Katy Kahla
James Kress

Part 1 – RegFile

Design

This part of the project asked us to create a register file that had the ability to reset itself to given values, read values, and write values to the registers. We also had to have setup and hold times in order to simulate hardware delays. The inputs to our register module were given to us in the project description and include the clock, reset, readRegA_sel, readRegB_sel, writeReg_en, writeReg_sel, and writeReg_dat. The outputs from the register modules were readRegA and readRegB.

We began our module by creating an array of eight 16-bit registers. From there, we designed the always block which would control whether a reset had occurred, a write had occurred, or a positive edge of the clock had occurred because it was specified that resetting and writing had to occur on a clock pulse or a reset. Below is a figure showing our always block design. Each non-blocking statements includes a time delay also to simulate hardware delay. When writing to a register, we check to make sure that register 0 was not the selected register so that the value could never be changed.

```
always @(posedge clock or posedge reset) begin    //alawys block for use when the reset or writeReg_en is active
    if(reset) begin                               //block to reset registers when reset is highf

      register[0] <= #5 16'h0000;                 //reset resgister 0 to 0
      register[1] <= #5 16'h0000;                 //reset resgister 1 to 0
      register[2] <= #5 16'h0000;                 //reset resgister 2 to 0
      register[3] <= #5 16'hc000;                 //reset resgister 3 to 16'hc000
      register[4] <= #5 16'h0000;                 //reset resgister 4 to 0
      register[5] <= #5 16'h0000;                 //reset resgister 5 to 0
      register[6] <= #5 16'h0000;                 //reset resgister 6 to 0
      register[7] <= #5 16'h0000;                 //reset resgister 7 to 0

    end

    else if(writeReg_en == 1'b1) begin                //else block to activate when the writeReg_en input is high
      if(writeReg_sel != 0) begin                     //Check to see if the selected reg is register is 0, if it is, we don't write to it
        register[writeReg_sel] <= #5 writeReg_dat;    //statement to save the data from the writeReg_dat input to the currently selected register
      end
    end
end
```

Figure 1: always block:

We initially attempted to have two always blocks, one to control reset and clock edges and one to control reading to a register but we realized that reading had to place asynchronous meaning that reading could not be contained in an always block. Therefore, we implemented two assign statements for reading from registers. Below is a figure showing this design. Each assign statement also has a time delay to simulate hardware delay.

```
assign  #5 readRegA = register[readRegA_sel];          //statemnt to read from the selected register and output the result to the readRegA output
assign  #5 readRegB = register[readRegB_sel];          //statemnt to read from the selected register and output the result to the readRegB output
```

Figure 2: Read Assign Statements

The final part of the project that we implement was the setup and hold time. We ran into many problems here figuring out what inputs to use to properly implement the hardware delays. We tried every input and ran into errors in modelsim. So, we finally concluded that we should use the 'writeReg_dat' in the setuphold command though this still has some errors that we were unable to figure out. We found that we would get a setup error on select times when we would write to a register. There wasn't any visible pattern as to why this was occurring so we were unable to fix the problem. In Figure 3, you can see our setuphold statement. In Figure 4, we included the error messages that we were unable to resolve. All errors revolve around writing to registers.

```
specify
   $setuphold(posedge clock, writeReg_dat, 3, 2);  //block to specify the necessary setup and hold times
endspecify
```

Figure 3: setuphold timing

```
# ** Error: ../src/regFile.v(37): $setup( writeReg_dat:24 ns, posedge clock:25 ns, 3 ns );
#    Time: 25 ns  Iteration: 1  Instance: /tb_top/UUT
# ** Error: ../src/regFile.v(37): $setup( writeReg_dat:274 ns, posedge clock:275 ns, 3 ns );
#    Time: 275 ns  Iteration: 1  Instance: /tb_top/UUT
# ** Error: ../src/regFile.v(37): $setup( writeReg_dat:454 ns, posedge clock:455 ns, 3 ns );
#    Time: 455 ns  Iteration: 1  Instance: /tb_top/UUT
# ** Error: ../src/regFile.v(37): $setup( writeReg_dat:554 ns, posedge clock:555 ns, 3 ns );
#    Time: 555 ns  Iteration: 1  Instance: /tb_top/UUT
# ** Error: ../src/regFile.v(37): $hold( posedge clock:755 ns, writeReg_dat:755 ns, 2 ns );
#    Time: 755 ns  Iteration: 1  Instance: /tb_top/UUT
```

Figure 4: Modelsim Errors

Schematic

Below, we have included the RTL schematic for the project. The first figure is the top level, and the second schematic is the pushed in view of the top level, showing the major components of the register file.

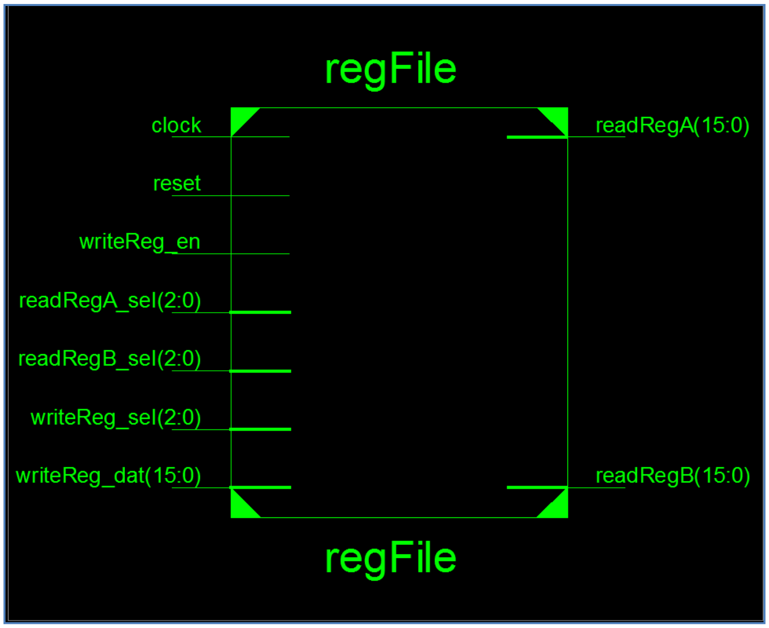

Figure 5: RTL Top-Level Schematic

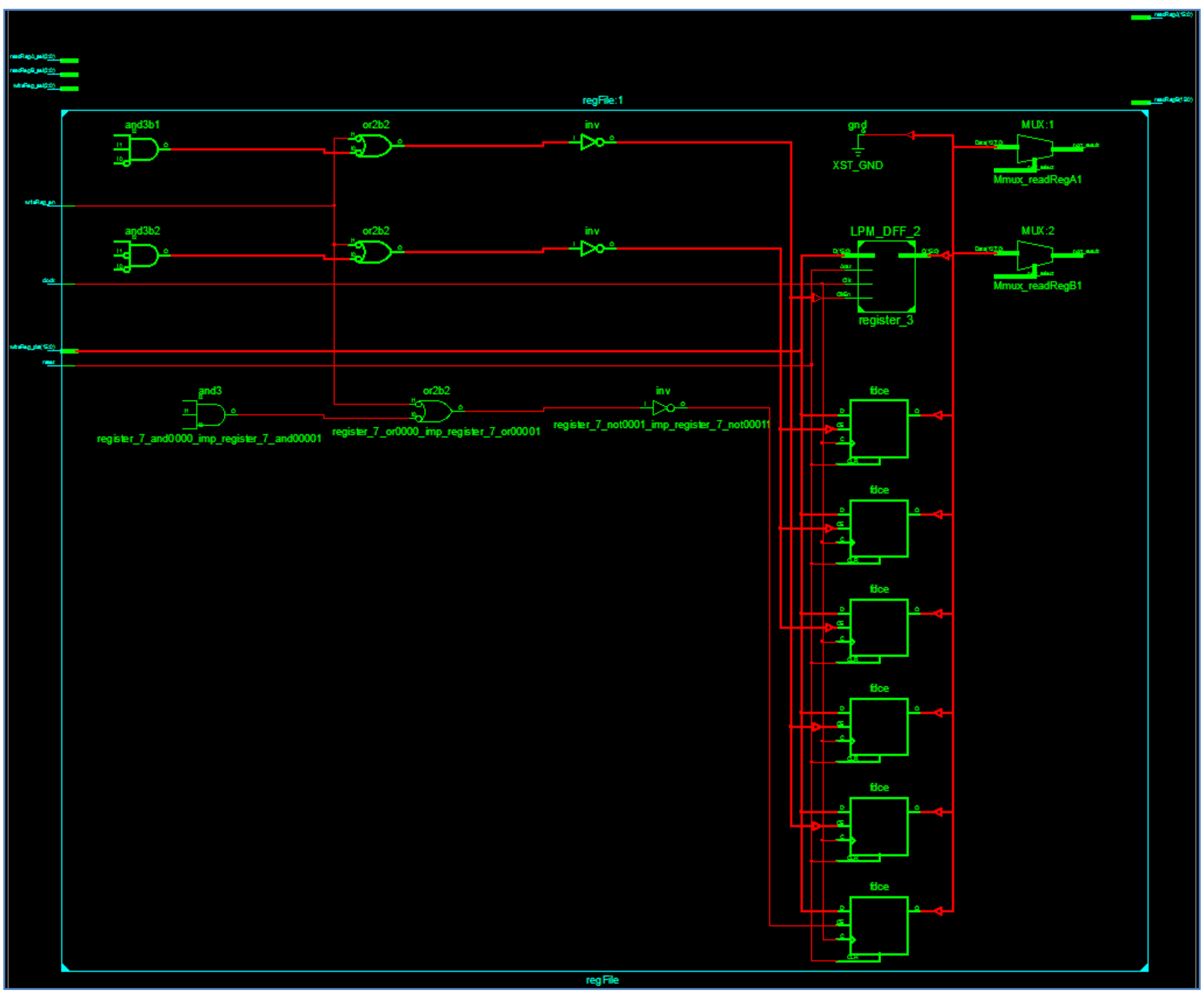

Figure 6: RTL Second-Level Schematic

Testing

We tested out project extensively to ensure that all of the program requirements were met. To start off, we tested using the simple testing script that had been provided with the program. Once we passed all of the tests it contained, we decided to add to and augment the given tests, to come up with a file that would test for all of the required functionality.

The first test we wanted to perform was to see what would happen if the register file was used without first being initialized or reset. The first set of tests, then, reads from the registers, writes to the registers, and reads from them again before the reset. The purpose of this test was to see the results, and gauge what the register file reaction would be. This test resulted in the result of 'FAIL' for the first reads, and 'FAIL for the second read from register 0. This confirmed for me, that register 0 was indeed able to not be changed by writing.

After the initial tests, we reset the registerFile, and began the real testing. First we tried a simple read, write, and read to the registers to ensure that the expected values resulted. After this test, we did two consecutive blocks of write with the same values to the registers. Thus, we write the same value twice, to make sure that writing redundant values does not affect performance.

Next, we did another write with different values to each of the registers to make sure that they all accept a new stored value.

The next test that we performed was to check that multiple resets in a row would not cause any problems, and that the register would still contain the correct values, these tests are documented in the test bench, and all passed.

An additional test that we decided to perform was to create a new method in the testing file to allow us to read from only one register. This means that we were able to test and see that we would read only from 'A' and only from 'B' at two given times. This seemed important because there will be times in the future that an operation will only require one register to be read.

One last test that we performed, but did not include in the test bench, is we tried to store too large of a value to a register. This test gave us an error on Modelsim, so we removed it, but the result of the test was that the number was simply truncated as expected.