Lab 0: Revisiting Digital Logic and SystemVerilog Simulation

Collin Bovenschen - A20323474
Koby Goree - A20311573

1. Introduction

This lab serves as a reintroduction to the fundamentals covered in Digital Logic Design, as we are presented with the task of constructing a simple register file in SystemVerilog. Our objectives are to reacquaint ourselves with ModelSim for simulation and fortify our understanding of SystemVerilog basics. This lab is the groundwork for which we will utilize in future labs.

For Part 1 of this lab, we are taken through a guided tour of SystemVerilog workflow, accompanied by a finite state machine. Provided with a DO file and sample testbench, we are able to dive into the world of behavioral simulation and batch file execution. For Part 2, we are required to go deeper into SystemVerilog by constructing a register file. In the following sections, we delve more into detail as we walk through SystemVerilog workflow, solidify our understanding of simulation through ModelSim, and describe the process of constructing a register file.

2. Baseline Design

The Finite State Machine used in this experiment was provided in full through the laboratory files. The Mealy design means that both a boolean input and output must be accounted for, as well as the three states used.

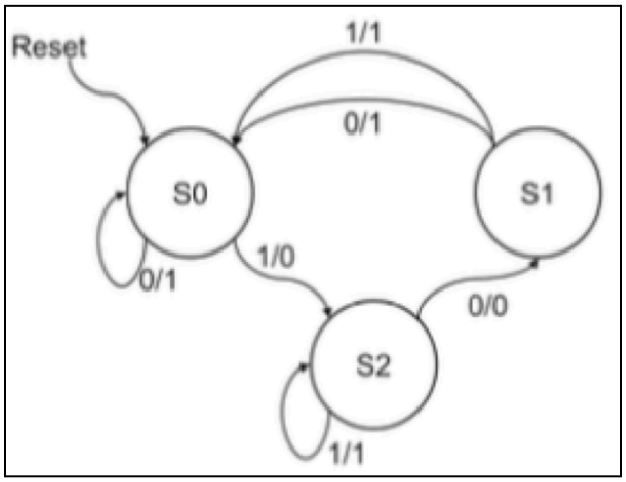


Figure 1: FSM Diagram

The register design was programmed by our team using the parameters outlined in the lab document. The register array consists of individual registers that need certain functions and parameters to be accessed in a meaningful way. To write to a specific register, the address of that specific register must be acquired. To read a register, only the address of the register is needed, as the value there will be supplied through its respective output. The variables needed to perform this system were provided, and the setup was performed by our team.

3. Detailed Design

The register array contains 32 32-bit registers. The address length fits within a 5-bit string, so address reads/writes occur with this length. In this register, writes are conducted synchronously by first determining if the writing is currently valid (write enable is high), and then copying the selected write data given forth by the user into a specific address, also written by the user. It is also required that register 0 cannot be written to, and must always retain a value of 0. This is done by setting the value of the register to 0 on the positive edge of every clock cycle. This also prevents all writes to the register. Reading can be done through 2 separate registers, and these can occur combinatorially.

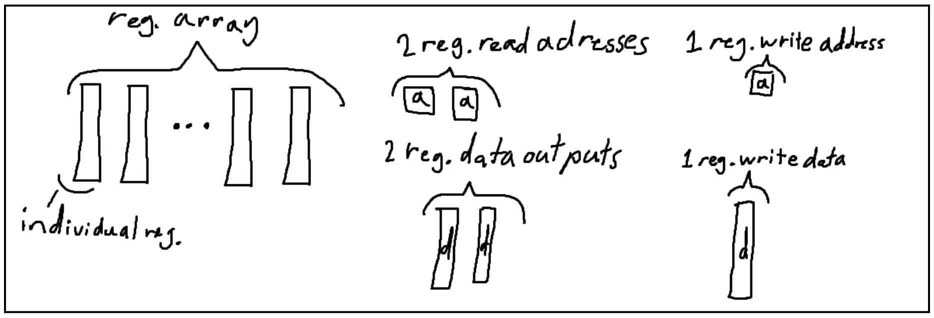
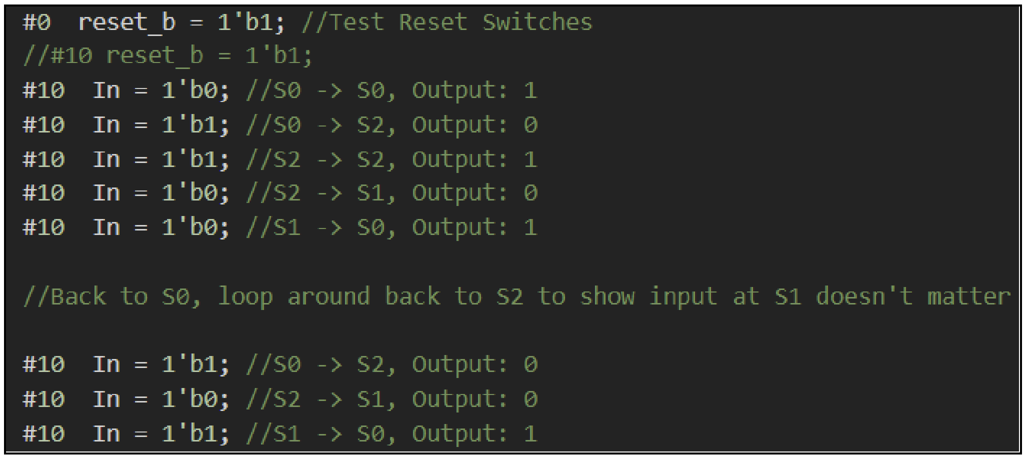


Figure 2: Register Outline

4. Testing Strategy

An incomplete test bench was provided, and this served as a baseline for our testing strategies. To properly ensure the machine was working as intended, each possible state was tested using predetermined vectors. The input values are set in such a way that the system will advance through the machine once to get one of the outputs of S1 and then again to obtain the other.



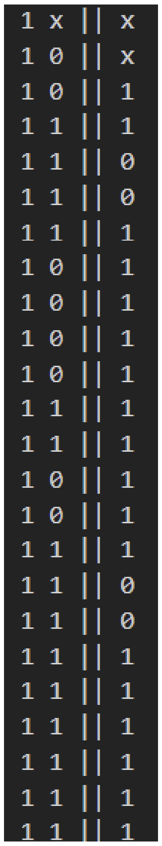
```
#0  reset_b = 1'b1; //Test Reset Switches
//#10 reset_b = 1'b1;
#10  In = 1'b0; //S0 -> S0, Output: 1
#10  In = 1'b1; //S0 -> S2, Output: 0
#10  In = 1'b1; //S2 -> S2, Output: 1
#10  In = 1'b0; //S2 -> S1, Output: 0
#10  In = 1'b0; //S1 -> S0, Output: 1

//Back to S0, loop around back to S2 to show input at S1 doesn't matter

#10  In = 1'b1; //S0 -> S2, Output: 0
#10  In = 1'b0; //S2 -> S1, Output: 0
#10  In = 1'b1; //S1 -> S0, Output: 1
```

Figure 3: FSM testbench test cases

To monitor if the outputs behave as intended, an output file displays the various properties of the system, including the mealy output, on the positive edge of every clock cycle. Going through each input/output here verifies the exportation is functioning as intended. In the output file shown below (Figure 4), The leftmost column indicates the active-low reset switch, while the other two sequentially show the input and output boolean. These outputs line up exactly as predicted in the commented lines.



```
1 x || x
1 0 || x
1 0 || 1
1 1 || 1
1 1 || 0
1 1 || 0
1 1 || 1
1 0 || 1
1 0 || 1
1 0 || 1
1 0 || 1
1 1 || 1
1 1 || 1
1 0 || 1
1 0 || 1
1 1 || 1
1 1 || 0
1 1 || 0
1 1 || 1
1 1 || 1
1 1 || 1
1 1 || 1
1 1 || 1
1 1 || 1
```

Figure 4: FSM Output file

To further prove the state machine is functioning as intended, the states can be viewed on Modelsim's waveform generator. These also appear as intended.

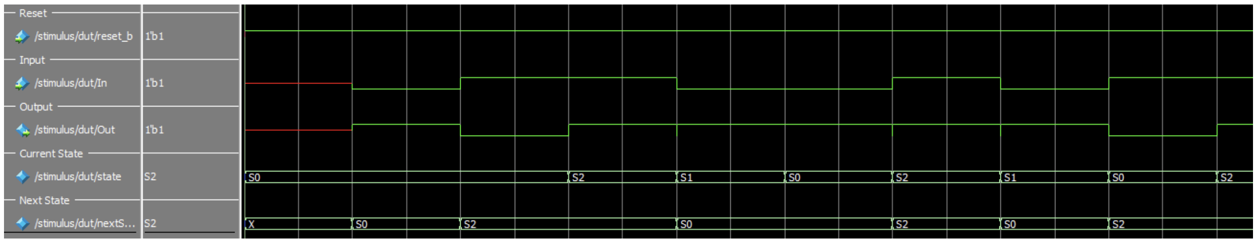


Figure 5: FSM Modelsim waveforms

For the registers, test vectors were used to determine the functionality of the machine. These were also self-checking, so any detected errors would be represented in the command log.

5. Evaluation

This lab offered a well-structured warm-up that allowed us to revitalize and solidify our SystemVerilog tools and concepts. The finite state machine implementation in part 1 provided a good foundation of theoretical understanding and practical application. part 2, involving the construction of a register file, highlighted the importance of thorough testing and validation when performing in a simulation environment. Overall, this lab set a good idea of the immense learning to come in Computer Architecture.