Computer Architecture Lab0 Report

Patrick Laverty
Ryan Stracener

Introduction of labs to Computer Architecture

College of Computer Electrical Architectural Technology
Oklahoma State University
Stillwater, OK
1/30/2023

Contents

1 Introduction

In this lab we touch on the basics of Verilog and HDL modeling. We will use ModelSim as our software to conduct our experiments. The test files are provided via a github link which we pulled by initiating the command '`git clone`' with the repo **.git**.

The **regfile** test-bench iterates through each address, assigning random values and ensuring that the same values are read out. The number of failers are counted, and reported at the end of the simulation. If there are no failures, then a success message is displayed.

The **FSM** test-bench simply prints the values in a human-readable form, to be confirmed with the existing FSM diagram.

2 Baseline Design

2.1 Part 1 (FSM)

The overall baseline design is given by the repository for the first part. We then tried the given test bench and then modified it to test variations.

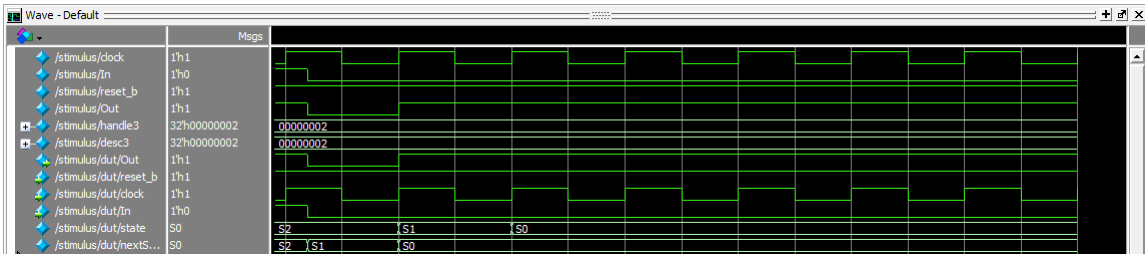

Figure 1: Given Simulation

We then modified the testbench file for the finite state machine and generated a new output simulation.

2.2 Part 2 (regfile)

Now that we understand how ModelSim works fundamentally, we can move on to Part II of this lab to design our own register file. In the second part of this lab we modified the given regfile and appended the necessary steps to create a register. We set the read ports to the internal memory and on every clock rising edge, we set the internal memory to the write data. With registers, we need to make sure that values are stored correctly and can be accessed from the appropriate read addresses. In our design, we will use internal **reg**s to store necessary data, and combinatorially set the read values to the appropriate internal **reg** value.

3 Detailed Design

3.1 Part 1 (FSM)

Using the already created testbench, we created 2 versions to test the given FSM module.

First Version: fsm_tb.sv

```
1  `timescale 1ns / 1ps
2  module fsm_tb ();
3     logic  clock;
4     logic  In;
5     logic  reset_b;
6     logic  Out;
7     // Instantiate DUT
8     fsm dut (Out, reset_b, clock, In);
9     // Setup the clock to toggle every 1 time units
10    initial begin
11       clock = 1'b1;
12       forever #5 clock = ~clock;
13    end
14    initial begin
15       // Tells when to finish simulation
16       #100 $finish;
17    end
18    // Write data to console
19    always @(posedge clock) begin
20       $display("reset, in||out: %b, %b||%b", reset_b, In, Out);
21    end
22    initial begin
23       #0  reset_b = 1'b0;
24       #12 reset_b = 1'b1; // -> S0
25       #0  In = 1'b0; // S0 -> S2
26       #20 In = 1'b1; // S2 -> S2
27       #20 In = 1'b0; // S2 -> S1
28    end
29 endmodule
```

Then, in the second version we outputted the state the machine is in which can then be manually observed to understand if it is working properly.

3.2 Part 2 (regfile)

The register file must contain the following requirements:

- **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.**

We designed a register module called **'regfile'** and designed a test-bench accordingly to define random unsigned values that will be fed into addresses to write to each register. Our complete implementation of the regfile is shown below.

regfile.sv

```
module regfile(
    input clk,
    input we3, // write enable
    input [4:0] ra1, ra2, wa3, // two 5-bit src regs, and one 5-bit dest reg
    input [31:0] wd3, // write data
    output [31:0] rd1, rd2 // read 1 and 2
    );

    // Internal memory
    reg [31:0] rf [31:0];

    // Set read ports to value, unless address is 0
    assign rd1 = (ra1 == 5'd0) ? 32'd0 : rf[ra1];
    assign rd2 = (ra2 == 5'd0) ? 32'd0 : rf[ra2];

    always @(posedge clk) begin
        if(we3 && wa3 != 5'd0) begin
            rf[wa3] <= wd3; // Set internal memory to "write data"
        end
    end
endmodule
```

4 Testing Strategy

4.1 Part 1 (FSM)

Using a .do file, we were able to simulate our SystemVerilog altered testbenches for the given FSM module.

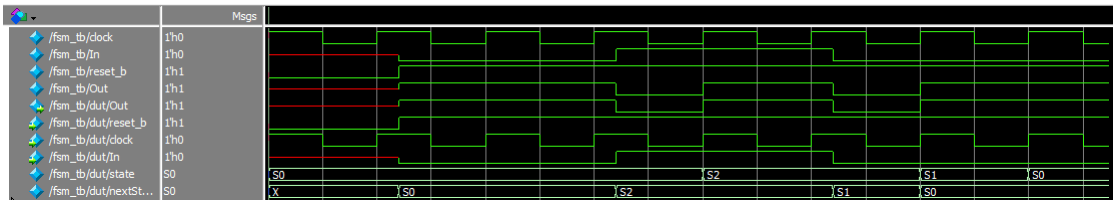

Figure 2: Testbench Version 1: Using Random Numbers

Using the output for each state, we manually observed that it was functioning properly.

Testbench Version 2: Manual Observation

```
reset, in||out: 1, 0||1
reset, in||out: 1, 0||1
reset, in||out: 1, 1||0
reset, in||out: 1, 1||1
reset, in||out: 1, 0||0
reset, in||out: 1, 0||1
reset, in||out: 1, 0||1
reset, in||out: 1, 0||1
```

4.2 Part 2 (regfile)

Our **regfile.tb** file automates the testing process by writing to each address then checking that the value read back is correct. A counter keeps track of all of the failures, and prints the diagnostic data. If no failures occur, then all tests are passed and the design is verified to be working. Our simulation's outputs (both the waveform and terminal output) are shown below.

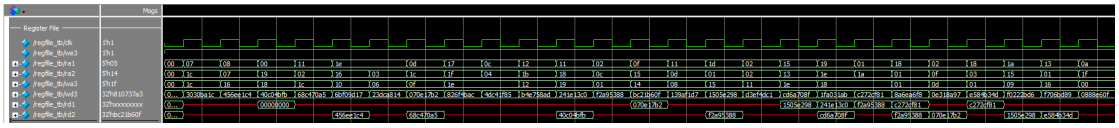

Figure 3: Registry Simulation

Our output using the better test-bench for our regfile to self-check and make sure that each address is writing properly.

```
Addr: 0x01 -- 0x00000000 = 0x00000000 = 0x00000000 -- PASSED
Addr: 0x02 -- 0x64c931db = 0x64c931db = 0x64c931db -- PASSED
Addr: 0x03 -- 0x6fa124bc = 0x6fa124bc = 0x6fa124bc -- PASSED
Addr: 0x04 -- 0x553ef0fc = 0x553ef0fc = 0x553ef0fc -- PASSED
Addr: 0x05 -- 0x468cca8e = 0x468cca8e = 0x468cca8e -- PASSED
Addr: 0x06 -- 0x6b3cf4f0 = 0x6b3cf4f0 = 0x6b3cf4f0 -- PASSED
Addr: 0x07 -- 0x59019edd = 0x59019edd = 0x59019edd -- PASSED
Addr: 0x08 -- 0x169bff1e = 0x169bff1e = 0x169bff1e -- PASSED
Addr: 0x09 -- 0x500626cb = 0x500626cb = 0x500626cb -- PASSED
Addr: 0x0a -- 0x48a2c399 = 0x48a2c399 = 0x48a2c399 -- PASSED
Addr: 0x0b -- 0x6bdaf01b = 0x6bdaf01b = 0x6bdaf01b -- PASSED
Addr: 0x0c -- 0x1692f159 = 0x1692f159 = 0x1692f159 -- PASSED
Addr: 0x0d -- 0x33dfb889 = 0x33dfb889 = 0x33dfb889 -- PASSED
Addr: 0x0e -- 0x44dc8ef8 = 0x44dc8ef8 = 0x44dc8ef8 -- PASSED
Addr: 0x0f -- 0xad2ef078 = 0xad2ef078 = 0xad2ef078 -- PASSED
Addr: 0x10 -- 0x03e5df54 = 0x03e5df54 = 0x03e5df54 -- PASSED
Addr: 0x11 -- 0x8d7f5292 = 0x8d7f5292 = 0x8d7f5292 -- PASSED
Addr: 0x12 -- 0x9909e093 = 0x9909e093 = 0x9909e093 -- PASSED
Addr: 0x13 -- 0x9a78d0ae = 0x9a78d0ae = 0x9a78d0ae -- PASSED
Addr: 0x14 -- 0x415f0b1b = 0x415f0b1b = 0x415f0b1b -- PASSED
Addr: 0x15 -- 0x5de66f8b = 0x5de66f8b = 0x5de66f8b -- PASSED
Addr: 0x16 -- 0x47a7c126 = 0x47a7c126 = 0x47a7c126 -- PASSED
Addr: 0x17 -- 0x4544ea70 = 0x4544ea70 = 0x4544ea70 -- PASSED
Addr: 0x18 -- 0x6b65c21d = 0x6b65c21d = 0x6b65c21d -- PASSED
Addr: 0x19 -- 0x60b1a1b9 = 0x60b1a1b9 = 0x60b1a1b9 -- PASSED
Addr: 0x1a -- 0x5fbdbb1e = 0x5fbdbb1e = 0x5fbdbb1e -- PASSED
Addr: 0x1b -- 0xacc4cd39 = 0xacc4cd39 = 0xacc4cd39 -- PASSED
Addr: 0x1c -- 0x3e981144 = 0x3e981144 = 0x3e981144 -- PASSED
Addr: 0x1d -- 0xa7657c44 = 0xa7657c44 = 0xa7657c44 -- PASSED
Addr: 0x1e -- 0x7209b7a9 = 0x7209b7a9 = 0x7209b7a9 -- PASSED
Addr: 0x1f -- 0xa9fdd362 = 0xa9fdd362 = 0xa9fdd362 -- PASSED
PASSED!
```

5 Evaluation

The overall conclusion of this lab is we now have the fundamentals to understanding ModelSim and creating test benches that interact with our SystemVerilog modules. Our regfile performs within the spec given, executing two reads and a write in a single clock cycle. We can now use our register file as a means of interacting with future designs in our labs.

Registers are very fundamental to any computer architecture as it is what stores ISA's, immediate-values, and register storage values. Many architectures today use registers as a means to machine code translation, such as RISC-V. This lab fundamentally showed us how we develop these kinds of machine interactions, as it takes an address to control the flow of commands by the processor. Using this we can create higher level modules such as interactive programs or arithmetic that utilizes these register values. As we were told in class, we will see how this lab truly will be utilized in the future.