Lab 0: Revisiting Digital Logic and SystemVerilog Simulation

Assigned: Monday 1/18; Due **Monday 1/30** (midnight)

Instructor: James E. Stine, Jr.

1. Introduction

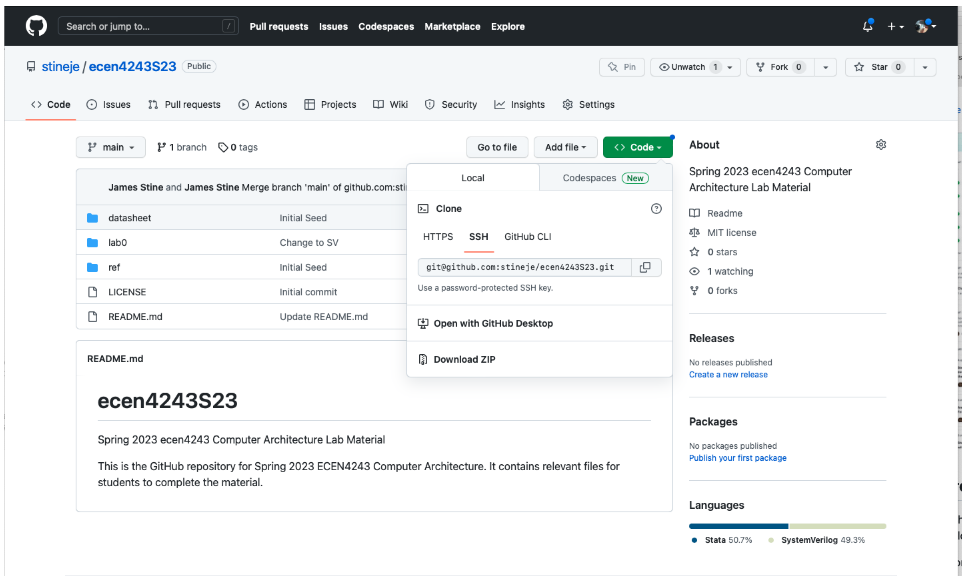
In this lab warmup, you will take a quick tour using the SystemVerilog tools that we will use in this class, and what it takes to run them by hand. You will then construct a simple register file in SystemVerilog, which should be a review of material from ECEN 3233 or DLD. By the end of this warmup, you should be familiar with MGC ModelSim for simulation and for viewing the output of ModelSim. If you have not touched SystemVerilog in a while, this lab should allow you to recall the basics. In later labs (beginning with Lab 2), you will be developing a more significant codebase in SystemVerilog and simulating it with these tools.

2. GitHub

As indicated in class, Linux Torvalds invented the idea of using `git` in 2005. It has now become an industry standard for all companies in terms of software and hardward development. You will be utilizing `git` for the entire class and I highly encourage you to learn how to use it well. Since it is so integral to the success of projects within corporations, it is something you should learn well and have in your belt of tools that you need after you graduate or utilized during your internships.

`GitHub` is an Internet hosting service utilized for handling `git`. You will use GitHub for the repositories that we created for this class. GitHub is now owned by Microsoft as of 2012.

Our current repository is held at `https://github.com/stineje/ecen4243S23`. Figure 1 shows a screenshot of our current ecen4243 repository. Using GitHub is quite easy to use. A great tutorial for beginners is found here:`https://www.freecodecamp.org/news/git-and-github-for-beginners/`. However, you can just click the green Code button and download a Zip file of files easily. For those who want more experience, I encourage you to utilize the command line to deploy version control with GitHub.



James Stine and James Stine Merge branch 'main' of github.com:sti

datasheet	Initial Seed
lab0	Change to SV
ref	Initial Seed
LICENSE	Initial commit
README.md	Update README.md

git@github.com:stineje/ecen4243S23.git

Use a password-protected SSH key.

Spring 2023 ecen4243 Computer Architecture Lab Material

README.md

ecen4243S23

Spring 2023 ecen4243 Computer Architecture Lab Material

This is the GitHub repository for Spring 2023 ECEN4243 Computer Architecture. It contains relevant files for students to complete the material.

Figure 1: Spring 2023 ecen4243 GitHub Site.

3. Part I : Simple Finite State Machine

For the first part of this lab, We will walk through the SystemVerilog workflow for this class with a simple finite state machine. The state transition diagram and SystemVerilog description of this Mealy finite state machine with 1-bit input and 1-bit output is given below. To help get you started, the SystemVerilog and testbench are given to you, however, it is up to you to get this to simulate.

3.1 Simulation

Simulation is key to making sure your hardware for any architecture or digital system works. It is often the difference between something that works and something that is a pure speculation. Therefore, it is vital that for lab that you understand well how to simulate and get results from ModelSim.

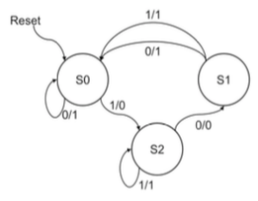
Siemens ModelSim/Questa is a popular Hardware Descriptive Language (HDL) compiler and simulator that is widely used in the industry. Although there are similar tools form other vendors (e.g., Synopsys VCS or Cadence Design Systems NCsim), the ideas are rather similar between Electronic Design Automation (EDA) tools. Regardless of the choice of simulator, the use of testbenches and batch files to invoke simulation are the staple of digital designers and architects.

Testbenches are essential to HDL designs and they are not unique to simulators. They are part of the Verilog standard and are typically written in a behavioral manner to make sure your simulation essentially works. Although many testbenches are utilized, using a testbench that verifies what the **true** result should be is essential. Therefore, it is encouraged that you utilize a self-checking style in your testbench. Although testbenches are usually written by the architecture, a sample testbench is provided to you. You should modify this testbench to make sure your design works as the state transition diagram works as indicated in Figure 2.

The next part of the simulation environment is called a `DO` file and is basically a batch file for ModelSim that allows the simulation to run regardless of a users set up. A sample `DO` file is given to you, but you should modify to make sure it runs your finite state machine design and its appropriate testbench. To run ModelSim with a `DO` file, type the following command at a command prompt.

```
vsim -do FSM.do
```

It is encouraged to consult your TA or the Internet to learn how to get to the terminal in Microsoft Windows, so you can run your DO file correctly.



```
module FSM (
    output reg Out,
    input reset_b, clock, In);

parameter [1:0] S0 = 2’d0, S1 = 2’d1, S2 = 2’d2;
reg [1:0] state, nextState;

always @ * begin
       case (state)
              S0:    begin  nextState = In? S2: S0;
                            Out = In?0:1; end
              S1:    begin  nextState = S0;
                            Out = 1; end
              S2:    begin  nextState = In? S2: S1;
                            Out = In?1:0; end
              default: begin nextState = 2’bx;
                             Out = 1’bx; end
       endcase
end

always @(posedge clock or negedge reset_b) begin
       if (~reset_b)
               state <= S0;
       else state <= nextState;
end

endmodule
```

Figure 2: Sample Mealy Finite State Machine.

3.2 Implementation

Go ahead and simulate your FSM. Consult Chapter 4 of the DDCA textbook for a refresher on digital logic design and implementation [1]. You should run your simulation and verify that the design indeed follows the FSM implementation shown in Figure 2.

4. Part II: Register File

In the second half of this practice lab, you will construct a register file (RF) in SystemVerilog. Although this lab is sort of practice of what you should already know, the module that you write in this section will be useful to you in Lab 2 if you write it properly.

4.1 Requirements

The register file is a unit in the processor which supplies the functional units (for example, the ALU) with operand values and stores the results of computation for subsequent use. Modern instruction sets typically supply the programmer with 8-32 architecturally-visible registers for integer computation. In this lab, you will implement a register file that is suitable for executing one ARM instruction per cycle. To support one instruction per cycle, the register file must allow two concurrent reads and one write per cycle because a RISC-V instruction can require up to two input operands and can produce one result value (e.g., from an Arithmetic Logic Unit). This is sometimes called dual-porting – in other words, it can read two values from the register file at the same time through two separate ports.

Building register files in SystemVerilog involves utilizing two-dimensional variables. This can be visualized as follows:

```
logic [63:0] A[31:0]
```

This two-dimensional example utilizes registers that are 64-bits long and the second Backus-Naur Format (BNF) indicates the number of registers. In this case, there are 32 of them. Therefore, this statement declares 32 64-bit registers. The key to using these statements is to remember the BNF after the logic declaration indicates the size in bits and the second BNF indicates the number of values.

Your register file should contain 32 registers, each of which holds 32 bits. It should have have the following input and output ports:

- Inputs: Two 5-bit source register numbers (one for each read port), one 5-bit destination register number (for the write port), one 32-bit wide data port for writes, one write enable signal, and clock.
- Outputs: Two 32-bit register values, one for each of the read ports.

The register file should behave as follows:

- Writes should take effect synchronously on the rising edge of the clock and only when write enable is also asserted (active high).
- The register file read port should output the value of the selected register combinatorially.
- The output of the register file read port should change after a rising clock edge if the selected register was modified by a write on the clock edge.
- Reading register zero ($0) should always return (combinatorially) the value zero.
- Writing register zero has no effect.

4.2 Implementation

Go ahead and write a SystemVerilog module RF (see Chapter 7 in [1] for help and perhaps a similar design) with the specification given above, and build a testbench for your register file. In order to help you get started, your repository should have an empty register file that is missing some pieces. However, it should have the ports to help you figure out what is an input or output. Although we have not covered register files in general, the idea should be similar to the idea of a register which you all should be familiar with. More information on the register file can be found within the Appendix of your textbook in Section A.8. Please note our textbook [2] does not use SystemVerilog; however, the differences are minor. It is advisable to consult the DDCA textbook [1] for SystemVerilog usage.

Ensure that you test all reasonable cases (e.g., read a register while it is being updated; read the same register with both read ports at the same time; reset the register file and ensure that all registers read zero). Use the waveform viewer in ModelSim as in the first part of this lab in order to examine the behavior of your register file. You should use modify the FSM testbench and `DO` file to help you verify your design properly.

5. Handin

You should electronically hand in your code (all files that you want us to see) into Canvas through the submission site. Please contact the TA or me for more help. Your code should be readable and well-documented. In addition, please turn in additional test cases or any other added item that you used. Please also remember to document everything in your Lab Report using the information found in the Grading Rubric.

References

[1] S. Harris and D. Harris, *Digital Design and Computer Architecture, RISC-V Edition.* Elsevier Science, 2021.

[2] D. Patterson and J. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface.* ISSN, Elsevier Science, 2020.