OSU ECEN 4243 Computer Architecture, Spring 2024

Lab 0: Revisiting Digital Logic and SystemVerilog Simulation

Group: Brendan Bovenschen A20323473 and Landon Fox A20319340

January 26th, 2024

Section 1: Introduction

This lab is a reintroduction to SystemVerilog, Modelsim, and the other tools we will use for future Computer Architecture labs. The first part of this lab focuses on the testing aspect of these labs providing us with an already completed FSM in SystemVerilog and a completed testbench and tasking us with simply simulating the program in ModelSim. This task provides us with a simple way to make sure that we can simulate our future programs with ModelSim since no edits need to be made to the files given to us. The second part of this lab tasks us with writing a register file and its testbench in SystemVerilog to remind us of how to create our own codes and testbenches. Our design has a toggleable write mode, one write port, two read ports, and 32 32-bit registers.

Section 2: Baseline Design

The Mealy FSM of three states (as seen in Figure 1) only contains a 1-bit input, an output of the same size, a reset, and a clock to handle the timing of the system. The rising clock edge is used for all time operations, and the reset (when set to low) will revert the current state to S0. As a Mealy machine, we should expect to see different outputs in the same state. This means that our input not only controls the progression of states but also the output of the system.

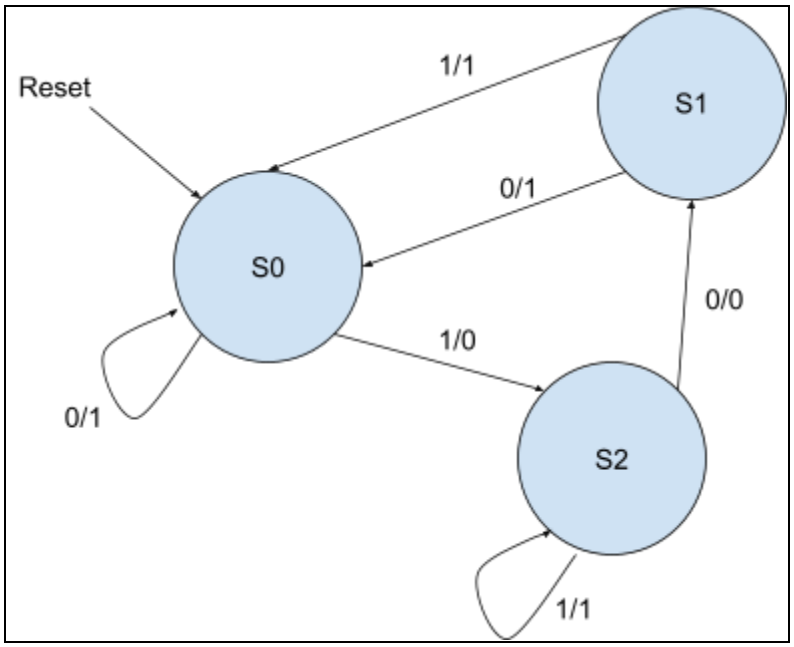

Figure 1: Mealy FSM

All processors need a structure to store data for operations, and registers are fast-access and low-capacity solutions. In the case of this system, each register in our array contains 32 logic bits at a count of 32 registers (Figure 2). Of these registers, only two may be read at the same time and one write may occur per clock cycle. To determine the address of these read-and-write operations, a set of 5-bit addresses is used to determine the position on the register array to be read or read as seen in Figure 3. Writing to the register will occur at the register that is pointed to by the write address, and this write operation is controlled by the write enable input to disable or enable at each clock cycle. Each read address corresponds to a register position to be read and output by the system. Unlike the rest of the registers, the first register in position zero will always output a 32-bit value of zeros. This was done by setting the register to zero internally at each clock cycle.

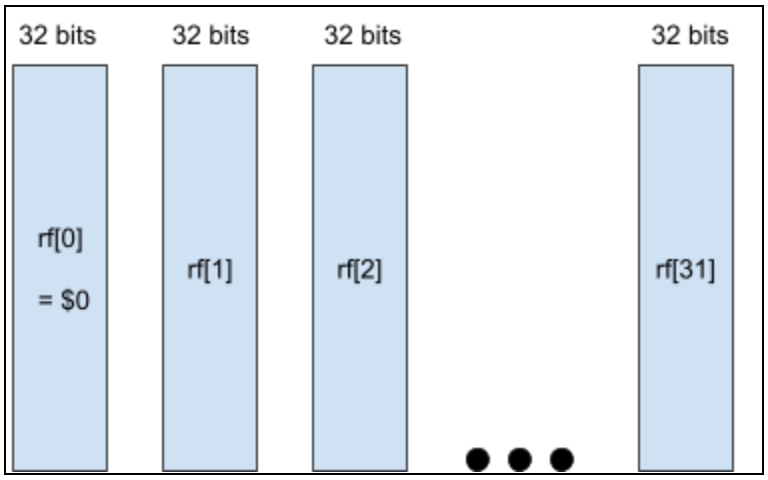

Figure 2: Register Array Diagram

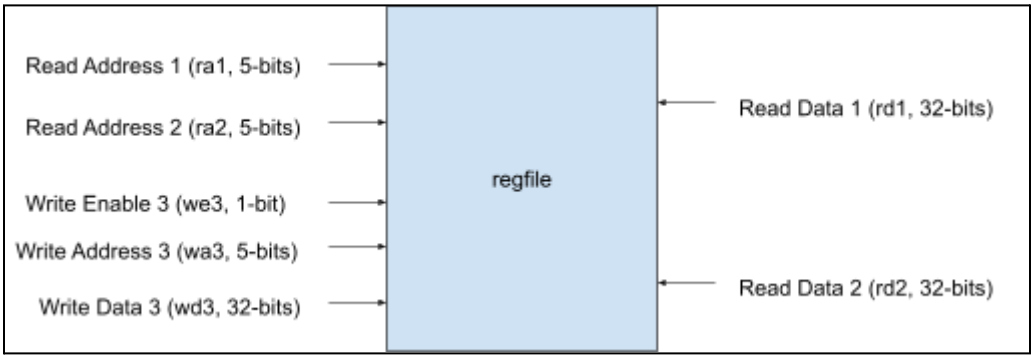

Figure 3: Registers I/O

Section 3: Detailed Design

The states in our finite state machine can be represented as a 2-bit enumerated logic variable. This allows us to define our state by a set of characters rather than its bit value to add readability to our design. This is done for both the current and next state in the Mealy machine. Our state logic can be driven combinationally, being updated on the rising edge of our clock cycle. The reset cycle will take effect on the falling edge of the reset signal. This will keep that current state at zero until the reset signal is high and the clock reaches it positive edge.

To accomplish the characteristics of proper registers, we must be able to read from our register array combinationally and must only write to our array on the rising edge of our clock cycle and save this value for a cycle via a flip-flop. This timing constraint prevents garbage data from being written to a register during the transition of the write signal. We can read and write to a pointed register using the addresses ra1, ra2, and wa3 which corresponds to the position on the register array with the same address value. We must also implement a $0 register that is read-only and always provides a value of 0. This was completed by multiplexing our

read-and-write data port to a constant 0 value if their corresponding address was also 0. This eliminates the need for a register zero in our rf array, changing our saved register range to [31:1]. The write enable port must also be added to conditional logic controlling our write process to only allow a register to be written to when this input is active.

Section 4: Testing Strategy

To confirm that our FSM would produce the intended Mealy behavior, a series of test cases were implemented with a time buffer that iterates through every possible state and output. These were saved to an output file that was updated twice per clock cycle. With this tool, as well as the waveform for the input and output, we can verify our state behavior.

The testing of our 32-bit register array required more extensive tools for the larger size of our I/O information. Using a series of test vectors, we can program the read and write operations of the register system as well as authenticate these operations with expected outputs. To make the composition of these test vectors easier and more human readable, a 28-character (112-bit) string with a hexadecimal base can represent each input, as well as an expected output for both read data ports. However, since the 1-bit write enable and three 5-bit address inputs become compacted in hexadecimal, and therefore less human readable or programmable, a 3-bit ignore array can be appended as the most significant side of each of these, increasing the character count to 31. Creating a test case for these inputs any higher than 5'h1f (5'b11111) will set the overflowing value to the ignore array. To validate the data from our read outputs, these will be compared to the test vector values at the negative edge of our clock cycle. Our error sum will monitor any discrepancy by incrementing it by one value. Ending on an error value other than zero will indicate something incorrect with our registers.

To verify the function of our register system, we first created a set of vectors to highlight key features required for our registers as seen in Figure. This includes writing on the clock cycle, reading a value of 0 for address 0 at all times, and writing to address 0 resulting in no change. In order to create a larger set of test vectors, a C++ program was used to create vectors using the same hexadecimal format which was added to the end of our vector file. These write and read each register in the array to demonstrate the functionality of the system.

```
//format:
//we3_ra1_ra2_wa3_wd3_rd1exp_rd2exp

//initialize write enable, write test value at address 1
1_00_00_01_a0a0a0a0_00000000_00000000

//read at addresses 0 and 1, write to 0 has no effect
1_00_01_00_b0b0b0b0_00000000_a0a0a0a0

//switch read addresses, write disabled by we3
0_01_00_01_c0c0c0c0_a0a0a0a0_00000000

//write disabled by we3
0_01_00_01_c0c0c0c0_a0a0a0a0_00000000
```

Figure 4: Initial Test Vectors for Regfile

Section 5: Evaluation

Our design only writes while having the write enable set to high which allows us to toggle this capability and choose when we are able to write. The read port from our register file outputs the value of the selected register combinationally. This same read port changes after a rising clock edge if the register that it was reading was modified on the same clock edge with no delay. Reading register zero always returns the value of zero and any attempts to write to it have no effect. This all culminates in a design that allows us to read and write to 32 32-bit registers in a simple and effective manner.

```
1 00 00 01 a0a0a0a0 || 00000000 00000000
1 00 00 01 a0a0a0a0 || 00000000 00000000
1 00 01 00 b0b0b0b0 || 00000000 a0a0a0a0
1 00 01 00 b0b0b0b0 || 00000000 a0a0a0a0
0 01 00 01 c0c0c0c0 || a0a0a0a0 00000000
0 01 00 01 c0c0c0c0 || a0a0a0a0 00000000
0 01 00 01 c0c0c0c0 || a0a0a0a0 00000000
```

Figure 5: I/O Behaviour of Regfile (5 ns spacing between points)

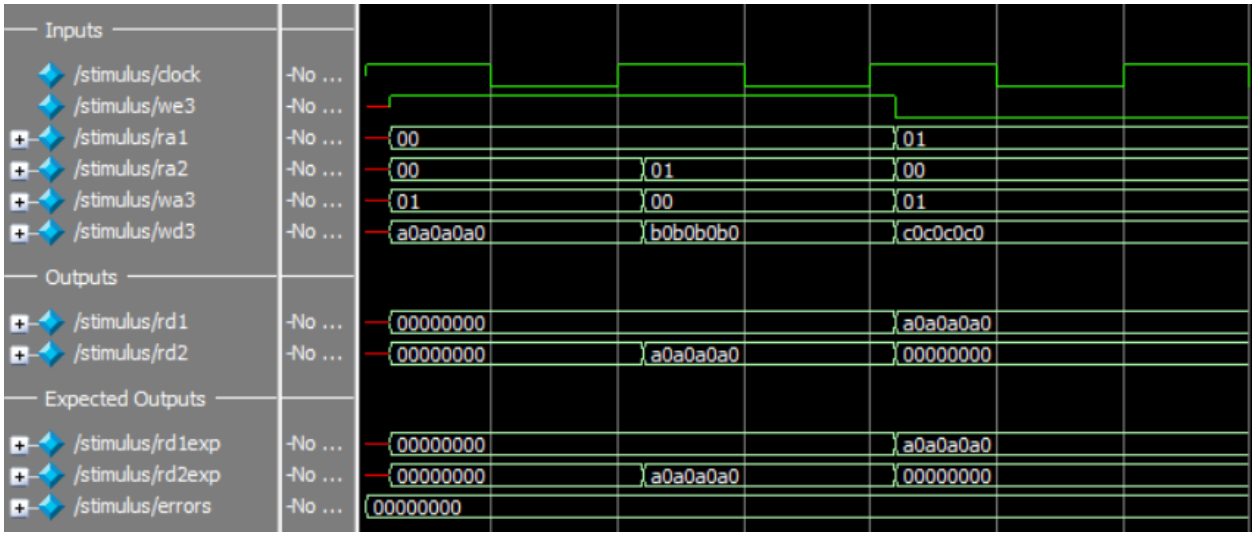

Figure 6: Waveform of regfile I/O Behavior for Test Cases