CMPE 260 Laboratory Exercise 2

Register File

Aden Crimmins
Performed: January 16, 2021
Submitted: March 1, 2021

Lab Section: 2
Instructor: Richard Cliver
TA: Corey Sheridan
Justin Soler
Jake Michalski

Lecture Section: 2
Professor: Mr. Cliver

Your Signature: *Aden Crimmins*

Abstract

The purpose of this exercise was to design the register file for an FPGA using the Xilinx Design Suite. The goal was to have the implementation capable of storing multiple words which can be overwritten or read from. This was tested using a testbench designed to confirm the successful function of each operation including edge cases. This lab was successful in showing how to design the register file as well as properly test it to ensure it functions as intended.

Design Methodology

The design was created using the Xilinx Vivado Design Suite and implemented on the Basys3 FPGA. This logic was created for the register file process. There are three sets of vector inputs Addr1, Addr2, Addr3 which contain a size-variable address within the register to allow the code to be modular. There are also two standard logic inputs which act as the write-enable and the clock, in addition to wd which contains the data to be written. There are two vector outputs RD1 and RD2 that contain the data in the address specified by the inputs. The schematic for the register file is shown in Figure 1.

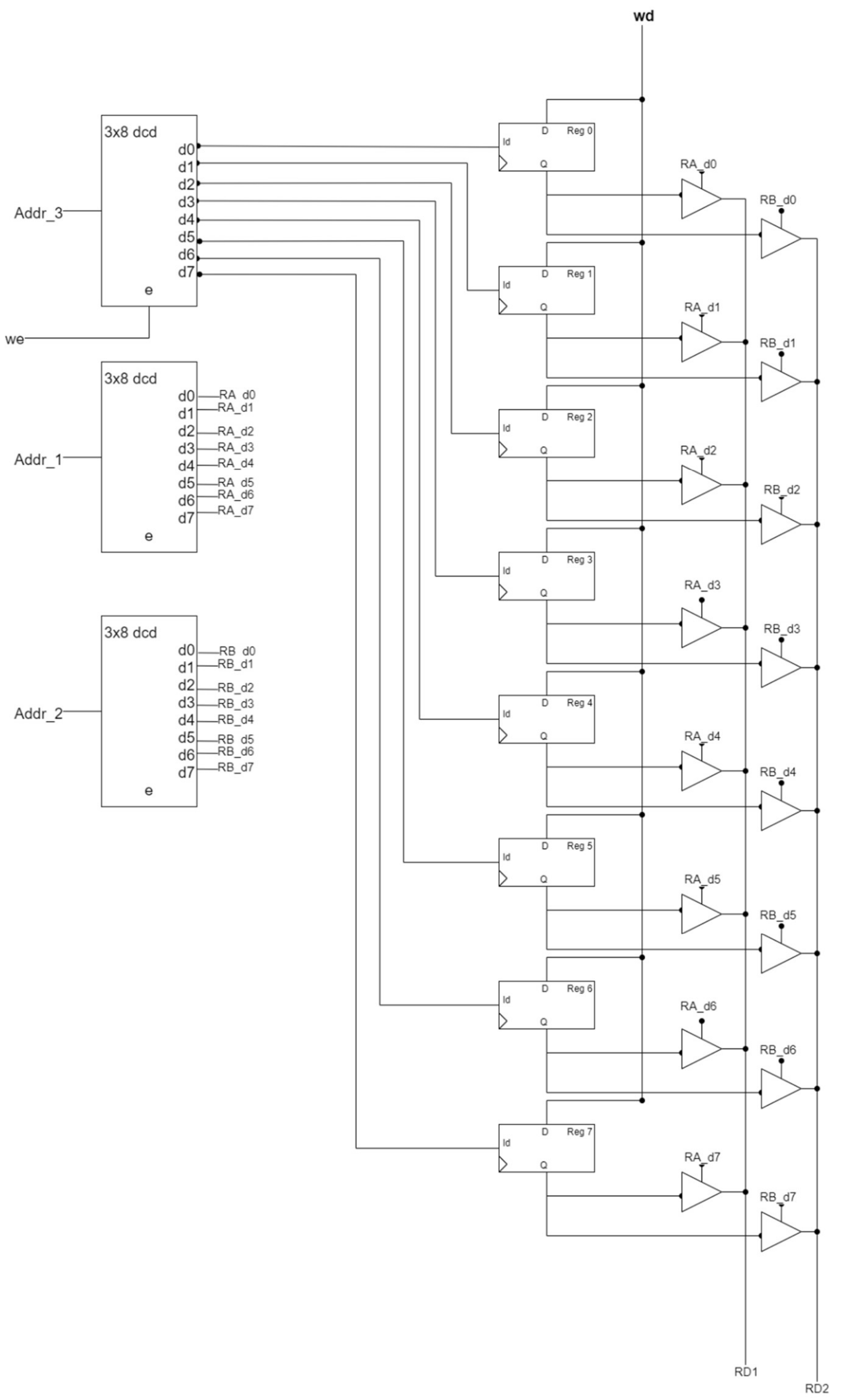


Figure 1: Register File Schematic

Figure 1 shows that the inputs Addr1, Addr2, and Addr3 which control which register is accessed. Each of the outputs is selected through the use of a tristate buffer only allowing the selected register to output to RD1 and RD2.

The boxes labelled Reg 0-7 shown in Figure 1 represent register modules used within the register file. Each register module is responsible for storing a number of bits specified by a provided bit width. If the write enable is set to one and any non-zero register is chosen then the specified register will be written with the data coming through the wd input. The output of these modules goes into a tristate buffer. If RD! or RD2 have selected the register module then the buffer will allow the data to flow from the module and determine the output of the complete register file. Figure 2 shows the block diagram of the whole register file rather than each individual component.

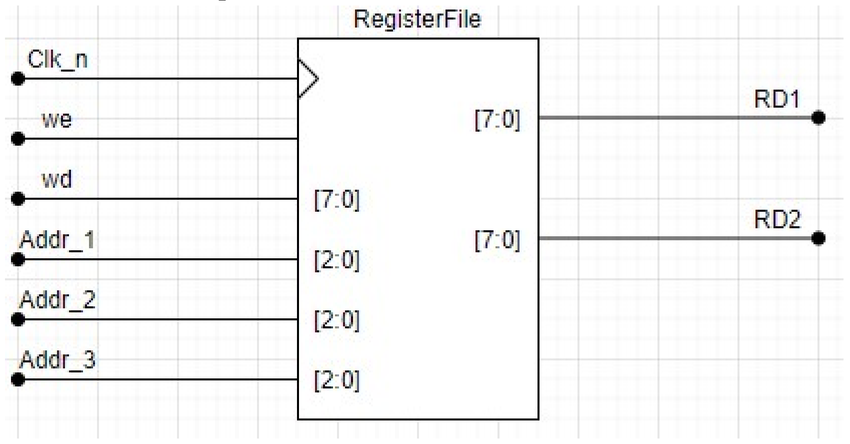
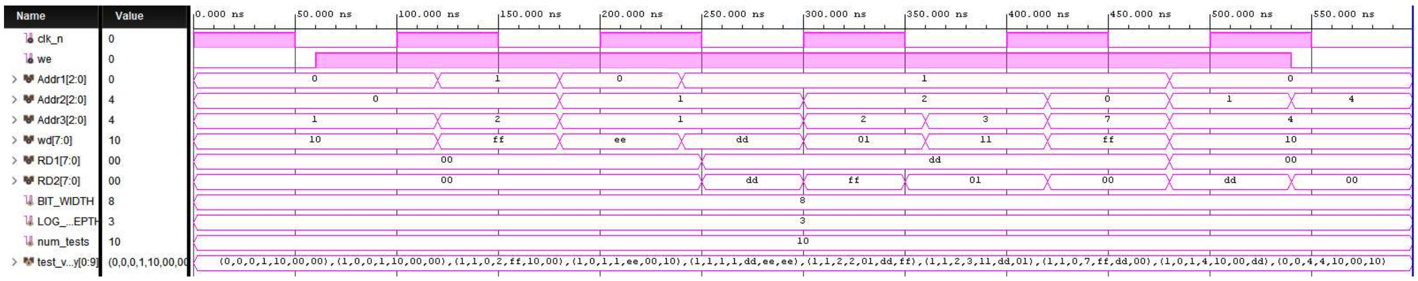


Figure 2: Register File Block Diagram

As shown in figure 2 he whole register file operates through 6 inputs and 2 outputs. The registers to be read from are specified by Addr1-2 and Addr3 indicates which register is to be written to. The we input turns on the ability for a register to be written when high and turns it off when low, and the data that is written to the registers comes from the wd input. The input clk_n controls the timing of the writing, with the data being written on the falling edge of the clock as long as the we is high. The registers specified by Addr1 and Addr2 have their output routed to the RD1 and RD2 outputs to be read by other components.

Results and Analysis

The testbenches created for this lab tested a variety of cases of each input values to ensure full functionality of the register file and its components. Figure 3 shows the Behavioral simulation of the register file.



```
(0,0,0,1,10,00,00),(1,0,0,1,10,00,00),(1,1,0,2,ff,10,00),(1,0,1,1,ee,00,10),(1,1,1,1,dd,ee,ee),(1,1,2,2,01,dd,ff),(1,1,2,3,11,dd,01),(1,1,0,7,ff,dd,00),(1,0,1,4,10,00,dd),(0,0,4,4,10,00,10)
```

Figure 3: 4-bit Behavioral Simulation

The simulation begins by testing the write-enable function with it being off on the first iteration, then on in the second the write-enable is turned to 1 but the register is still not written as the writing occurs when the clock is on its falling edge. On the next falling edge register 2 is written with a value of "ff", then on the next falling edge register 1 is set to "dd". Both outputs reflect this as they were set to get the value from register 1, then on the rising edge Addr_2 is changed to register 2 and the value of "ff" that was previously written is outputted. The next three clock cycles write to three different registers then the Addr1-2 inputs are set to 0 one after another causing both outputs to show "00".

The next test was the Post-Synthesis Timing Simulation shown in figure 4 which tests the same set of input as the behavioral simulation.

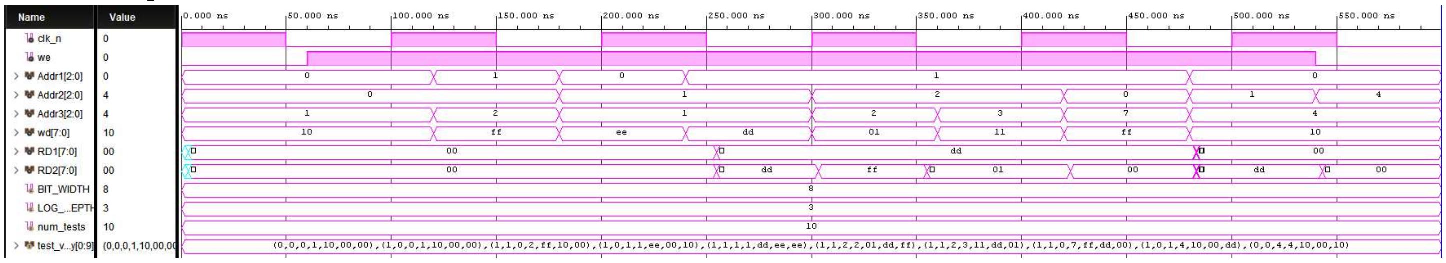


Figure 4: Post-Synthesis Timing Simulation

The Post-Synthesis timing simulation tests the gate level model of the code without accounting for any delays. This waveform shows the same results as the behavioral simulation which means that there arent any major design flaws that are visible within the synthesis simulation.

The next test was the Post-Implementation Timing Simulation shown in figure 5 which tests the same set of input as the behavioral simulation.

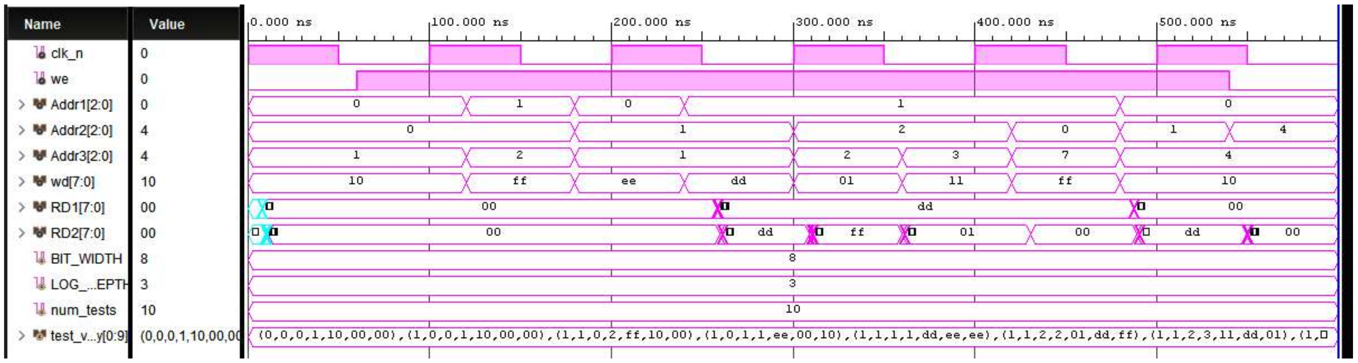
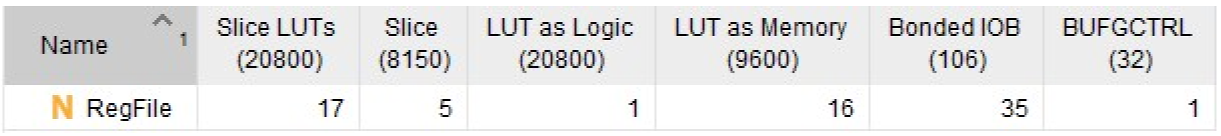


Figure 5: Post-Implementation Timing Simulation

The Post-Implementation timing simulation tested the implemented model as it would behave on the actual hardware. This includes the delays resulting from each gate in the model which can be seen whenever there is a change in the inputs as there is delay between that change and the resulting output. This delay is relatively small as there is mostly delay from the tristate buffers and the dcd inputs for the Addr1 and Addr2.

In order to see how heavy the requirements of the design would be to implement, the Utilization report shown in figure 6 is generated.

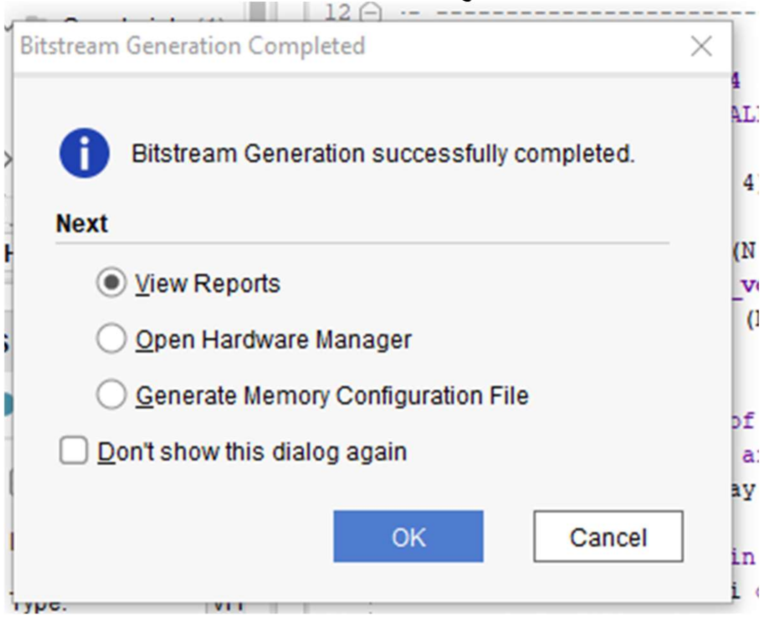


Name	Slice LUTs (20800)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Bonded IOB (106)	BUFGCTRL (32)
RegFile	17	5	1	16	35	1

Figure 6: Synthesis Utilization Report

The Synthesis Utilization Report shown in Figure 6 shows the number of slice LUTs utilized by the design. These are Look up tables create with small asynchronous SRAM and are used to implement combinational logic. The more LUTs used to implement the circuit the more resource intensive the design is.

If the code was to be tested on an actual board then the bitstream would need to be created then uploaded to the hardware. The creation of the bitstream is shown in figure 7.



Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

[✓ View Reports]

Open Hardware Manager

Generate Memory Configuration File

Don't show this dialog again

Figure 7: Bitstream Generation Successful

The Bitstream generation message in Figure 7 shows the successful generation of the Bitstream file, which is the file containing the necessary Information to program the FPGA with the created design. If there was an available FPGA to load the bitstream onto then the FPGA would be able to perform the operations of a register file when given certain inputs. would be produced in accordance with the simulated designs.

The schematic of the register file generated by Xilinx is shown in Figure 8.

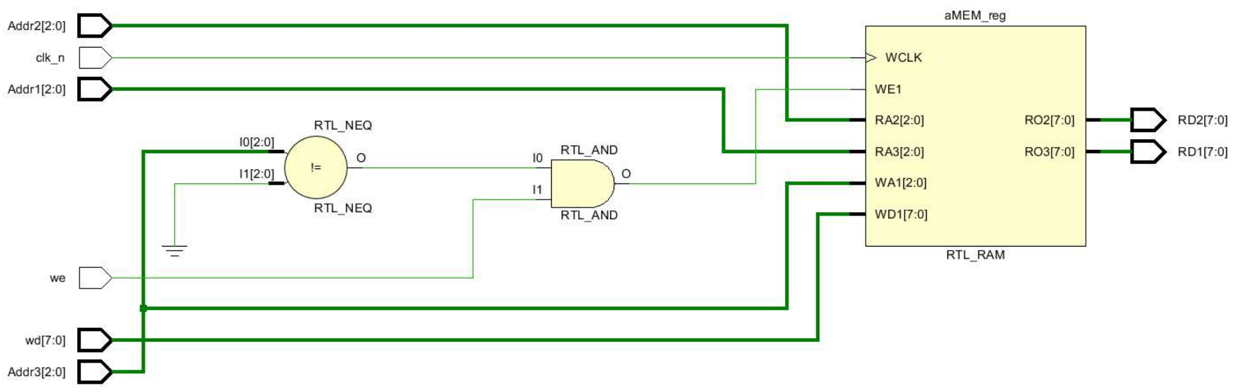


Figure 7: Bitstream Generation Successful

This schematic shows the inputs each going into the register file and the two outputs that are read from the register modules. The extra components show the case where register module 0 cannot be written to.

Conclusion

The register created was implemented successfully to complete the reading and writing functions that it is used for. The register file was tested in the Xilinx architecture through the Behavioral, Post-synthesis and Post-Implementation simulations when tested with the designed test bench. The circuits were inspected through the different design schematics to ensure that they were built and connected correctly. Both the schematics and simulations showed that the register file was designed correctly. A bitstream was also successfully generated which, if a physical board was present, would have allowed the design to be imported into an FPGA and tested in practice. There were minor issues when the register file due to the use of generics, but the generics were removed from the test bench it compiled correctly and there were no other major issues.

Questions

Q) What is assembly language?

Assembly language is the human-readable representation of the computer's native language.

Q) In the assembly code (sub a,b,c), which operand is the destination?

'a' is the destination operand

Q) Why are operands stored in registers?

Operands are stored in registers because they need to be retrieved fast and if it is stored in memory then it takes a long time.

Q) This lab creates 8 registers. How many registers does a MIPs have?

The MIPs architecture uses 32 registers

Q) How many operands can be read from a MIPs register at a time?
2 operands

Q) How many operands can be written to a MIPs register at a time?
1 operand