**Project 1 - Register File Report**

Programmer: Victor Espinoza

SID: 010657450

CECS 440, Section 5; Tu/Th 9:30 - 11:45 A.M.

Lab Section 6

Due: Tuesday, February 10, 2015

**Project Description:**

The purpose of this lab was to create a (32x32) register file. This means that the register file contains 32 registers that are each 32-bits wide. This register file is capable of reading the data of two different registers at a time. In order to write data into the 32 different registers in this register file, I needed to have a specified address (wr\_addr) and the data that the user wanted to write to that specific address (wr\_data). I used a synchronous write enable port for this lab, meaning that if the write enable variable was a 1 on the active edge of the clock, then the register file wrote the desired data into the register referenced by wr\_addr. If the write enable variable was not asserted, however, then the register disregarded the request to write the given data into the register. In other words, the only time that data was written into a register is if the write enable variable was asserted during the active edge of the clock. Apart from these variables, my register file also had a reset bit that, once activated, reset all of the data within the register to zero. Additionally, I had two address ports (rd\_addr1 and rd\_addr2) that determined which registers were going to have their data read. These two variables determined the reference registers and the data contained within those registers was what was displayed on my two asynchronous read ports, rd\_data1 and rd\_data2. The write address and read address variables needed to be 5 bits wide in order to accommodate the 32 different register addresses; 4 bits would simply not be enough. The data ports were 32 bits due to the fact that our register file is 32 bits wide. That is basically how the register file works.

**Verification Description:**

For my test bench, after declaring all of the inputs, outputs, and local variables and instantiating the register file being tested, I first initialized all of my inputs to zero. After that, I then tried viewing the data on the register files, which was all zeros as expected. Then I set my write enable variable to high (in order to allow my register file to write data to the registers) and then I started writing data to each of the 32 registers in my register file. For my original data pattern, I chose to use the alternating values of 5555\_5555h and AAAA\_AAAAh to make sure that all of the 32 bits of data were being addressed in each register. Once I found this to be true, I then discarded this pattern and then chose a different pattern which consisted of writing down the register address as the actual data within that register. This made it extremely easy to make sure that each register that I was writing to / reading from had the correct data stored within it. I used my rd\_data1 port to display the first 16 register values and I used my rd\_data2 port to display the last 16 register values. This format ended up working rather well for me. After I verified the correctness of this pattern, I then made sure that my reset input worked as desired by asserting the input and making sure that this caused all of the data in the register file to revert to zeros. Then I simply reversed the order of the previous pattern and verified that the registers were working correctly once more. After verifying that all of the waveforms contained the values that I was expecting, I was convinced that my register file was working correctly and that I had successfully finished the project.

**Source Code:**

-Starts on next page

**Register File:**

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 1 - Register File

// Course: CECS 440

// Create Date: 11:05:58 2/1/2015

//

// Module Name: register\_file

// File Name: register\_file.v

//

// Description: This module creates a register file that is capable of writing

// to and reading from an array of 32 different registers. Which

// registers get written to and which registers have their values

// read from depend on the various 5-bit data address inputs. In

// order to write a value into a register, three conditions must

// be met: 1) A register address must be provided, 2) Data to

// store into the register must be provided, and most importantly,

// 3) The wr\_e input must be asserted on the active edge of the

// clock. If the write enable input is not asseted, then the

// register file does not write data into any register. Only once

// this input is asserted is the register file allowed to write

// data into one of its registers. The reset input clears all of

// the data within the register file and the rd\_data1 and rd\_data2

// output ports display the data values of the register referenced

// by their respective addresses (rd\_addr1 and rd\_addr2).

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module register\_file(clk, rstb, wr\_e, wr\_addr, wr\_data, rd\_addr1, rd\_addr2, rd\_data1,

rd\_data2);

//Initialize Inputs

input clk;

input rstb;

input wr\_e;

input [4:0] wr\_addr;

input [31:0] wr\_data;

input [4:0] rd\_addr1;

input [4:0] rd\_addr2;

//Initialize Outputs

output [31:0] rd\_data1;

output [31:0] rd\_data2;

//Initialize local variables

reg [31:0] REG [0:31];

//assign the output data to its respective register

assign rd\_data1 = REG[rd\_addr1];

assign rd\_data2 = REG[rd\_addr2];

//Behavioral section for writing to the register

integer i;

always @(posedge clk, negedge rstb) begin

if(rstb)

begin

//if the reset bit is asserted, then go

//through the register file and clear all

//of the data that was written to the

//individual registers.

for(i=0; i<32; i=i+1)

REG[i] <= 0;

end

else

begin

if(wr\_e)

//write the data into the desired register

//if and only if wr\_e is asserted.

REG[wr\_addr] <= wr\_data;

end

end

endmodule

**Register File Test Bench:**

`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 1 - Register File

// Course: CECS 440

// Create Date: 11:05:58 2/1/2015

//

// Module Name: register\_file\_tb

// File Name: register\_file\_tb.v

//

// Description: This module creates a test bench used to test the register

// file and make sure that it was working properly. I achieved

// this by writing a RegDump task and multiple RegWrite tasks

// to read and write different values into the registers within

// the register file. I first called the RegDump task to view the

// initial data in the registers, then I wrote data to the

// various registers using the RegWrite task. Finally, I called

// the RegDump task once more to verify that I was able to write

// the correct data to the registers in the register file.

//

// Verilog Test Fixture created by ISE for module: register\_file

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

////////////////////////////////////////////////////////////////////////////////

module register\_file\_tb;

//Inputs

reg clk;

reg rstb;

reg wr\_e;

reg [4:0] wr\_addr;

reg [31:0] wr\_data;

reg [4:0] rd\_addr1;

reg [4:0] rd\_addr2;

//Outputs

wire [31:0] rd\_data1;

wire [31:0] rd\_data2;

//Local Variables

integer i;

// Instantiate the Unit Under Test (UUT)

register\_file uut (

.clk(clk),

.rstb(rstb),

.wr\_e(wr\_e),

.wr\_addr(wr\_addr),

.wr\_data(wr\_data),

.rd\_addr1(rd\_addr1),

.rd\_addr2(rd\_addr2),

.rd\_data1(rd\_data1),

.rd\_data2(rd\_data2)

);

//Generate 10 ps Clock

always

#5 clk = ~clk;

initial begin

//Initialize Inputs

clk = 0;

rstb = 0;

wr\_e = 0;

wr\_addr = 0;

wr\_data = 0;

rd\_addr1 = 0;

rd\_addr2 = 0;

//verify the correctness of the register file

@(negedge clk)

rstb = 1;

@(posedge clk)

rstb = 0;

RegDump;

@(posedge clk)

//display the various data and address values

#1 $display("rd\_addr1=%h rd\_data1=%h || rd\_addr2=%h rd\_data2=%h",

rd\_addr1, rd\_data1, rd\_addr2, rd\_data2);

//write given data to the registers

RegWrite;

//verify that the registers had the correct data written to them

RegDump;

@(posedge clk)

//display the various data and address values

#1 $display("rd\_addr1=%h rd\_data1=%h || rd\_addr2=%h rd\_data2=%h",

rd\_addr1, rd\_data1, rd\_addr2, rd\_data2);

//reverse the pattern and verify contents once more

//make sure rstb works correctly

@(negedge clk)

rstb = 1;

@(posedge clk)

rstb = 0;

RegDump;

@(posedge clk)

//display the various data and address values

#1 $display("rd\_addr1=%h rd\_data1=%h || rd\_addr2=%h rd\_data2=%h",

rd\_addr1, rd\_data1, rd\_addr2, rd\_data2);

//write given data to the registers

RegWrite2;

//verify that the registers had the correct data written to them

RegDump;

@(posedge clk)

//display the various data and address values

#1 $display("rd\_addr1=%h rd\_data1=%h || rd\_addr2=%h rd\_data2=%h",

rd\_addr1, rd\_data1, rd\_addr2, rd\_data2);

$stop;

end

//this task declares the appropriate values for the

//read addresses. For this lab, I decided to have

//the first 16 registers' data be referenced by data1 and

//the last 16 registers' data to be referenced by data2.

task RegDump;

for (i=0; i<16; i=i+1)begin

@(negedge clk)

rd\_addr1 = i;

rd\_addr2 = i+16;

wr\_addr = 0;

wr\_e = 0;

wr\_data = 0;

end

endtask

//this task asserts the wr\_e input and provides the

//register file with a given register address and data

//to write into the register. This effectively gives

//the register file permission to start writing data

//into the provided registers, which it starts doing.

//For this particular pattern, I decided to write

//the same value that I was using as the register address

//to the register's data. For example, a register with an

//address of 1Ah will have a data value of 1Ah written to

//it.

task RegWrite;

begin

for (i=0; i<32; i=i+1)begin

@(negedge clk)

{rd\_addr1, rd\_addr2} = 10'b0;

wr\_addr = i;

wr\_data = i;

wr\_e = 1;

end

end

endtask

//this task asserts the wr\_e input and provides the

//register file with a given register address and data

//to write into the register. This effectively gives

//the register file permission to start writing data

//into the provided registers, which it starts doing.

//This pattern is simply the reverse of the pattern

//described in the RegWrite task. For example, a register

//with an address of 01h will have a data value of 1Eh

//written to it.

task RegWrite2;

begin

for (i=0; i<32; i=i+1)begin

@(negedge clk)

{rd\_addr1, rd\_addr2} = 10'b0;

wr\_addr = i;

wr\_data = (31-i);

wr\_e = 1;

end

end

endtask

endmodule