_tree
[ecestudent@centos7 ~]$> tar -xvzf linux_tutorial.tgz
```

3.6 TA Checkoff

Check off the following items with your TA:

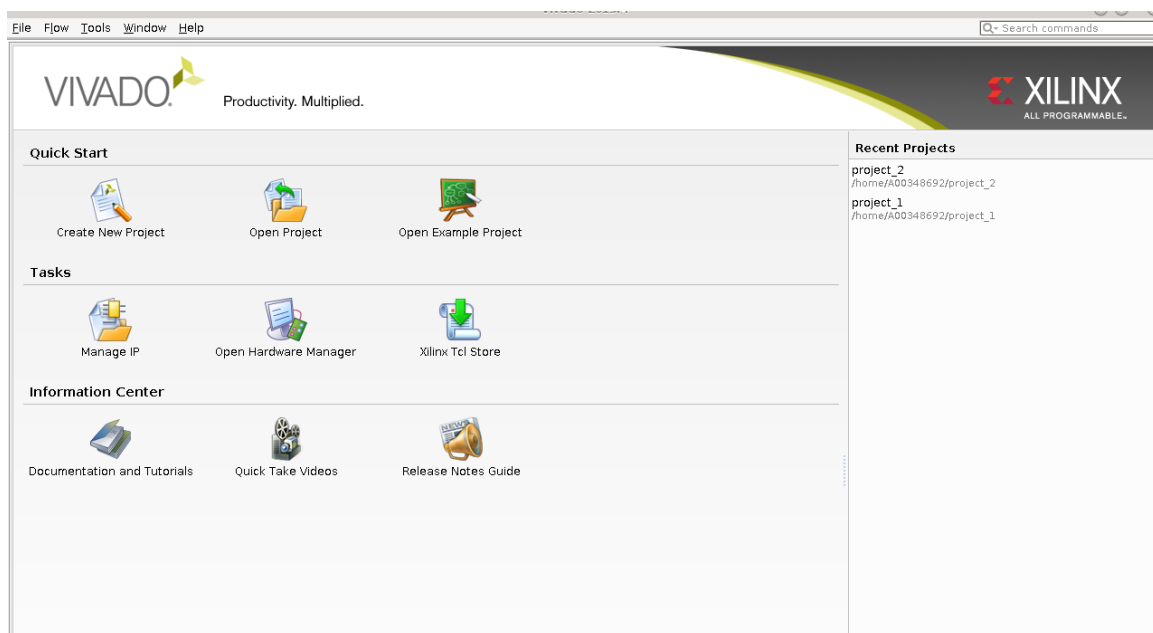
1. (10 points) Complete pre-lab work.
2. (4 points) Directory listing.
3. (4 points) `history` from Section 3.2.
4. (4 points) Program `hello.cpp` created in Section 3.4 compiles and runs.

5. (4 points) Changes to `.bashrc`.
6. (8 points) `history` from Section 3.3.
7. (8 points) `history` from Section 3.4.
8. (8 points) Script file `hello.sh` created in Section 3.5 runs properly.

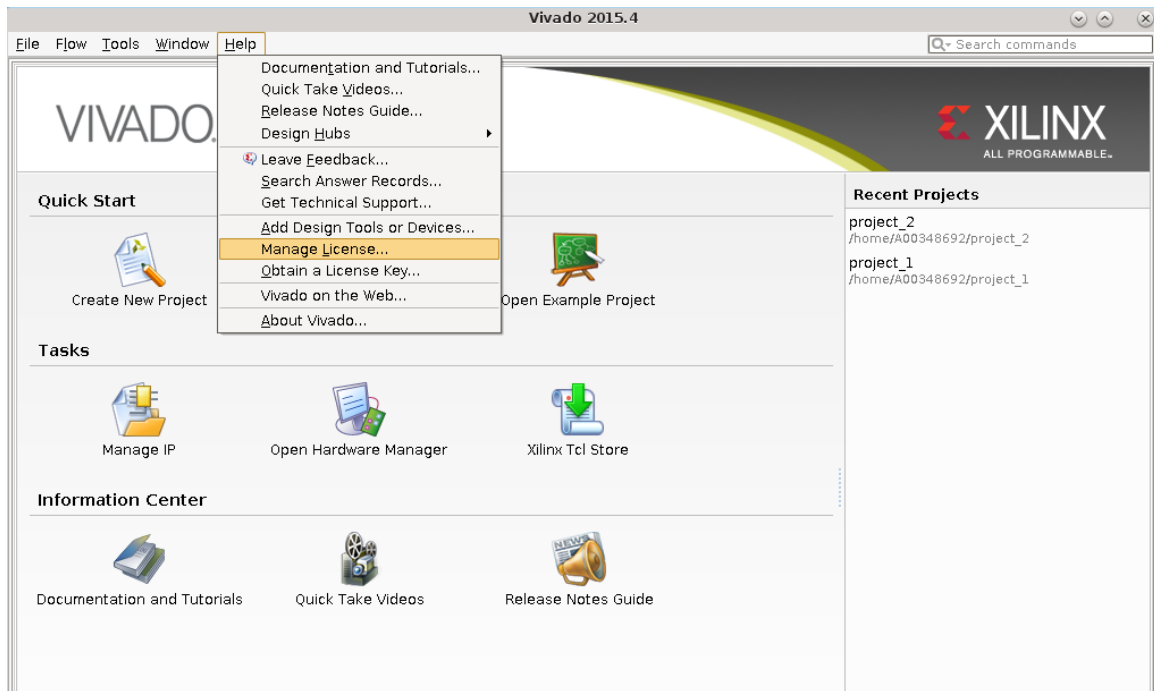
4 Part 2: Digital Design in Vivado

Inside your user home directory, create a directory called `ECE2700` or something similar. Then, create a subdirectory called `Lab1` inside your `ECE2700` directory. Maintaining a clear file organization will be useful not only for yourself, but also for your TAs.

The program we will be using is called Xilinx Vivado. To run the program, open a terminal and launch the Vivado Design Suite by clicking on the Centos logo located at the bottom left corner (similar to the “start” menu on Windows). After a moment you should see the Vivado GUI appear.



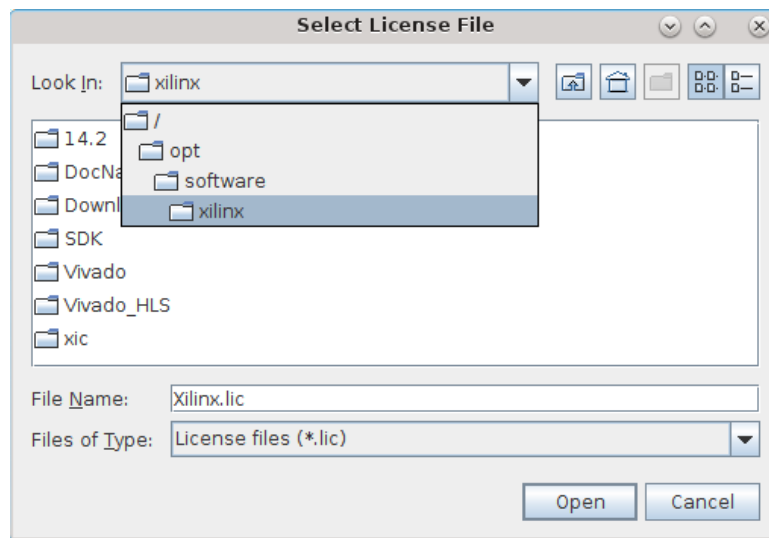
If you are using the ecestudent account on the ECE External Hard Drive, then you can skip the license setup step and proceed to Section 5. For your first time running Vivado, you need to configure the user license. In the menu bar, click Help→Manage License...



After clicking it, it should popup the Vivado License Manager. (You may see a “Trusted Storage” popup warning; you can click OK to dismiss it.) It will take some time (usually about 30 seconds) scanning all the university network licenses. Once this is finished you will be able to interact with the License Manager. Click on “Load License” in the right menu panel, then click the big “Copy License” button:



In the file browser, navigate to `/opt/xilinx/` and select the file **Xilinx.lic**. Then click **Open** to load this license into your user settings.



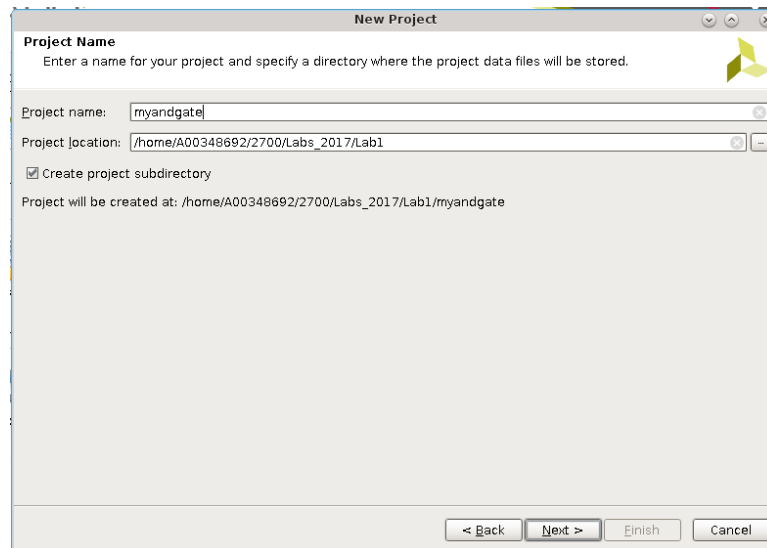
It may take a moment, but once this is complete you can close the License Manager and continue with the lab.

5 Writing Verilog Code and Simulating Your Design

Now go back to the Vivado start menu, and click **Create New Project**. For the project name, do the following:

- **Project Name:** myandgate
- **Project Location:** click on the ...button (at the far right) and navigate to your 2700 Labs subdirectory. Then click the starred-folder icon to create a new subdirectory called Lab1. Select the new Lab1 subdirectory to organize all of your Lab1 projects.

Your window settings should look something like this:

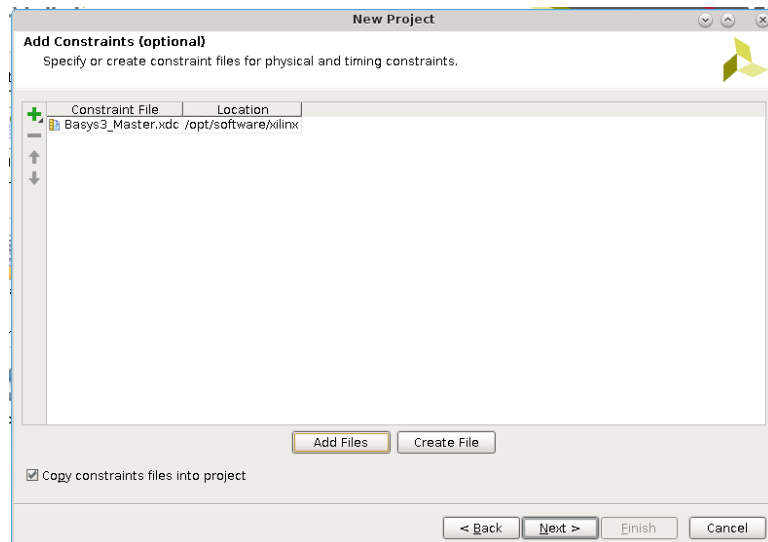


Now click **Next**, and in the next wizard screen select **RTL project**. Click **Next** again.

The next wizard screen gives you the option to **Add Sources**. In this screen, click **Create File**. A popup window should appear titled “Create Source File.” In this popup, enter the name `myandgate.v` in the **File Name** box, then press OK. Back in the New Project wizard, click **Next**.

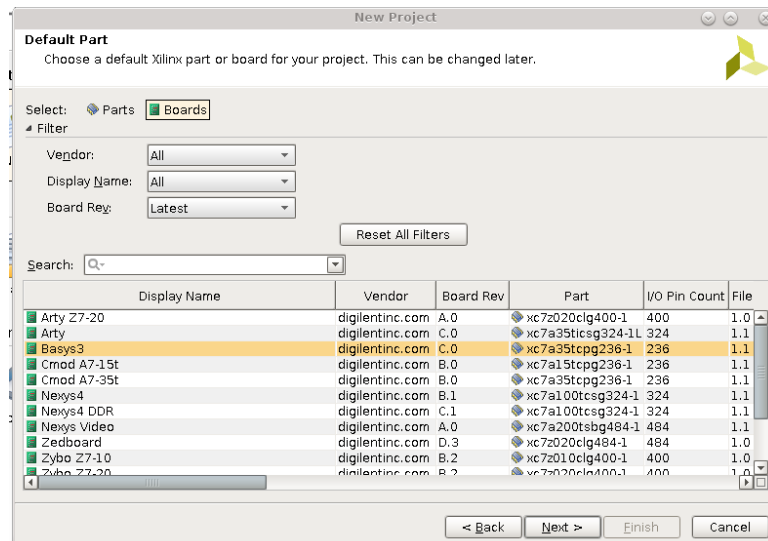
The next screen asks if you want to Add Existing IP. We won’t be doing that here, but we’ll pause to note that “IP” stands for “Intellectual Property,” and basically refers to any pre-packaged submodule or resource that you got from someone else. Click **Next**.

The next wizard screen lets you **Add Constraints**. A constraint file is a crucial part of any physical design, since it defines how your Verilog signals relate to the physical resources on the Basys3 board. Without constraints, your design is just a ghost. Download the constraint file, `textbfBasys3_Master.xdc`, from Canvas first. Then click **Add Files** to locate this file, then click OK. **In the New Project wizard, make sure the box is checked that says “copy constraints files into project.” DO NOT SKIP THIS STEP.** Your wizard screen should look like this:



Once everything looks good, click **Next**.

The next wizard screen is really important, since you need to pick the right hardware in order for the process to succeed. In the wizard screen, click **Boards**, then in the menu select **Basys3**.



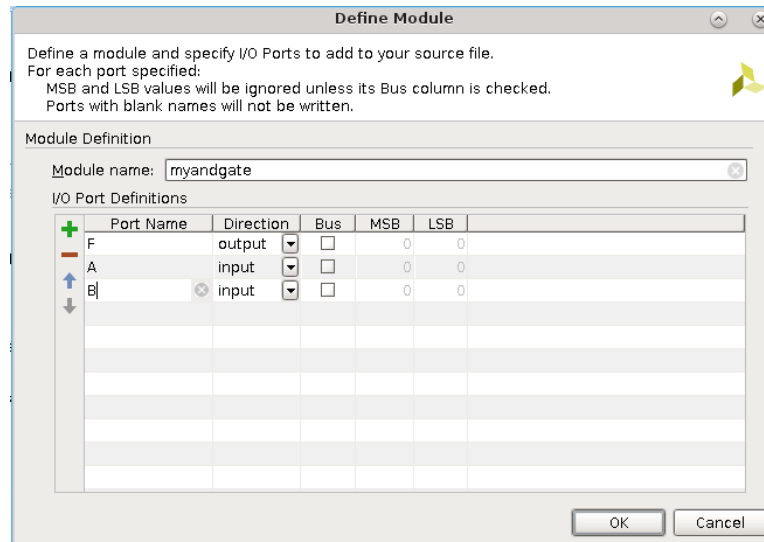
Then click **Next**.

On the final wizard screen, you will see an overview of your new project settings. Check over them and make sure they match what was described in this tutorial. If they don't, you may encounter frustrations and could have to start over again.

When you're happy with your configuration, press **Finish** to launch your new project. You will use this same basic procedure for creating all lab projects in this course.

6 Writing the Source Code

After you finish with the New Project wizard, you will often see a popup screen that encourages you to specify I/O signals for your new module. Using this window is optional (since you can just type the I/O ports directly in the Verilog file), but it can save time to enter them into this window. You want **one output named F** and **two inputs named A and B**, like this:



Click OK. Finally you should be at the main Vivado project screen. In the Project Manager, you should see a **Design Sources** tab with **myandgate** in bold. Double-click on **myandgate** to open the Verilog source file. Notice that Vivado has supplied a substantial comment-header at the top of the file; it's a good engineering habit to fill in this information (or at least some of it).

6.1 Complete the Module in Verilog

In the source listing below, notice the indicated line. Add this line into your file to create a structural description of an AND gate using the Verilog primitive.

```
'timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
// Company: Utah State University
// Engineer: ECE2700 Instructor
//
// Create Date: 08/23/2017 04:31:24 PM
// Design Name: Lab1
// Module Name: myandgate
// Project Name: myandgate
// Target Devices: Basys3
// Tool Versions: Vivado 2015.4
// Description:
//   a trivial module to implement an and gate
// Dependencies:
//   none.
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////

module myandgate(
    output F,
    input A,
    input B
);

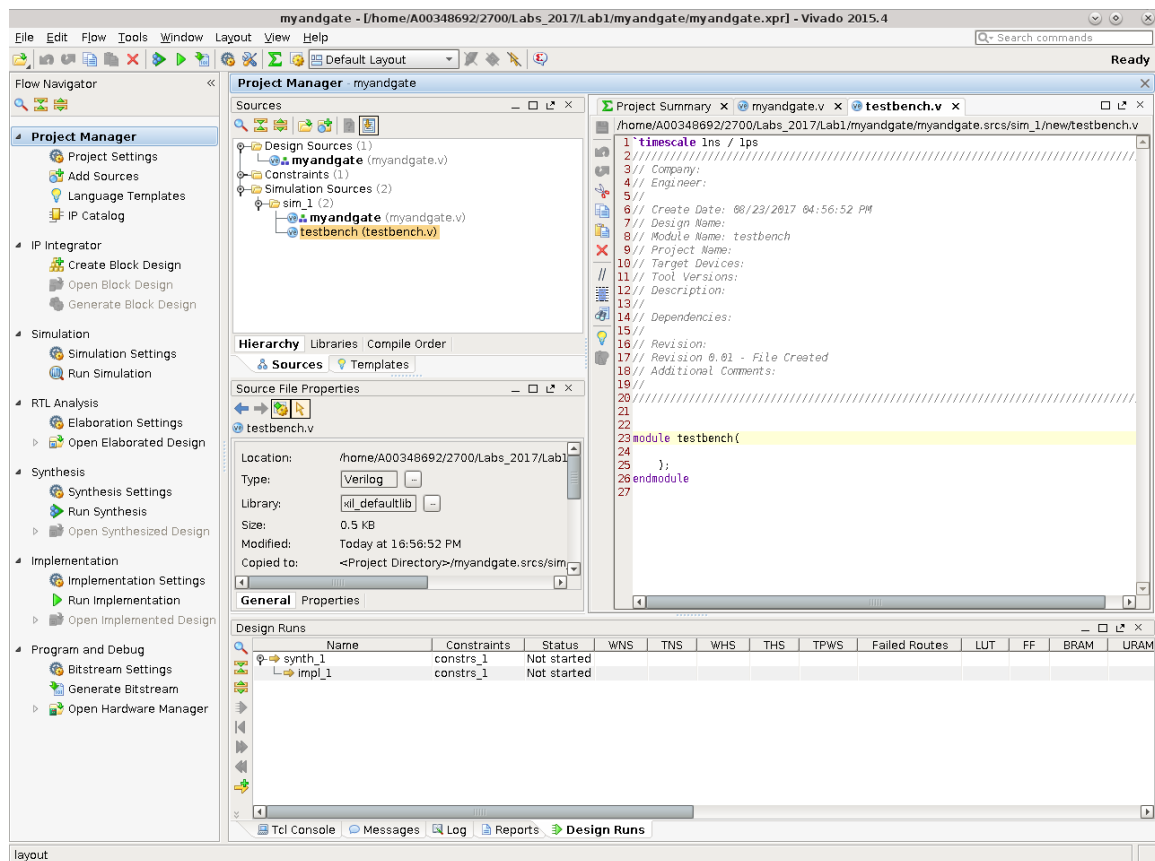
    // ADD THIS LINE:
    and U1(F,A,B);
endmodule
```

6.2 Create a Testbench for the Module

You will now create a testbench to verify your design. To make a testbench, look at the **Flow Navigator** tab on the far right side of the Vivado project window. Under the **Project Manager** tab, click **Add Sources**. This will popup a wizard screen. In the wizard, select **Add or create simulation sources**. Then click **Next**.

A new wizard screen should appear. Click **Create File**, and in the popup window type **testbench.v** and click OK, then click Finish in the wizard screen.

The **Define Module** popup should appear. Since a testbench has no inputs or outputs, just click OK. It will ask if you're sure; say Yes. If everything was done properly, your screen should look like this:



Under the **Simulation Sources** folder, expand the **sim_1** subfolder and you should see your new testbench file. Double-click it to open, and let's enter some code for a basic four-pattern behavioral test:

```
module testbench(    );
    reg  A, B;
    wire F;

    myandgate DUT(F,A,B);

    initial begin
        A = 0; B = 0;
        #10
        A = 1; B = 0;
        #10
        A = 0; B = 1;
        #10
        A = 1; B = 1;
        #10
        $finish;
    end
endmodule
```

Save your work. Remember that A and B in the testbench correspond to A and B in your code.

Now let's examine each line of the testbench. First we see the **timescale** directive. This tells the compiler that our time unit is one nanosecond, and the internal precision should also be one nanosecond. In this course, we will usually set the units and precision to be the same value.

Next, we see the **reg** and **wire** declarations. This tells the compiler that signals A and B are *behavioral* signals that will be defined by the testbench code. By contrast, the F signal is a *structural wire* that will be defined by the DUT submodule.

The next line declares an *instance* of **myandgate**. This instance is named DUT, which stands for "Design Under Test." Whenever we create a specific module of some design, we say that the module is *instantiated*. The syntax for instantiating a module is:

```
[ecestudent@centos7 ~]$> <module_type> <instance_name>(<I/O list>)
```

After instantiating the DUT module, there is an **initial** block that defines the testbench's behavior. Verilog allows lines of behavioral code to be grouped together when they are surrounded by **begin** and **end** statements, as is done here.

With the **initial** block, the first line declares a *blocking assignment* of A and B using the = operator¹. Because this assignment occurs at the start of the **initial** block, it instructs the compiler that this will be the circuit's initial condition at the start of simulation.

Subsequent lines in the **initial** block begin with the **#10** directive. This tells the compiler to *delay* for 10 time units before implementing the next assignment. Since our **timescale** is set to 1ns, this implies a delay of 10ns before each new input combination. Our simulation will be finished after three such delays, with the final state appearing after 30ns. At 40ns the simulation will terminate due to the **\$finish** system task.


Now we will check for syntax errors and simulate the AND gate design. Near the bottom of the Vivado Project window, you should see a set of tabs with names **Tcl Console**, **Messages**, **Log**, **Reports**, and **Design Runs**. Click on the **Messages** tab and see if there are any **Critical Warnings**. A syntax error will appear automatically as a critical warning. If you see any, read the messages and correct any mistakes you may have made in your two files.

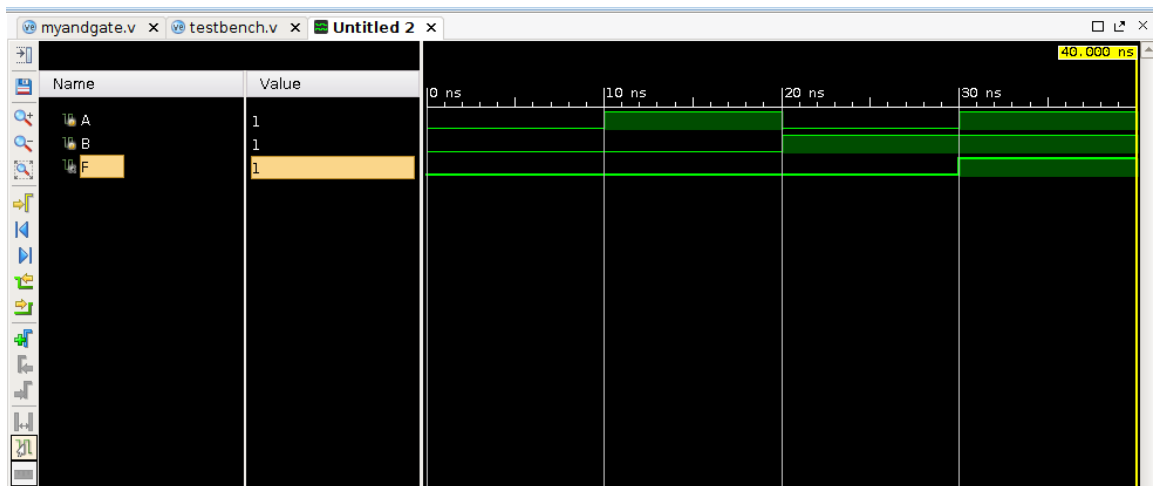
When everything looks good, look for the **Simulation** group at the far left side of the window, and click **Simulation Settings**. A settings window will appear. In the center of the settings window are tabs

¹We will talk more about blocking vs non-blocking assignments in the future.

for **Compilation**, **Elaboration**, **Simulation**, and so on. Click on the **Simulation** tab and notice the first option, **xsim.simulate.runtime***, which is set to 1000ns by default. This tells the simulator to automatically run for 1000ns on startup. But our testbench will finish after just 40ns, so change the value to 40ns and click OK.

Next, under the Simulation heading, click **Run Simulation**. It will produce a drop-down menu; select **Run Behavioral Simulation**. You should see the simulation view appear. Click on the green **Untitled** waveform tab. You may see some flat lines because the waveforms are zoomed out. Click the **zoom-**

to-fit button  to show the actual simulated time window. You should see the waveforms shown below.



Notice the yellow cursor and move it to different times in the simulation. Next to each signal name is a **Value** field that reports the signal's value at the cursor time. **Verify that the output F is correct for all four combinations of A and B.** It should be behavior of an AND gate, zero for all cases except when A=B=1.

7 Implementation on the Basys Board

Now that you have verified that your design is correct, you are ready to load your design onto the board. Return to the Project Manager View by clicking the **Project Manager header** in the Flow Navigator on the left side of the Vivado window.

Now we need to tell the program which of the Basys3's physical resources (i.e. buttons, switches, LEDs and so on) to associate with the signals A, B and F. Since A and B are inputs, it is sensible to associate them with switches. Since F is an output, it is sensible to associate it with one of the board's indicator lights, called Light Emitting Diodes (LEDs). The Basys3 has 16 switches and 16 LEDs; we will use the lowest-numbered among them, namely sw0, sw1 and led0.

In Vivado, resource associations are defined in the **Basys3_Master.xdc** file within the **Constraints** folder in the source browser. Double-click to open the constraints file. It contains a lot of commented-out lines showing examples of constraint definitions. Each constraint has two lines, one to define the associated pin on the FPGA chip, and another to define the logic signal voltage. Find the lines corresponding to sw[0], sw[1] and led[0], copy them to the top of the file, then uncomment them. Look for the `[get_ports sw[0]]` and change the port names to match the signal names in your design. The final settings should look like this:

```
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 {F}]
set_property IOSTANDARD LVCMOS33 [get_ports {F}]
```

Note that the constraint syntax is quite different from Verilog. Comments are entered with a `#` sign, and lines do not require semicolons at the end.

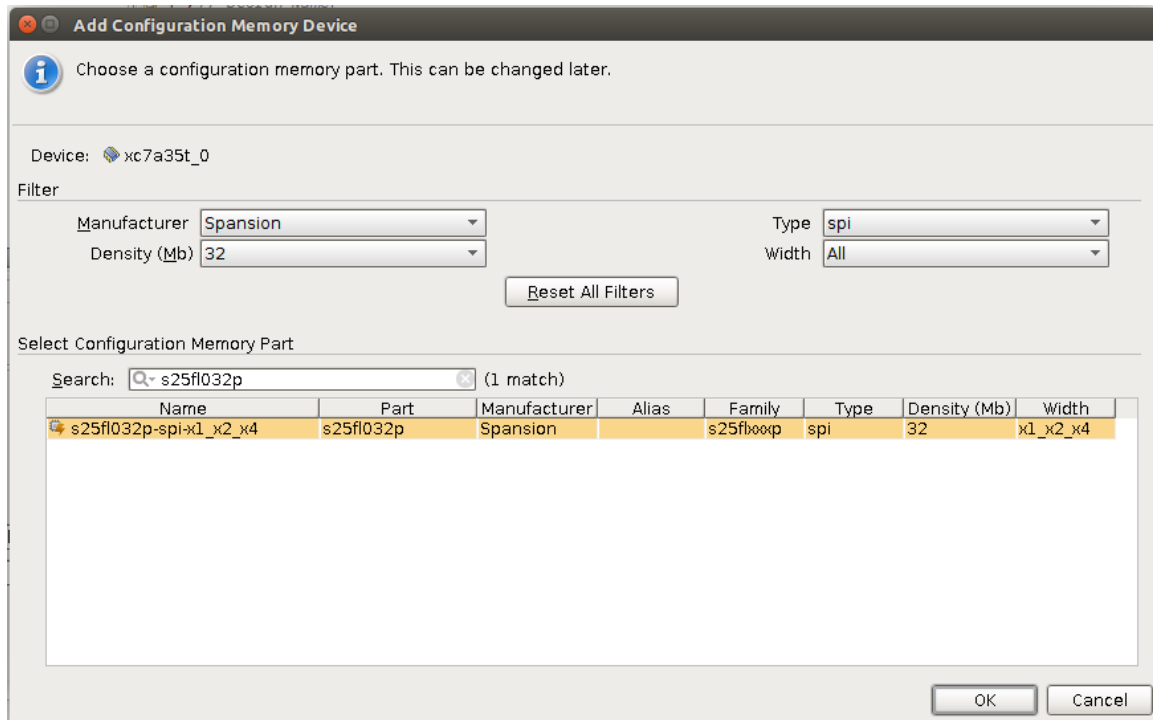
Save your changes to the constraint file. Now, in the Flow Navigator, click **Run Synthesis**. This will map your design to the types of physical logic cells available within the FPGA chip. You should see a roving indicator at the upper right corner of the window, telling you that synthesis is in progress. When its done, a popup window will appear. If successful, select **Run Implementation** and click OK. In this stage, the tool will map the synthesized design onto *specific* resources within the FPGA. Whereas synthesis reduces your design to available resource-types, implementation chooses exactly which resources will be used and how they will be connected together. When it finishes, you will get another popup. Click **Generate Bitstream** to create the final program file that gets sent to the FPGA.

If all goes well, you will get another popup window saying Bitstream successfully completed. Now click **Open Hardware Manager**. Now, **BEFORE you connect your Basys3 board via one of the USB sockets, make sure the jumper JP1 (in the upper right corner of the board) is in the middle position. Also make sure JP2 (in the upper left corner of the board) is in the lower position.** After verifying this, connect your board. In the Hardware Manager view, click **Open Target** and then **Auto Connect**. You should see a Xilinx device appear in hardware list. If it doesn't, make sure the board is properly plugged in, that the power switch is turned ON, the red power indicator light is ON, and the jumpers are in the correct positions. If you checked all this and it still doesn't work, try a different USB cable.

Once you see your hardware in the list, click **Program Device**, select your device (should be the only one listed), and then click OK in the popup window. A progress indicator should rapidly complete the programming. After programming, verify your design by manipulating the two right-most switches along the bottom of the board (note that these are labeled SW0 and SW1). When both switches are ON, the green LED labeled LD0 should light up.

7.1 Loading your Program into Flash Memory

On your Basys3 board, try toggling the power switch OFF and back ON. Now see if you can get LD0 to light up again. It won't, because your design is now dropped from the FPGA. It will be useful to make your design *persistent* by programming it into the Flash memory. To do this, you will need to modify the **Bitstream Settings** as follows: first, click Bitstream Settings in the Flow Navigator. Then, in the settings window, check the box next to **bin_file**, and click OK. Then click **Generate Bitstream** again to create both a bit file (for temporary programming) and a bin file (for persistent Flash programming). The process should complete with no problems. Next, click **Add Configuration Memory Device** under the Hardware Manager heading in the Flow Navigator. A popup window will appear showing a catalogue of memory devices. In the dropdown boxes, select the settings shown in the screenshot below, or just type **s25fl032p** in the Search field and press Enter.



Click OK after selecting the correct memory device. A new popup will appear asking if you want to program the device now. Click OK. Now you get another popup asking for the configuration file. For this, you need to use the file browser to navigate into your myandgate project folder, find the subdirectory named **myandgate.runs**, and within that go into **impl_1**. There you should find **myandgate.bin**. Select this as the configuration file, then click OK to program your device.

It may take a little while...

When programming is done, turn off the power to your Basys3 board (Vivado may flash a warning message but it doesn't matter, just get rid of it). **With the power OFF, move JP1 to the upper position.** Then turn the power back on, and wait for the DONE LED to light up in the upper right corner of the board. Now test your AND gate functions with SW0 and SW1 – your design should be there. **Flash programming will be a useful way to show your designs to TAs, instructors, friends, family, neighbors and random passersby.**

Carry your Flash-programmed board to the TA and demonstrate your design so that it can be checked off.

8 Alternative AND Gate Description

To get a little more out of this lab experience, go back to the `myandgate` project and try implementing it using the other two methods described below. For both cases, run through the steps and verify that you get the same behavior.

Demonstrate to your TA both the simulation results and correct FPGA implementation for the above two alternative designs.

First, using continuous assignments:

```
module myandgate(  
    output F  
    input A,  
    input B,  
);  
    assign F = A & B;  
  
endmodule
```

Second, using a behavioral model:

```
module myandgate(  
    output reg F  
    input A,  
    input B,  
);  
  
    always @(A, B) begin  
        F = A & B;  
    end  
  
endmodule
```

9 TA Checkoff

- (4 points) Complete pre-lab work prior to start of the lab.
- (16 points) Correct simulation of the AND gate.
- (10 points) Correct implementation of the AND gate on the Basys3 board.
- (20 points) Correct simulation and implementation of alternative AND on the Basys3 board.