18-240: Structure and Design of Digital Systems

Lab0: Introduction to SystemVerilog Tools

Objective and Overview

The purpose of Lab 0 is to give you an introduction to the hardware and the software that you will use in the lab portion of 18-240. You will gain familiarity with the SystemVerilog language and simulator. You will download a SystemVerilog specified design to an FPGA from your lab PC.

Schedule and Scoring

The lab is worth 45 points, 5 points for each of four tasks: User Account, Simulation, Hardware, Gradescope. 5 points are for code style that conforms to the course coding standards. The lab report is worth 20 points.

A lab report must be turned in via Gradescope. SystemVerilog files must also be turned in, via **`handin240`**, to Gradescope.

All four tasks must be checked off by 8:00pm on Friday, 8 September 2023. Your lab report is due at the same time.

A Note about Collaboration

Lab 0 is to be accomplished individually. All work must be your own. You may ask other students for general assistance, but you may not copy their work.

Task #1: User Account (Prelab)

You should have a computer account for logging into the ECE Unix/AFS systems. The ECE account is independent from your Andrew account, though it uses your Andrew ID and password to log in. If you are not an ECE student, navigate to https://changepass.its.cit.cmu.edu/ to set up your ECE account. This action may take up to 24 hours, so you must complete this action 24 hours prior to showing up for lab.

The ECE Unix account is good for the Linux machines in lab as well as the 32+ remote machines run by ECE. These remote machines are called ece000, ece001, ... ece031.ece.cmu.edu (or ece0XX.campus.ece.cmu.local if you are on a campus network). You may make use of these machines in the future to remotely run simulators and other tools for homework and for labs.

The 18-240 Wiki has lots of advice on how to set up your computer to access the ECE servers, including SSH, X11 Forwarding and SSH Shortcuts. Please read it: `https://github.com/CMU-18240/240-How-to/wiki`

Please note that the AFS account you received gives you some disk space where you can store files for your course work. The top level (**`/afs/andrew/usr/your_Andrew_ID`**) is readable by anyone! This is useful since you might want to make certain information available to others working on a project with you. There is a **`Private`** directory there too. No one else has access to that directory. Protect your work against copying by keeping your course work in the **`Private`** directory.

The computers in the lab are Linux machines, shared by all of the lab sections of this course. You log into the lab computer using your Andrew ID and password. You should keep your working files and directories somewhere convenient and private. We suggest you create a directory, in your AFS space, to contain all of your 18-240 related files in one place. We suggest that, to the degree possible, you work on files in your AFS space. The lab computer mounts your AFS storage space (secure and reliable) upon login.

For credit: Show the TA that you can log into the 18-240 lab machine.

Lab report: Answer the following questions in your lab report:

1. (True or False, explain why) I need to use **`ssh`** when I am working from my laptop.
2. (True or False, explain why) I need to use **`ssh`** when I am in HH 1305 and working on one of the computers in lab.
3. What program[1] is used to provide remote access to the VCS simulator, and thus can be used prior to coming to the lab?

Task #2: SystemVerilog Hardware Description and Simulation

SystemVerilog is a hardware description language used to model hardware logic designs. The simulator is a software program that mimics what the hardware you described (in SystemVerilog) would do. It drives the inputs of your logic model and tells you what the outputs are. It allows you to test your design before you commit to creating an actual hardware implementation.

In this part of the lab, you will simulate the SystemVerilog description of a simple logic circuit. More background on the SystemVerilog language will be covered in Lecture #3. The textbook (*Logic Design and Verification Using SystemVerilog*, by Don Thomas) provides an overview of the SystemVerilog language. Specifically, Chapter 1 is relevant to this lab.

You do not need to be logged onto a HH 1305 computer to complete this section of the lab. If you would like to complete this section before coming to lab, you can **`ssh`** into ECE machines to do your work (18240 Lab Guide, Section 3. Also, 240 Wiki).

[1] or, maybe what protocol? Depends on how your setup is working

The VCS Tutorial

On Canvas, find the tutorial for the simulator: **SimulationWithVCS.pdf**. You can find a link to this file on the front page, under the label "Simulating SystemVerilog with VCS." You may find the tutorial helpful with this second task.

The Multiplexer

Type[2] in the following example into a file and call it **lab0.sv**. This SystemVerilog example describes a module that chooses which of two input bits should be passed through to a single output bit. The choice is made with a third input bit. The multiplexer module is:

```
1  `default_nettype none  // Required in every sv file
2
3  module multiplexer
4    (output logic f,
5     input  logic a, b, sel);
6
7    logic f1, f2, n_sel;
8
9    and #2 g1(f1, a, n_sel),
10          g2(f2, b, sel);
11   or  #2 g3(f, f1, f2);
12   not    g4(n_sel, sel);
13
14 endmodule: multiplexer
```

Let's take a look at what you typed. Line 3 gives the name of the module. Lines 4 and 5 list the module's three input ports named **a**, **b** and **sel** and an output port named **f**. These ports have a type of "logic." Line 7 defines some internal signals; you can think of these as named connection wires. Line 9 instantiates an **and** gate (named **g1**), with **a** and **n_sel** driving its input ports, and **f1** driven by its output port. (Basic gates like **and** and **or** are built into the SystemVerilog language. On these built-in primitive logic gates, the output is always in the first position and the rest are inputs.) This **and** gate instantiation has a combinational propagation delay of two time units. That is, two time units after an input changes, the output will follow according to the **and** boolean function. Three more combinational gates are instantiated on Lines 10 - 12. The **not** gate on Line 12 has no delay specified, and thus the **n_sel** output will change in the same time unit that the input **sel** changed. Line 14 is the **endmodule** statement which indicates the end of a module description. Having defined the multiplexer module, we can instantiate it (as we have with the built-in **and**, **or** and **not**) in another module hierarchically.

[2] Be careful with cut-and-paste from the lab PDF file. Such a process often replaces some characters with ones that are not legal SystemVerilog syntax. For instance, the back-tick or a "smart quote."

The Testbench

After we have described this module, we should see if it works correctly. To do this in simulation, we need to create a "testbench" – the **muxTester** module. The idea is that we're going to connect the input and output ports of the **muxTester** module and the **multiplexer** module. The **muxTester** module will exercise the behaviors of the **multiplexer** module by driving the **multiplexer** module's inputs to different values at different times.

Type the following code into the same file.

```
1  module muxTester
2    (output logic a, b, sel,
3     input  logic muxOut);
4
5    initial begin
6      $monitor($time,,
7        "a = %b, b = %b, sel = %b, muxOut = %b",
8        a, b, sel, muxOut);
9          a = 0;
10         b = 0;
11         sel = 0;
12     #10 b = 1;
13     #10 a = 1;
14     #10 b = 0;
15     #10 sel = 1;
16     #10 b = 1;
17     #10 a = 0;
18     #10 b = 0;
19     #10 $finish;
20   end
21
22 endmodule: muxTester
```

The **initial** keyword introduces a "behavioral" initialization block that starts executing once when the simulation starts. The body of this behavioral initial block (Lines 6 19) are enclosed by the **begin...end** keywords and can be read sequentially much like you would read a C program (but, we will come to see many differences).

This initial block starts with a **$monitor** statement (the unwieldy line 6). The **$monitor** statement causes information to be printed on to the display console whenever one of the monitored values changes (i.e., when either **a**, **b**, **sel**, or **muxOut** changes, in this case). The formatting string is similar to **printf** in C — **%b** indicates a binary format. The first parameter **$time** causes the monitor to print each new output with a timestamp. The double comma after **$time** is deliberate.

The next specified event is to assign **a**, **b** and **sel** to be **0** (Lines 9 - 11). **#10** tells the simulator to wait for 10 time units before continuing execution. Thus, in **10** more time units, **b** is set to **1** (Line 12). After another 10 time units, **a** is set to 1 (Line 13). Over the course of this simulation, the three inputs will be assigned to all eight (2^3) possible combinations. Finally, **$finish** causes the simulator to exit (Line 19).

The outputs of the test module will need to be connected to the inputs of the "Device Under Test" module (thus, often called DUT). Create a system module to connect the **multiplexer** and**muxTester** modules. Type this module into the same file:

```
1  module system();
2    logic wire_a, wire_b, select, mux_out;
3
4    multiplexer DUT (.a(wire_a),
5                     .b(wire_b),
6                     .f(mux_out),
7                     .sel(select));
8
9    muxTester    mt (.a(wire_a),
10                    .b(wire_b),
11                     .muxOut(mux_out),
12                     .sel(select));
13
14 endmodule: system
```

As a top-level module, the **system** module has no input ports or output ports. The two modules we have created are instantiated in the **system** module and given instance names: **multiplexer** is named **DUT**, and **muxTester** is named **mt**. Using logic variables, we have connected the input and output ports of the **DUT** module and the **mt** module. Note that the logic signals names have meaning only in the particular module. An output of **mt** is **a**, which in **system** is connected to a logic signal (essentially a wire) called **wire_a**, which is then connected to the **a** input of the **DUT**. Note the use of named port specifiers, which is good style.

Do not think of the module instantiations (Line 4 and 8) as first executing the **multiplexer**, and then executing the **muxTester** — these are not function calls. Rather, we are describing a the structural interconnection of hardware. A module *instantiation* specifies that an instance of the module should be created and its input and output ports associated with the logic variables accordingly. These logic variables act like wires to carry signals when used in this fashion. Thus, as you might expect, exchanging the ordering of the modules does not change the structure being described, much like moving some gates around on a circuit doesn't change the operation of that circuit.

After adding the **muxTester** and **system** modules into the file **lab0.sv**, simulate the system, using what you know about VCS from the tutorial. The simulator should display all the monitored signals when they are changed.

Reason your way through the simulated sequence of behaviors for these modules. The **muxTester** module is supposed to set **a**, **b** and **sel** to certain values and then wait **10** time units before changing **a**, **b** and **sel** to another combination. Under these input stimulus over time, how should the **and** and **or** gates in the **multiplexer** module behave? What effect do the time delays of the **and** and **or** gates have? Play around with their values. Try making them larger than **10**.

For credit: Show your TA that you can simulate this example. Be prepared to answer questions from the TA about how the modules interact and how VCS works.

Lab report: Answer the following questions in your lab report:

4. What do you need at the top of every **.sv** file and why?
5. What is the difference between module instantiations and function calls?
6. What does the **logic** keyword mean and what is it used for?
7. What happens if you run multiple **initial** blocks in one file?
8. What is a top module? Does it have inputs and outputs?
9. What effect does swapping lines 4-7 with lines 9-12 of the **system** module have on the simulation output? Explain.
10. Why is the waveform viewer useful? When would you use it?
11. Where are the input signals to the **multiplexer** module generated?
12. Where are the input signals to the **muxTester** module generated?

Task #3: Download the design to hardware

You must be logged into one of the HH 1305 computers and have access to the DE2-115 for this task.

We are now ready to "realize" the SystemVerilog specified multiplexer design on an FPGA on the Altera DE2-115 board. In place of the simulation testbench modules, we need to introduce an interface module that will take the input and output ports of your multiplexer and connect them to the physical switches and LEDs on the FPGA board.

chipInterface

Create a new file named **chipInterface.sv** with the following "wrapper" module in it. We call **chipInterface** a wrapper because it wraps around the instantiated multiplexer module and creates an interface to the pins of the FPGA chip.

```
1  `default_nettype none // Required in every sv file
2  module chipInterface
3    (input  logic SW  [17:0],
4     output logic LEDR[17:0]);
5
6     // TODO:
7     // - Instantiate your multiplexer module
8     //   inside this chipInterface
9     // - Hook up SW[0], SW[1] and SW[7] to a, b and
10    //   sel inputs on the mux
11    // - Hook up LEDR[15] to the output of your multiplexer
12    // - Finish the module (you'll need something else
13    //   to indicate your module is complete)
```

Once you have both the **chipInterface.sv** file and **lab0.sv** file, you are ready to synthesize your design and download it to the DE2-115 board using the steps described in the short Altera usage tutorial available on Canvas (**QuartusII_Tutorial.pdf** on the front page). Before you get started, however, read the following three paragraphs.

On line 3, **SW[15:0]** is a vector of all the wires coming into the FPGA from the switches. If you use the Pin Planner (hint: don't do this), you can choose individual pins for wires number 0, 1 and 15 to connect the **multiplexer** inputs to. Or, you can simply include the **Lab0.qsf** pin assignment file (found posted on Canvas) into your project. In this case, you have to specify all 16 switches (by using the **SW[15:0]** notation in your port definition), otherwise Quartus will complain.

Same for the LEDs on line 4.

Make sure you understand this second method. It works more reliably and is lots easier than the first. You'll be doing it a bunch for other labs this semester. However, you should take care not to forget

what Quartus is doing for you whenever you include inputs and outputs on your **chipInterface** module that you want hooked up to particular pins of the FPGA.

Quartus

Now that you've read those paragraphs, create a new Quartus project. You should read through (and perhaps work through) the tutorial available on Canvas (**Quartus_Tutorial.pdf**). For future reference, your TAs have written a handy Lab Guide (on Canvas as **LabGuide.pdf**) with a quick reference that is intended to be a reminder so you don't have to read through the entire Tutorial again next week.

- Create a new Quartus project.
  - Name your "Top Level Entity" as **chipInterface**. This is the module that is the top of your synthesis hierarchy.
  - Add both the **chipInterface.sv** and **lab0.sv** files to your project.
- Import Pin Assignments
  - The FPGA is part of a larger printed circuit board (PCB) where pins touch specific resources on the board, so correctly assigning FPGA pins that connect to your internal logic ensures you are driving the correct outputs and reading the correct inputs.
  - Go to **Assignments** ⇒ **Pin Planner** to pull up the Pin Planner. In the table at the bottom, the **Node Name** column displays your input and output variables of your top level module. Since you have not set any pin assignments, the **Location** column is empty and the internal logic of your design does not know where to read and drive the inputs and outputs.
  - Import pin assignments (i.e. the **Lab0.qsf** file). Use the process described in the Quartus Tutorial.
  - Reopen the **Pin Planner**. You know that you have correctly imported the pin assignments if there are now red dots on the **Top View** and the **Location** column is filled.
- Synthesize your design
- Use the programmer to download the resulting bitstream onto the FPGA

For credit: Demonstrate, to the TA, the multiplexer design on the DE2-115 FPGA board; which should operate properly. Make sure you actually understand how the circuit should work prior to the demo. Be prepared to answer questions from the TA about the various steps involved in synthesizing your code.

Lab report: Answer the following questions in your lab report:

13. What is the difference between synthesis and simulation?
14. What significance does importing the pin assignments have? (Hint: Take a look inside `Lab0.qsf`)
15. How do you know that you have successfully imported pin assignments? What happens if you do not successfully import pin assignments. Will you get an error message? Will you notice it if you do? Why or why not?
16. What happens if your top module does not match the **`Top Level Entity`** of your Quartus project?
17. What happens if you instantiate a testbench in your Quartus project and try to compile? Why?

Task #4: Upload to Gradescope

You should now have a **lab0.sv** and **chipInterface.sv** files, both of which should be submitted to Gradescope for grading. This task is intended to illustrate the handin process you will use for labs and homeworks.

Follow the directions on the 18-240 Wiki to set up your handin process (`https://github.com/CMU-18240/240-How-to/wiki/Overview-of-handin240`).

While logged into an ECE server and with your current directory containing the code files, execute the handin script. It will create a file named **lab0_code.pdf**, which has a prettily printed version of your files.

Carefully examine any output from running the command, which might tell you that your files couldn't be compiled by the simulator. Such a situation would result in a zero grade, so you'll want to fix any such problems before the final submission.

Also, carefully examine the code output in the **lab0_code.pdf** file. It also flags a few minor errors that you might have made (using tabs, long lines), which you probably would want to fix prior to submission.

Submit the **lab0_code.pdf** file to the **Lab0 Code** assignment on Gradescope. Note that there is also a **Lab0** assignment. That assignment is intended for your lab report. Don't get them mixed up!

One of the major reasons for this task is for you to get familiar with the handin procedures that will be used for labs and homework. Therefore, task 4 must be accomplished while you are in lab so you can ask a TA for assistance.

Your lab report should be submitted as a PDF to the **Lab0** assignment on Gradescope. You may finish writing it and submit it after you have left the lab spaces, though it would be better to get it done while you are near the equipment and can easily get TA help.

Lab report: Answer the following questions in your lab report:

18. What is the filename of the file where you can read the SystemVerilog coding standards for 18-240?
19. What initialization script must be executed at some time before you try to run **handin240**?
20. Why must the **handin240** script be executed on an ECE server? What directory must you be in when you execute it?

For credit: Demonstrate to the TA that you have uploaded the code file to Gradescope. Prepare to answer questions about the upload process from your TA.