**CMPE-260 Laboratory Exercise 02**

**Register File**

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

John Judge

Performed 2/25/16

Submitted 3/3/16

Lab Section 04

Instructor: Dr. Lou Beato

TAs: Nick Barlow

Sergio Martins

Lecture Section 01

Professor: Dr. Reza Azarderakhsh

**Table of Contents**

1. Abstract …………………………………………………………………………...2
2. Design Methodology………………………………………………………………2
3. Results……………………………………………………………………………..5
4. Conclusion………………………………………………………………………...6
5. Appendix…………………………………………………………………………..6

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Abstract**

In this exercise a register file was designed and simulated. The register file was designed to store a generic value of bits with both read and write functionality to each register. In order to achieve this functionality the register file contained a total of eight registers, two multiplexers, and a 3-8 decoder. Once designed in Xilinx ISE, the design was simulated using ModelSim and a post-rout simulation took place. The design and functionality of the system was verified to be correct and the exercise was successful.

**Design Methodology**

This exercise was designed using the Xilinx ISE development tools and written in VHDL hardware description language. Before the register file could be written the overall functionality of the design as well as the parameters for the exercise needed to be comprehended. A register file is digital system that can store multiple bits of input data. Based on other inputs the system could either write the input data to a register or read the data stored in a specific register. A reset function was also necessary to clear the contents of all registers simultaneously. Figure (1.1) shows the block diagram with the input and outputs of the register file labeled. Once the functionality of the system was understood, the individual components could then be designed. In order to create the system to the parameters specified the following components needed to be designed; a register, a 3-8 decoder, and a multiplexer.

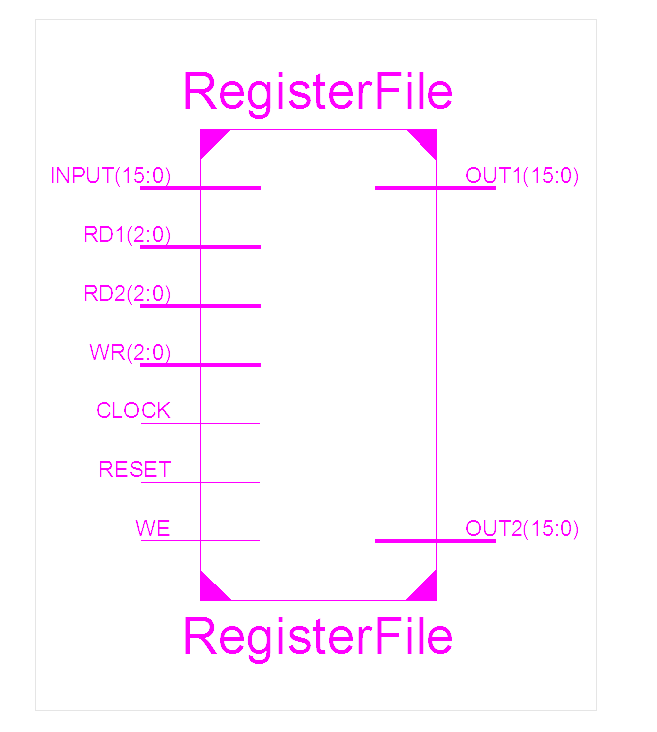


Figure (1.1): Top-Level Diagram of Register File

The first of these components, the register, was written using behavioral modeling. A register works as a memory unit being that it stores a value of multiple bits. A block diagram of the register with labeled inputs and outputs is shown in Figure (1.2). Using an active high 1-bit write enable signal an input value could be stored. An added active high 1-bit reset was used to clear any input written to the register. All changes to the contents of the register occurred on the rising edge of a clock input parameter. In this exercise the register was designed using generics for the bit width of the input data and the reset value. This allowed for the amount of bits the register stored and the value of the output when the register was reset to be easily changed later on. For the purpose of the simulation it was decided that the register should hold a 16-bit value.

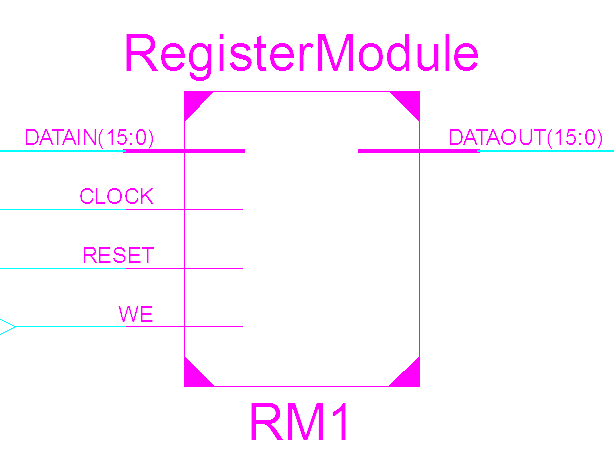


Figure (1.2): Block Diagram of Register Module

The next component, the multiplexer, was designed using architectural modeling.

A multiplexer uses a select signal to decide which one of its inputs will be output. In this case an 8-1 multiplexer was used, meaning, there was eight inputs and one output. A block diagram with labeled inputs and outputs can be seen in Figure (1.3). Using a 3-bit select signal one of the inputs could be selected and that data could be output. Once again, the input and output of the multiplexer needed to be a generic width and, once again, it was declared to be 16-bits.

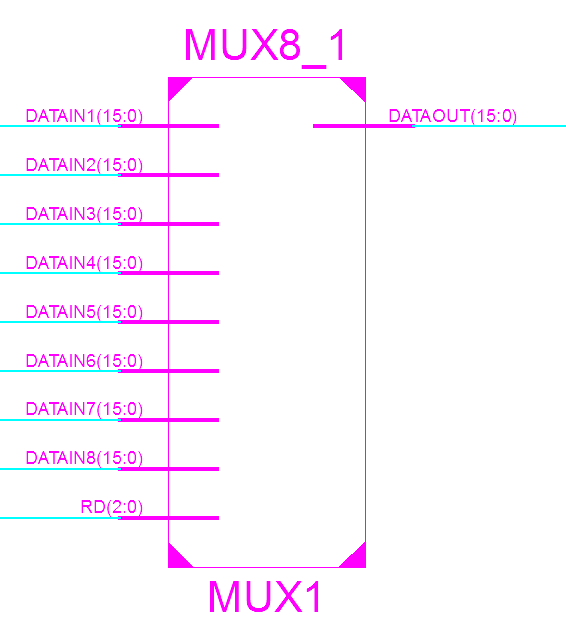


Figure (1.3): Block Diagram of Multiplexer

The final component of the register file was a 3-8 decoder. The decoder was written using dataflow modeling. This 3-8 decoder worked by using a 1-bit write enable signal combined with 3-bit write select signal and an 8-bit output signal. The inputs and outputs of the decoder can be seen in figure (1.4), the decoder’s block diagram. By combining the write enable signal to the most significant bit of the write select signal a 4-bit signal was created. When the most significant bit of the signal was 1 next three bites would select which register would be written to. This decoder was designed so that a 1 at the nth bit of the 8-bit output would select the nth register. The decoder provided a way to turn the register file’s 1-bit write enable and 3-bit write select signals into signal that could be used to decide which register was being written to.

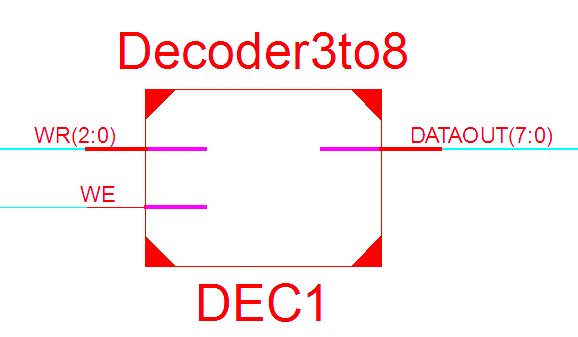


Figure (1.4): Block Diagram of Decoder

Once these individual components were designed the complete register file could be designed by mapping instances of the components together. The design of the register module was to first instantiate one decoder, eight registers, and two multiplexers. By mapping write select and write enable signals to the decoder a signal used to select which register was being written to could be created. Using the nth bit of this signal the nth register could be selected so this bit was then mapped to its corresponding register as the write enable. Each register also had the clock value mapped to its clock input and the 16-bit input data mapped to its input signal. The output of each register was mapped to the inputs on each multiplexer. The register-read-select signals were mapped to the corresponding multiplexer’s select signal. Finally, the output of the each of the multiplexer’s was mapped to the corresponding output of the register file. The entire system’s diagram can be seen in Figure (1.5) including all the connections.

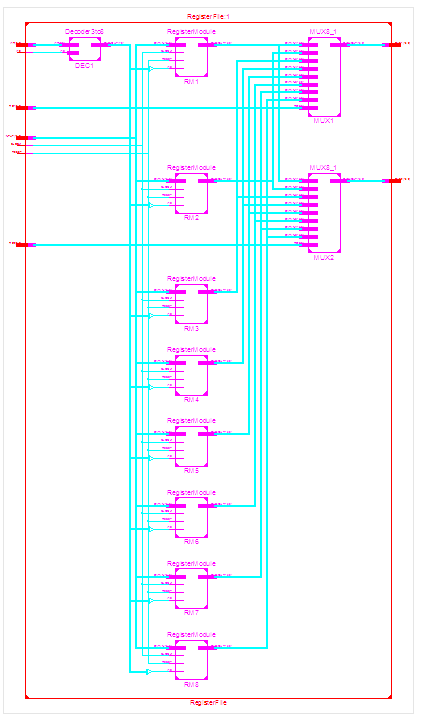


Figure (1.5): Register File Mapping Diagram

**Results**

*Design of Test Bench*

Once the components of the signals were mapped to the correct pins of the components a test bench could be written to test the design. The test bench was written in such a way that writing and reading to each register could be systematically tested. By first writing a process to oscillate the value of the clock with time a second process could be written to test the functionality of the design. By using two separate for loops the design was tested by writing values to each of the registers and then reading from each of the registers. The reset function was tested before and after the writing and reading took place. This test was used ensure the correct functionality of the circuit.

*Test bench Results*

The simulation results from ModelSim, which can be seen in figure (1.6), show the waveform generated by the test bench. As seen in the figure, the writing functionality was tested by setting the write enable to high. Then by cycling through the write select signals and incrementing the data input by 1 different values could be stored in each of the registers on the rising edge of the clock. Once each register had a value stored in it the reading functionality was tested by bring the write enable signal down to 0 and then cycling through the possible values of both of the register-read-select signals. By doing this the values of each of the registers was read by using both multiplexers. The simulation results matched what was expected. As the registers had values written to them the output of the circuit did not change. Only when the registers were being read from did the outputs changed. The reset function caused the outputs to go to zero. Using this testing strategy each of the components as well as the overall design was tested.

*Post-Route Simulation Results*

Following the simulation, a post-route simulation could take place. The post-route simulation added delays that could be expected in actual hardware. The results of this simulation can be found in figure (1.7). The results matched what was expected, the values changed with correct timing and the design provided the correct outputs. The results matched the expected results.

In order to check the use of generics in the design the code was manipulated to set the reset value to all 1’s. Another simulation took place, the results of which can be found in figure (1.8). The results of this simulation were identical to the other simulations except this time the reset value went to all 1’s when reset was triggered. This verified the use of generics was correctly implemented in the design.

**Conclusion**

In this exercise a register file was designed using smaller components and tested using ModelSim. The results of the simulation were verified to be correct. This indicated the design of the system was correct. The register file reacted accordingly to the different stimuli introduced. The use of generics, as well as the functionality of multiple bit memory systems, was reinforced during the exercise and these techniques can be implemented in future exercises. Since the objectives were met and the theory behind the exercise was proves the exercise was then concluded to be successful.

**Appendix**

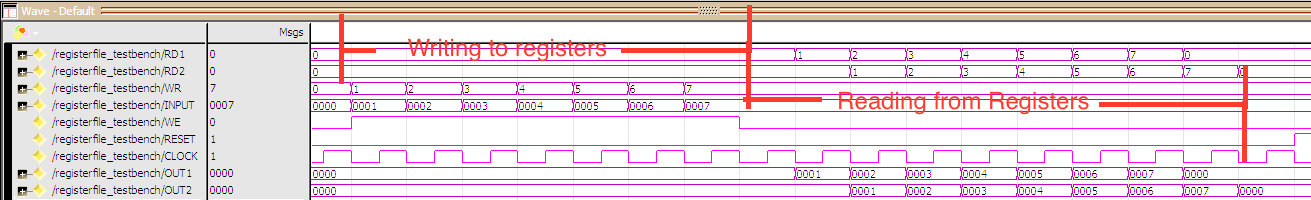


Figure 1.6: Simulation Results

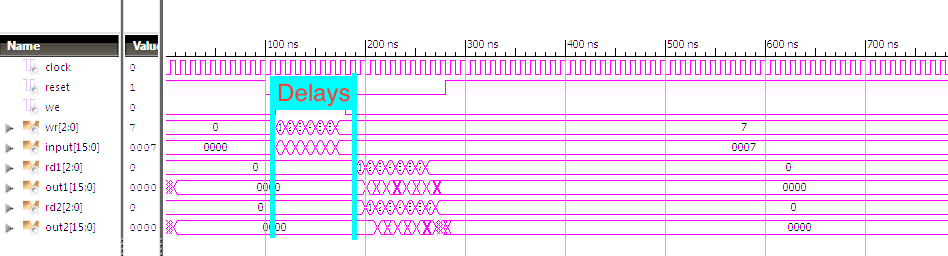
****

Figure 1.7: Post-Route Simulation

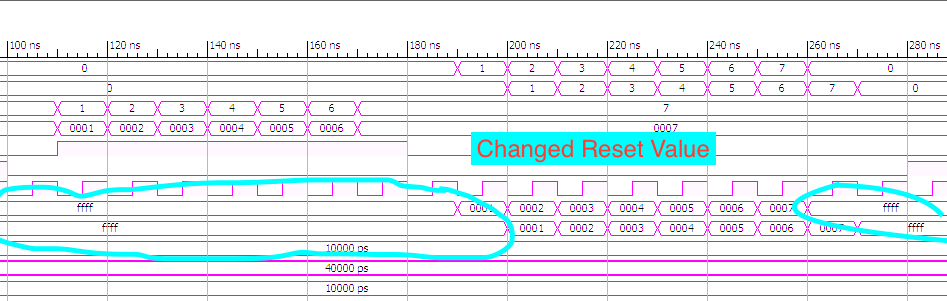


Figure 1.8: Post-Route Simulation With Changed Reset Value

