Jaci Reichenberger, A20131719
Stephen Potter, A11341625
ECEN 4243: Computer Architecture
10 April 2021

Lab 4: Pipelined ARMv4 Microarchitecture Implementation in Verilog HDL

Section 1: Introduction

The purpose of this lab is to implement a pipelined ARM machine using SystemVerilog similar to the single-cycle machine created in Lab 3. This pipelined structure must implement the five stages discussed in the class lecture. The five stages include: Instruction Fetch (IF), Instruction decode and register file read (ID), Execution or memory address calculation (EX), Memory access (MEM), and Writeback to the register file (WB). Compared to the single cycle architecture from Lab 3, the pipelined architecture in this lab was harder to implement as it was more difficult to identify the errors for when the testing failed. This required more effort in the testing and debugging stage of this lab. In order to complete this lab, the pipelined ARM machine was implemented using SystemVerilog, then tested through simulation to ensure correctness. Through multiple iterations of the code, it was found that the completed product worked as expected.

Section 2: Baseline Design

The first thing done to create this lab was to understand the key differences between a single cycle and a pipelined ARM architecture. As seen in Figure 1, this diagram helped in understanding the key differences between the two. Whereas the single-cycle architecture must go through the five stages for one instruction before moving onto the next instruction, the pipelined architecture can work with multiple instructions at once in order to streamline the overall process and not have excess time go to waste like in the single-cycle.

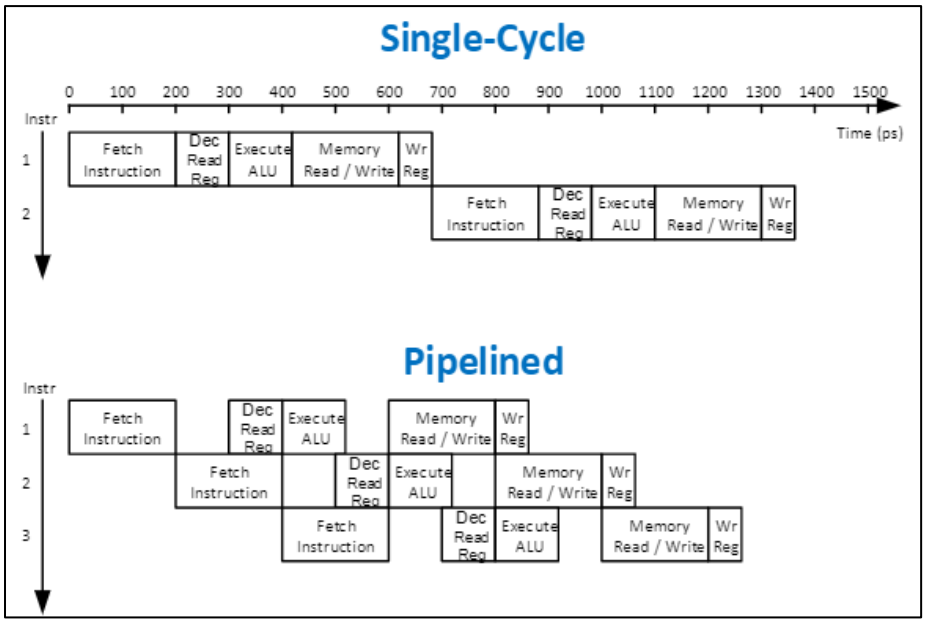


Figure 1: Single-Cycle vs Pipelined Architecture

After analyzing this figure, the block diagram in Figure 2, as well as the code already written from Lab 3, were used to write the code for this lab. The primary goal for implementing the pipelined structure was to ensure that each stage worked as intended, that the code compiled and worked as intended. That way, the first instruction was fetched, then the next instruction was fetched while the first instruction was decoded, and so on and so forth based on Figure 1. The majority of the code was already given, so all that really needed to be checked was the instructions working in tandem with the pipelined architecture in order to obtain the correct results.

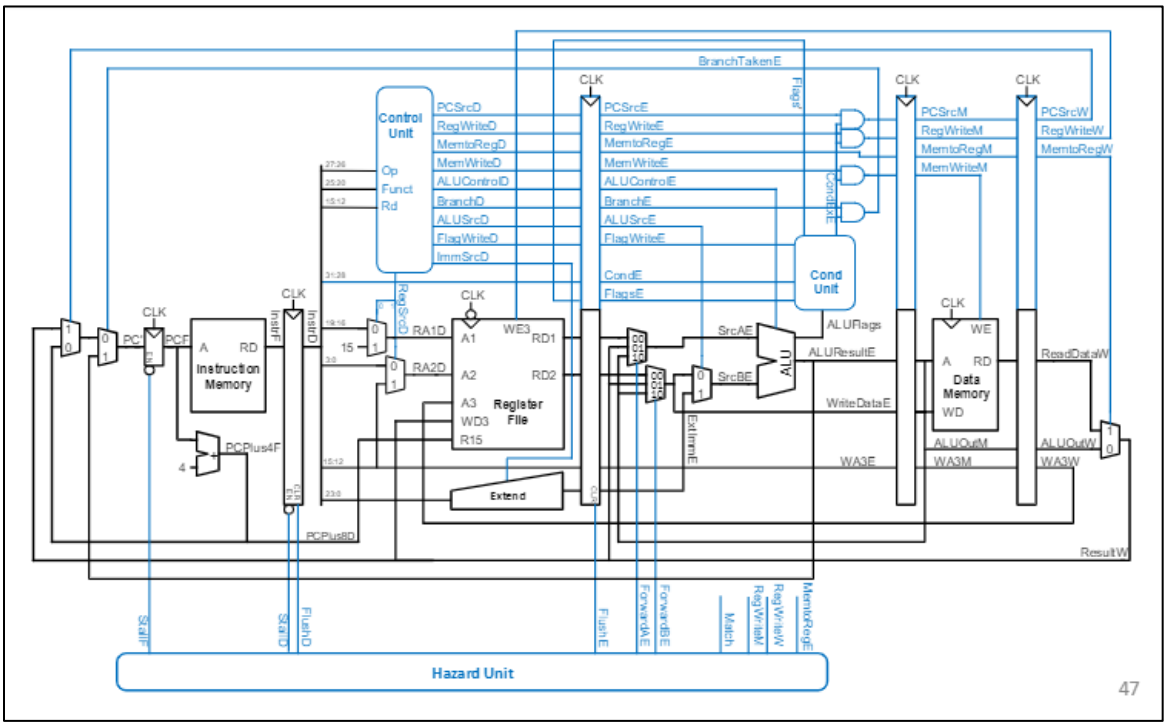


Figure 2: ARM Pipelined Datapath

Section 3: Design

Similar to Lab 3, the primary changes were made in the controller and the alu sections of this lab. This was to accommodate the following instructions: ADC, ADD, AND, ASR, B, BL, BIC, CMN, CMP, EOR, LDR, LSL, LSR, MOV, MVN, ORR, ROR, SBC, STR, SUB, TEQ, and TST. The Hazard Detection unit was already given, so it was fairly simple to modify the code from Lab 3 in order to create the pipelined structure. The primary concern for this lab was to see if the five stages for the pipelined architecture were working as intended with the instructions. The hardest part was identifying where the code was breaking when compiling errors occurred. The errors that were encountered were between rectifying variable names that were not the same and adjusting some logic from the previous lab that was not quite correct. This primarily centered on commands that were Immediate instructions. After parsing through those errors, the simulator compiled and allowed for the start of testing individual input files to test whether each instruction was operating correctly. Close attention needed to be paid to the Extend section, but after a few changes, the logic was corrected. After the instructions were implemented and the logic was checked the simulator was ready for testing, which is discussed in the next section.

Section 4: Testing Strategy

For testing the individual instructions, a similar approach to the arm_single.sv implementation was used. The input files underwent a conversion process by running them through a python shell that converts the assembly code into a .dat file, which can be read into the SystemVerilog implementation being tested. These files are then saved in the working directory, and the simulation is run using the command, “vsim -do arm_pipelined.do” in order to see the results in ModelSim. The .do file then sets up the window and references the main source, input file to be tested, the testbench (tb), and any other files that are needed to run the simulation. If it all compiles correctly in the source files, then ModelSim displays a window with waveforms of all the signals that were defined in the simulator. These waveforms are then inspected in the testing environment and used to confirm the proper functionality of the simulator. The waveforms of multiple tests can be seen in Appendix A.

When running the simulations for ADD and MOV, the addiu.dat assembly file was chosen because the assembly program contained those functions and was relatively short to be able to check all the results. The additest.dat was also used to double check the functionality of the ADD and MOV functions as well as the CMP, BNE, and ORR instructions were working correctly. The andor.dat file was used to again verify the previously discussed instructions. It also was a good candidate to show the pipelining in action because many of these instructions could be executed simultaneously. The next test that was performed was the arith.dat test to double down on a lot of the data processing operations and to check the results of those functions as well. The next test that was performed was the memfile.dat test to check that memory operations were performed correctly, and memory addresses were being activated correctly along with retaining the correct values. The last test that was performed was the memtest0.dat file which tested the STR, STRB, LDR, and LDRB instructions as well as some ADD and MOV instructions. This was another program that showed the pipelining in action and showed the hazard handling process was taking place.

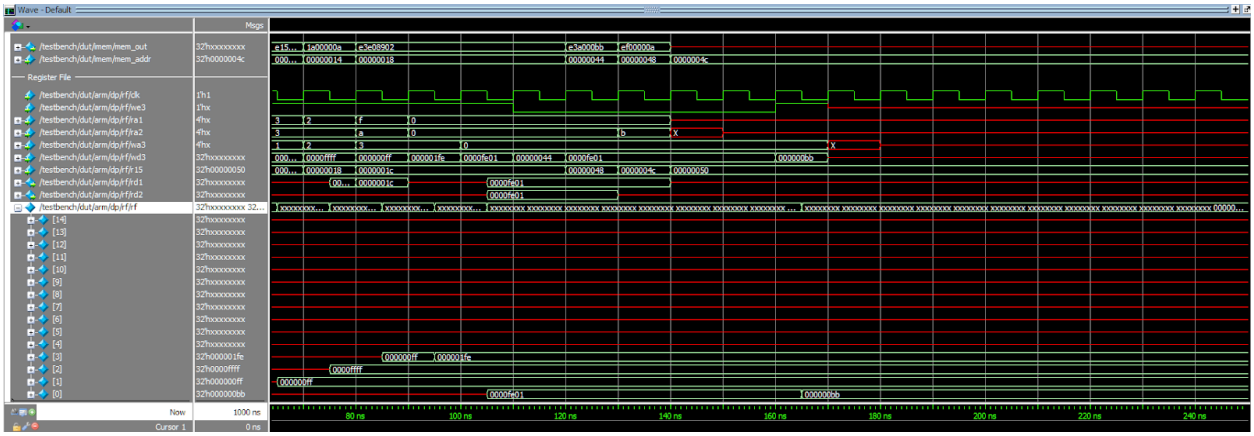
The results of each of these instructions were analyzed by looking at the output waveforms in ModelSim. When looking at the results in the register files and the memory data sections, the results showed that each of the registers were holding the correct values according to the operations in the input files. Each test produced similar and correct results except for a few particular operations regarding the LSL instruction. This may have been due to some bad logic in the implementations of the instructions. Other than the last errors that were encountered, the simulator worked correctly and displayed the correct results, as seen in Appendix A.

Section 5: Evaluation

As seen in Section 4, the results from the simulations worked as expected. After testing each input, it was shown that the written code was working as intended within the pipelined architecture. Though there was a problem with the LSL instruction that could not be resolved, the overall results were still correct. Being able to see the pipelining in action shows just how much more streamlined pipelined architecture can be compared to that of single-cycle architecture. Comparing the results from Lab 3 to Lab 4, it can be seen that pipelining is a faster process in simulation. Therefore, it can be concluded that, though single-cycle architecture is a simpler process, pipelined architecture is preferred as it will not waste as much time and energy as the single-cycle.

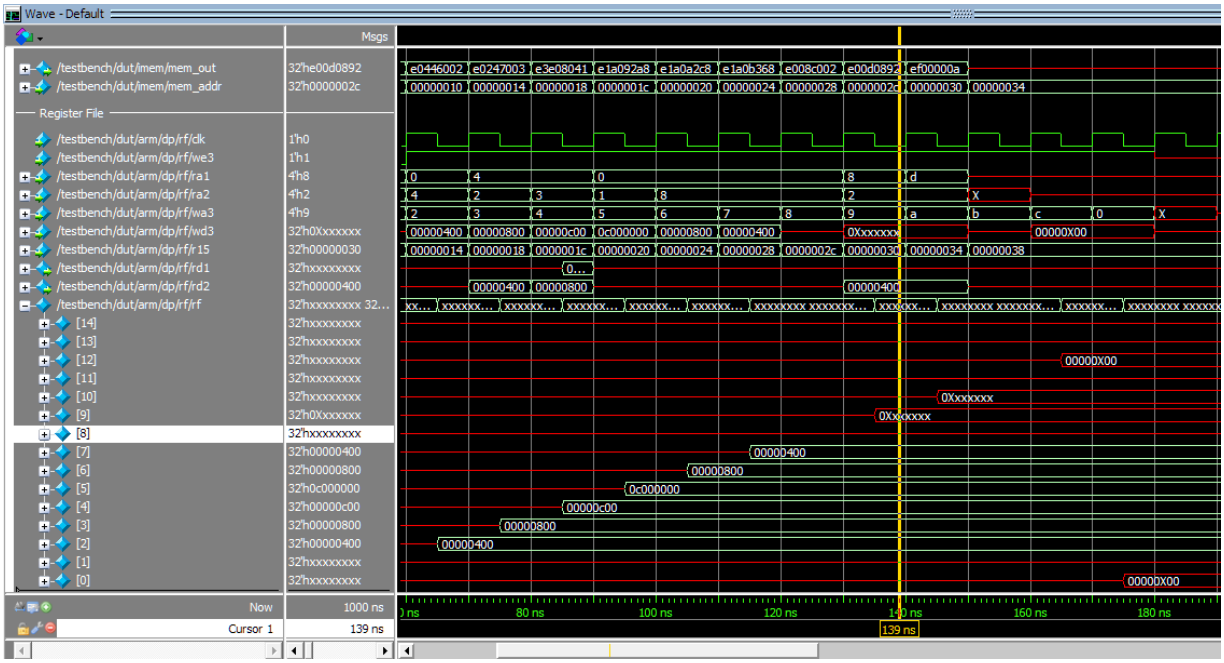
All in all, lab 4 showed how to implement an ARM pipelined architecture. Similar to lab 3, various instructions needed to be added into the given code in order to get a fully operating architecture. The code was written, then simulated using ModelSim in order to verify the operations were working correctly. After debugging the code, it was seen that all the tested simulations, minus the LSL instruction, were producing the correct results. The most interesting part of Lab 4 was seeing how it differed from Lab 3, even though the code that was required for both was practically the same. It showed the stark differences between a single-cycle architecture and the pipelined architecture. Being able to see the visual difference between the two helps with understanding what was taught in lecture. Being able to write the code for both and implementing what was learned in class helps to drive home the knowledge. Lab 4 exemplified a simple pipeline in order to show how various instructions can be run simultaneously. Being able to see pipelining in action from the memtest0.dat was really interesting and very different from running this test through Lab 3. Thus, Lab 4 taught how to implement a pipelined architecture using SystemVerilog.

Appendix A: Waveform Results

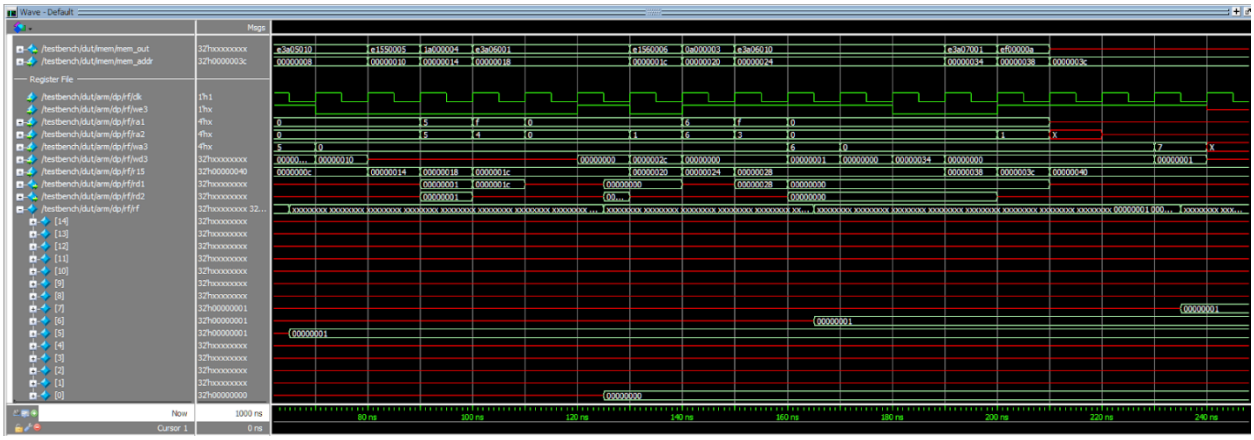


Additest.dat

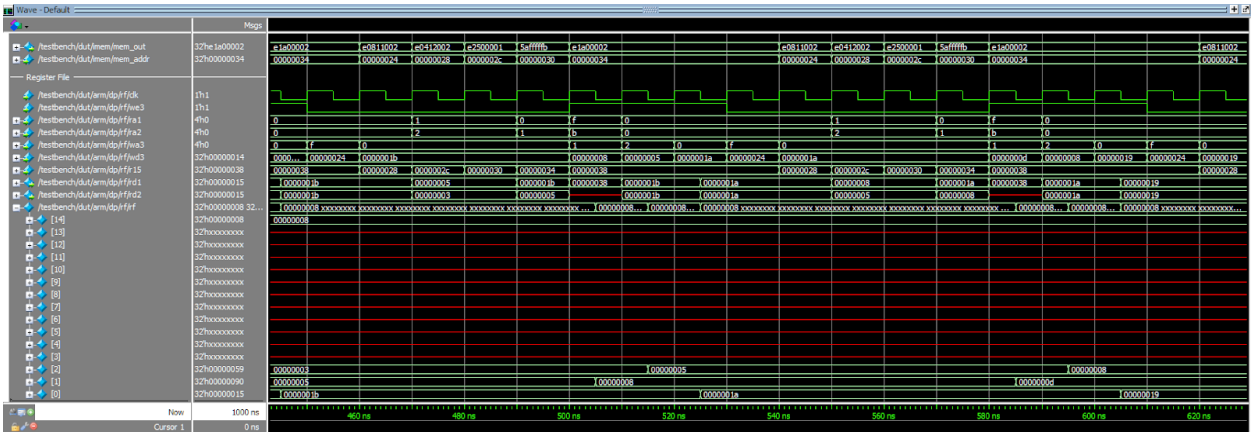
Addiu.dat



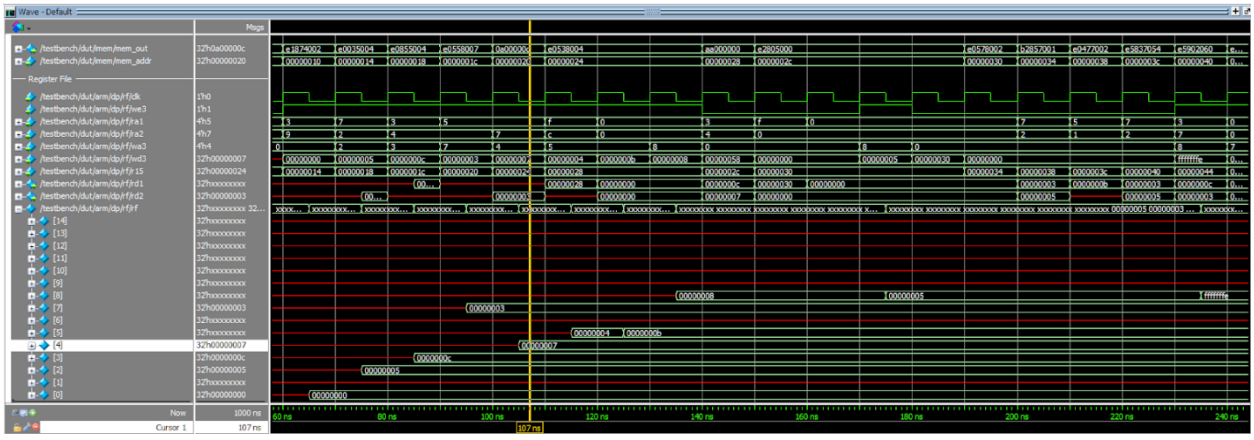
Arithtest.dat



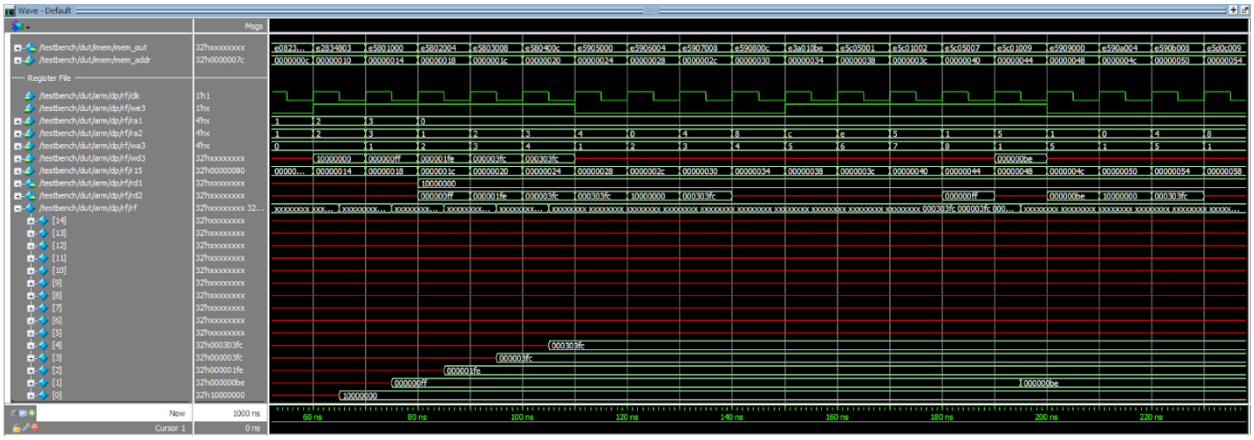
Brtest0.dat



Fib.dat



Memfile.dat



Memtest0.dat