Jaci Reichenberger, A20131719

Stephen Potter, A11341625

ECEN 4243: Computer Architecture

21 March 2021

Lab 3: Single-Cycle ARMv4 Microarchitecture Implementation in Verilog HDL

Section 1: Introduction

The purpose of this lab is to implement a single-cycle microarchitecture of the ARMv4 using System Verilog. The objective of this lab is to take Lab 2 and see the issues related to the implementation in hardware. For this lab, the tasks were to implement a single-cycle ARM machine in Verilog and ensure the implementation is correct through simulations of the test inputs. From testing various inputs, it was found that the code that was written was correct in the simulations. Thus, this lab showed the ARMv4 Machine working correctly in simulation.

Section 2: Baseline Design

The baseline design for this was to create code in the Decoder for the following instructions: ADD, SUB, ADC, CMP, CMN, MOV, LSL, LSR, ASR, ROR, MVN, AND, TST, ORR, EOR, TEQ, SBC, RSB, RSC, and BIC. Code was also created for the ALU in order to implement these instructions. The code followed the block diagram in Figure 1, which takes a 32-bit instruction and deconstructs it into bits for Cond, Op, Funct, Rd, and several others seen below. All these bits provide further bits that will determine what the instruction requires. For example, if the instruction is immediate, it will read the ImmSrc and utilize the Extend in order for it to have the ExtImm for the ALU. The ALU will then perform the function that the instruction requires (i.e., ADD, SUB, etc.) and produce a result, called ALUResult, in order to store in Data Memory, or, if the instruction does not use Data memory, loop back into the instruction memory. Following this logic, code was written based on what was required for the ALUControl, SrcA, SrcB in order to obtain the ALUFlags and the Result. ADD, AND, ORR, and SUB were given as examples that assisted in designing the remainder of the code in the decoder and the ALU sections. After the code was written, it was then compiled and debugged in order to run in simulation. Using the inputs provided and ModelSim, it could be determined whether or not the written code was working correctly by looking at the resulting waveforms.

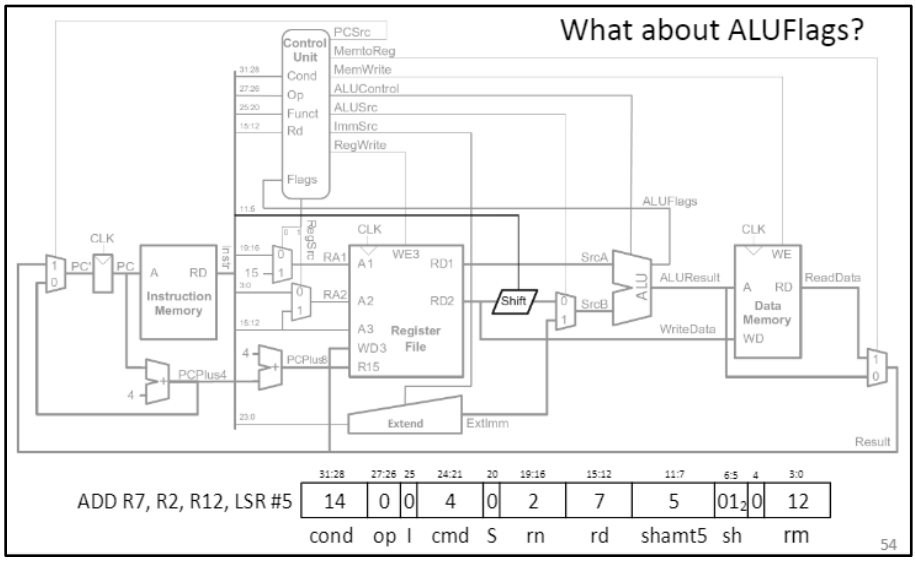

Figure 1: Block Diagram for Code

Section 3: Design

The most important aspect for writing the code was to understand both Figure 1 and the code that was provided for this lab. Figure 1 was already explained, so the provided code was very important in generating the code for the various instructions. The most important key to understanding the code is to be able to follow it in from the block diagram. Through careful perusal, the provided variables were identified and their functions for the code were understood. Thus, the code for the decoder and the ALU could be written.

The primary thought process was to determine which instruction to be decoded was based off of bits 20 to 25 of the 32-bit instruction. The code was deconstructed by calling these bits Funct and, for the decoder section of the code, putting in a case statement in order to further determine what 5 bits the ALUControl would be. A 5-bit code was chosen in order to differentiate between the 20 different instructions, which will be discussed in the following paragraph. For a Funct example, it was found from the ARM Reference Manual that the Funct for the instruction, ADD, was 0000. Likewise, AND was found to be 0010, and so on for the various instructions. Therefore, the Reference Manual was used to determine the 4-bits of the Funct, then the ALUControl could be set to further differentiate between the instructions.

The ALU for the code was written based off of the decoded instructions. This was completed by using the ALUControl determined from the decoder as a basis for the various cases in which a function was performed. Say, for example, it was determined that the ALUControl would be 00010 from the decoder section of the code. Then, jumping down to the ALU section of the code, the case for 00010 will make the Result be the logic AND between a and b, which are SrcA and SrcB, respectively, from Figure 1. This was done for all the required commands, and, through trial and error, was refined and debugged in order to get the desired results. The shift commands (MOV, LSL, LSR, ASR, and ROR), were slightly different from the other commands as their Funct from the decoder was the same. In order to differentiate between these commands, the immediate (I) and the shift operation (sh) variables were created and implemented in the code. This was done by ensuring that I pointed to Instruction bit 25 and sh pointed to Instruction bits 6 and 5 from the 32-bit instruction code. Based on this, the various operations could be differentiated, and the ALU can perform the correct operation for the instruction.

Special cases that only changed the flags were a bit more difficult to implement. However, through several iterations, it was possible to look at the flags for all the non-shifting operations by making the ALUControl for the shifting operations 10000. This allowed for an if statement to exclude them, and then the NZCV Flags could be determined by their various operations. Thus, the flags could be set for instructions like CMN, CMP, TST, TEQ, etc. For example, CMN adds SrcA and SrcB together in order to compare the negative. This instruction utilizes the ADD function in order to look at the Flags of the result. This means that the ALUControl bits can be the same for these two instructions, but the flags must be checked for the CMN. This logic was used for all the instructions that could be combined logically.

Therefore, the overall design of the written code was simple to implement once the logic for all of the code was understood. As the majority of the code was already written, the decoder and the ALU were added in order to complete the ARMv4 Machine. After the code was written, it was debugged, and then simulated in order to see if it worked.

Section 4: Testing Strategy

After finalizing the design for the single-cycle ARM ISA in System Verilog, the next task was to test the output behavior of the System Verilog code. To do this, there were several tools already provided, such as the arm3hex python program, which is similar to the arm2hex program from the previous lab but is built to handle system Verilog code instead of C source code. Other tools that were provided were the Inputs, written in Assembly, that would be converted and used to test if the ISA is working correctly. These assembly files must first be converted by the arm3hex shell and the resulting file is a .dat file. The .do file is then altered to include the particular .dat file that is to be tested, and the simulation includes this file when compiling the necessary files. Once the .do file is run, ModelSim takes over, and displays the GUI for inspection of the waveforms. The testbench for using this System Verilog was also provided as part of the source files for use. Most of these tools were all written and ready for testing, except the arm_single.sv file that had to be implemented for the lab. After figuring out how the Datapath and control logic worked in the main file, the next step was to test and verify the behavior of the ISA.

When testing the simulator, there was only one main course of action used to simulate and test the implementation. The way it was tested was to compile the proper input file into a readable .dat file for the simulator. Then that .dat file was referenced by the arm_single.do file and loaded in to be compiled. After initiating the simulation, the waveforms and output hex values were cross checked to verify results were correct. This process was repeated until the verification of correct behavior was satisfactory, or until all tests had been verified. Due to time constraints, the implementation of the input files was stopped after verifying some of the harder, more function filled tests.

The first test implemented was the addiu.s assembly file, which is an add immediate unsigned assembly program. This input file was run to determine if the implementation of the first few ALU functions were working correctly. When implemented, the waveform for addiu.s produced the correct results in ModelSim, as seen in Figure 2. The next test that was implemented was the fib.s input file. After running this file, the results showed that the simulator program ran correctly up to the 19th term. Through further analysis, it was found that it stopped at the 19th term due to the simulation runtime being capped at 765ns. Figure 3 shows the correct sequence for the first few terms of the Fibonacci Sequence. The next trial that was implemented on the simulator was the memfile.s. This was a very good candidate for testing because it contained a plethora of the functions that needed to be tested. If there was going to be a surefire test of the simulator, this would be it. The result listed in the comments of the assembly code stated that, at memory address 100, the result should be 7. After implementing this file into the simulator, the result obtained matched the previously given result. The result that was obtained from this implementation of the memfile.s assembly program can be seen in Figure 4. The andor.s assembly file was the last to be tested and, as expected, produced the correct results, as seen in Figure 5. This concluded the tests that were run with the simulator, which verified that the simulator was behaving as expected.

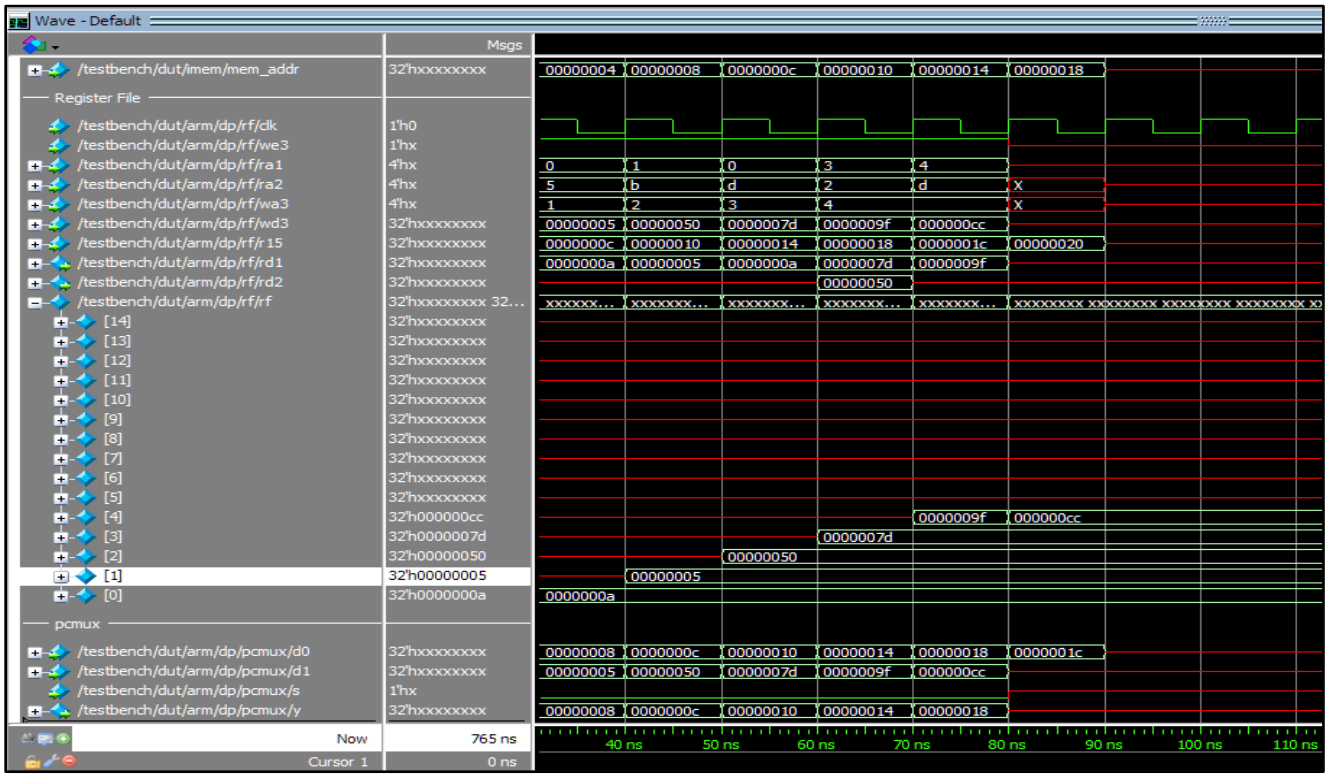

Figure 2: Simulation results of the addiu.s

Figure 3: Simulation Results of the fib.s

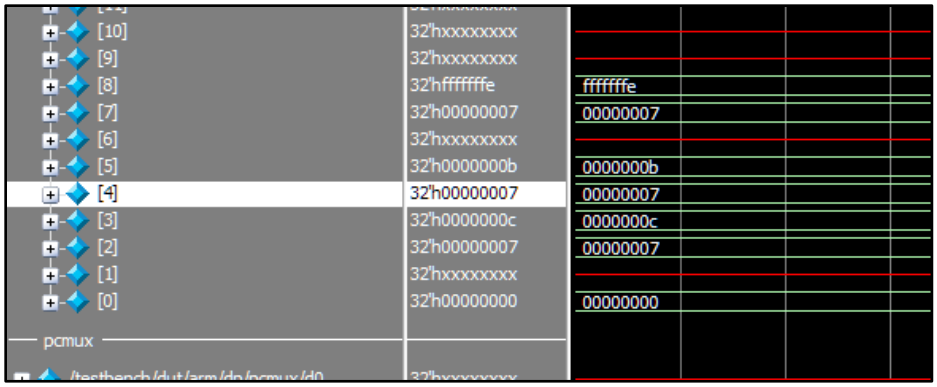

Figure 4: Simulation Results of the memfile.s (Shows 7 as the correct result in the mem address 100)

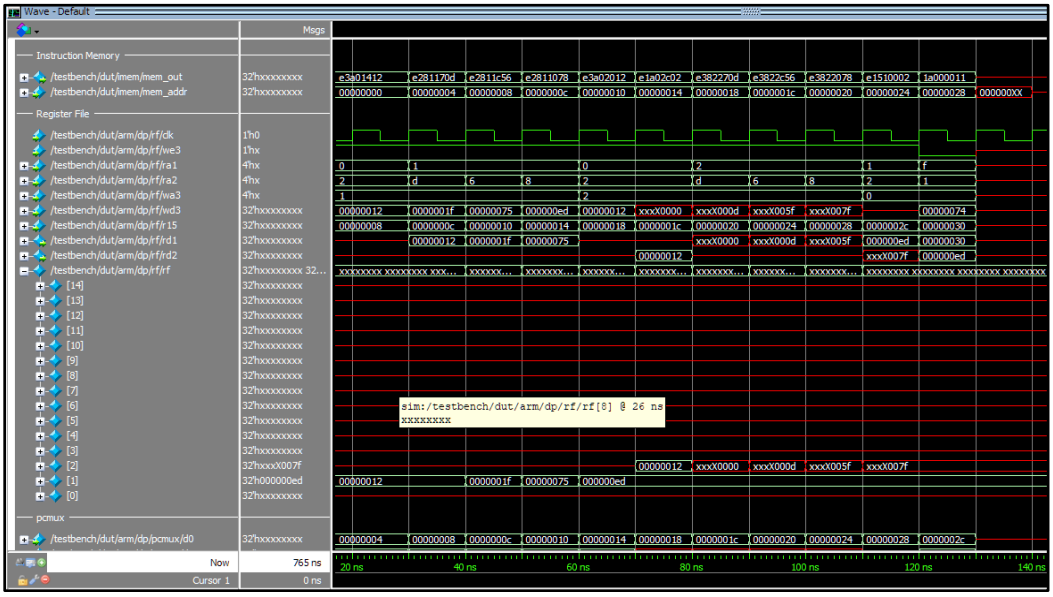

Figure 5: Simulation Results of the andor.s

Section 5: Evaluation

The results that were obtained from testing each of the input files showed that the code written for this lab was working as expected. The simple inputs from addiu.s and andor.s showed that the decoder was not only choosing the correct instruction, but also that the ALU was executing the instruction correctly. When trying more complicated inputs, such as the Fibonacci Sequence in the fib.s or the memfile.s, it was found that the instructions were executing correctly. The Fibonacci Sequence is able to show that the code can take two numbers, one being the previous number, the other being the current number, and adding them together to create the next number in the sequence (i.e., 1, 1, 2, 3, 5, 8, etc.). Likewise, the memfile.s was shown to work as intended as it successfully wrote the value 7 to the address 100. Since the written code was able to successfully complete these simulations, this means that the ARMv4 ISA Machine is working correctly for this lab.

Thus, the lab began with some of the source files preconfigured from the start. This left filling in the Decoder and the ALU in order to be implemented in Verilog. This was achieved by analyzing the SystemVerilog code in conjunction with the Logical block diagram (Figure 1). While tracing the Datapath in the simulator code, it was determined how the data from inputs and outputs would be handled. The next step was to understand the ALU control pathways and assign the proper functions to make them work correctly to the various functions that needed to be implemented. These include ADD, SUB, ADC, CMP, CMN, MOV, LSL, LSR, ASR, ROR, MVN, AND, TST, ORR, EOR, TEQ, SBC, RSB, RSC, and BIC. Some of these functions were already implemented prior to starting this experiment as examples. After understanding how these functions needed to be written, the functions were then tested for verification of results. This was done by doing a series of tests described in the testing strategy section. In sum, they were compiled and simulated in the ModelSim software. Then, the results were cross checked to confirm the simulator was running as intended. After testing several of these files, the conclusion was made that the simulator was working as expected and the implementation was successful.