Lab 0: Revisiting Digital Logic and SystemVerilog Simulation

Assigned: Monday 1/12; Due **Monday 1/26** (midnight)

Instructor: Shahriar Shahabuddin

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 2233 or DLD. 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. 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 (FSM). The state transition diagram and SystemVerilog description of this FSM 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.

This FSM is the infamous string detector we talk about extensively in ecen 2233 digital logic design and discussed in Chapter 3 of the Harris and Harris text [1]. In this example, Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last two bits that it has crawled over are `01`. This FSM computes when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles. You should test the design with an input sequence of `01_0011_0111`.

You should simulate your design and validate the design so the timing matches what is in Figure 2. As discussed in ecen 2233, it is highly advisable to track the current and next state on the wave window to make sure it is adhering to the behavior in Figure 1.

2.1 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 Questa.

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.

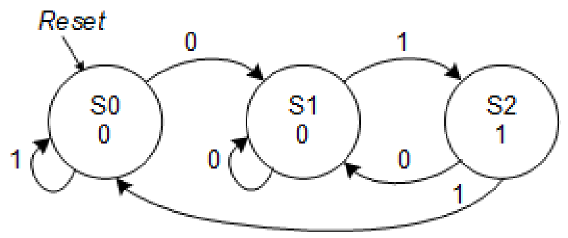

Figure 1: Sample Finite State Machine.

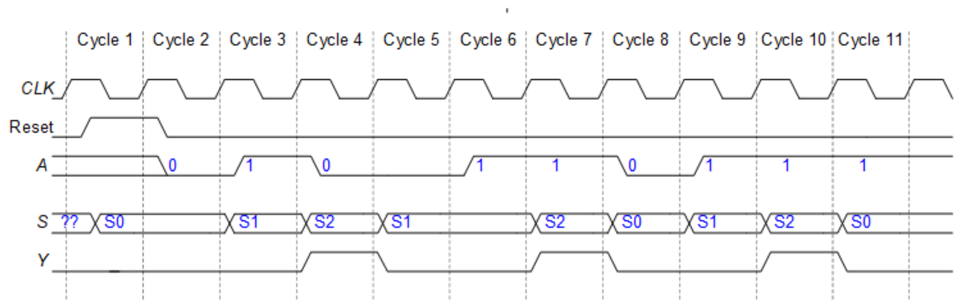

Figure 2: Timing diagram for the Snail Finite State Machine.

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 1.

The next part of the simulation environment is called a `DO` file and is basically a batch file for Questa 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 Questa with a `DO` file, type the following command at a command prompt.

```
vsim -do fsm.do
```

2.2 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 1 and Figure 2.

3. 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.

3.1 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 [31:0] A[31:0]
```

This two-dimensional example utilizes registers that are 32-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 32-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.

3.2 Implementation

Go ahead and write a SystemVerilog module RF 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. 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 Questa 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.

References

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