Kaleb Frey & Justus Matteson

CPEG324-010

5/3/19

**Lab Project Option 2: A FPGA Calculator**

**Abstract:**

The goal of this lab was to build an FPGA model of the 8-bit VHDL calculator that we designed and implemented in the previous labs. As a overarching goal for this lab, we had to create a visual presentation of our work that showed that our FPGA implementation was correctly working. While working on this lab we were able to better understand what an FPGA was and how to go through the process of implementing a program onto a board. Use of the FPGA was an important topic of this course, and we feel by doing this lab we have accomplished one of the goals that this course set.

Overall, we were able to successfully create the FPGA model of our 8-bit VHDL system. Our model can add or subtract two registers, load an immediate value into a register, compare two registers and skip instructions accordingly, and print out the value stored in a register. We accomplished this by mapping each of the on board switches to an instruction bit, and then displayed the outputs using the

**Division of Labor:**

This lab had four major areas that we needed to work on. Our first step was to fix the VHDL code for our single-cycle datapath so that it would correctly compile. Following the finishing of our original code, we had to implement our FPGA model based on this code. Once the FPGA version of our program was completed, we tested our code using the testbench that we had created previously. Finally, we had to complete the lab report that summarized our process and findings.

**Single-Cycle Datapath Finishing:** Kaleb Frey and Justus Matteson

**FPGA Implementation:** Kaleb Frey and Justus Matteson

**Testing FPGA:** Kaleb Frey and Justus Matteson

**Lab Report:** Kaleb Frey

**Detailed Strategy:**

We began by designing the single-cycle gate level circuit of our calculator, which involved a register file, controller, ALU, as well as many other muxes and gates for passing specific signals to the write data and other sections of the circuit. After making the necessary changes from our original design, the final circuit design can be seen below:

We began the process of transferring our gate-level circuit design to VHDL code by first tackling the two components that would be a basis of our code. When creating the ALU we used the code that we had previously made for the adder and subtractor in Lab 2; however, we made the necessary changes to provide fully functional code. For the register file component, we first implemented all the necessary inputs and outputs that would be required, and for the write data input we implemented a series of if statements that would only write to the desired register if the write enable signal was ‘1’.

After completing these components, we moved on to implementing the overall code of the calculator ISA itself. This was by far the most difficult part of the creating the VHDL model due to figuring out how the components interacted and making sure that everything worked correctly. All the instruction fetch was done in this section of the code and was determined through the opcode of each instruction. Determining how to appropriately go about skipping the instructions was also one of the hardest tasks of this lab.

Once we had all the code for our system done, we moved on the creating the testbench for the ISA. We accomplished this by having the testbench read each individual line of a text file and each individual bit on the lines. Our testbench would then send these bits to the main program of our system and the desired output would be produced. You can see the text file that we used, which lists the instructions that we used and the result of each instruction in Appendix C. We made a few changes to the testbench we had previously made in regards to the order of each instruction, and now each instruction should be accurately read and tested without any problems.

**Results:**

After completing the VHDL implementation of the design that we had fine-tuned at the beginning of the lab, we created a VHDL testbench that would verify whether the five instructions we were tasked with implementing worked correctly. The testbench used a test text file that had numerous binary levels instructions. In Appendix C at the bottom of this report you can find the testbench code alongside the description of what each section of lines should result with. We specifically tested the load immediate instruction first to make sure that the output matched what we were looking for. After verifying the load immediate instruction was functional, we moved on to making sure the add and subtract functions worked correctly using the print instruction and the values loaded in using the load immediate instruction. Finally, we tested the compare function with all the possible outcomes including: not skipping any lines, skipping 1 line, and skipping two lines. All the tests that we created produced the output that we anticipated and desired.

**Conclusion:**

We had to make a lot of changes to our ISA and code based on issues we had in our original plans for the calculator, but we believe that we were able to fix all the mistakes that we had made previously. The most difficult part of this lab was the concept of taking a circuit level design and creating a code model for it, since this was our first time ever completing a task of this depth. We struggled to make sure that all the components interacted correctly within the code, and if we had more time for the project, we would have added more functionality such as a reset “button” and made the current code easier to follow.

**Appendix A: Notebooks**

**Kaleb Frey:**

**Justus Matteson:**

**Appendix B: VHDL code**

**RegisterFile:**

**ALU:**

**Calculator:**

**Testbench:**

**Appendix C: Testing**

00000000 --prints R0 which contains 0

00001000 --prints R1 which contains 0

11010101 --loads 5 into R1

00001000 --prints R1 which contains 5

11101000 --loads 8 into R2

01111001 --adds R1 and R2 and stores in R3

00011000 --prints R3 which contains 13

10111001 --subtracts R1 from R2 and stores in R3

00011000 --prints R3 which contains 3

11010010 --loads 2 into R1

11100010 --loads 2 into R2

00101001 --compares R1 and R2 and skips 1 line

00000000 --this line is skipped

00100001 --compares R0 and R1 and does not skip

00001000 --prints R1 which contains 2

00110110 --compares R2 and R1 and skips 2 lines

00001000 --this line is skipped

00010000 --this line is skipped

00011000 --prints R3 which contains 3