**Design**

In a similar manner to the previous lab, the provided multi-cycle datapath was lacking a shifter module to handle the addressing modes for data processing instructions. While the multi-cycle architecture would have enabled us to implement the register-shifted register addressing mode, it would have required significant modification of the provided control logic and so was omitted for time.

The shifter module was placed in the decode stage of the datapath immediately in front of the second output of the register file, RD2. This is because the shifter module needed access to the current instruction, so implementing it in any other portion of the datapath would have required the instruction word to be propagated through every stage, leading to additional hardware complexity.

**Testing Strategy**

The first tests that were ran are essentially the same tests that were ran for the single-cycle processor. Programs that produced known outputs were loaded into the simulation, and then these known outputs were verified. For example, one of the provided test programs was intended to calculate the 32nd Fibonacci number and store it into memory. To verify this program, the number was calculated in advance and was compared to the value being written to memory at the end of the program. The numbers matched in our case, indicating that the program was functioning correctly.

Additional tests were performed to verify the hazard behavior. Two programs were written in order to trigger a data hazard and a control hazard. The data hazard was created by running a data-processing instruction and then having the following data-processing use the result of the first as an operand. If the data hazarding logic had not been implemented correctly, the second instruction would have read an incorrect value. The control hazard was triggered by a branching instruction that was set to take the branch. If the control hazard logic was incorrectly implemented, values pipelined from instructions following the branch directly in memory would interfere with the instructions that happen immediately after the jump location.

The same two testing strategies were used to test the FPGA implementation of the design.

**Evaluation**

The implemented pipelined ARM machine produced correct results for all provided input programs. This included programs that were designed to test the multi-cycle logic, including programs that intentionally created control hazards and data hazards.

This lab was much easier to understand and complete because of the experience gained from the previous lab. Since similar changes to the provided codebase were necessary between each lab, a significant portion of our code for the previous lab could be reused in this lab. Neither the shifter module nor the changes made to the ALU were greatly affected by multi-cycle architecture, save for some new signal names.

One thing that was readily apparent was the potential performance improvement between the single-cycle implementation and the pipelined one. While this wasn’t directly measured, the simulations in either lab were run at the same clock speed. But brief examination of the two implementations reveals that the critical paths between each pipeline register is much shorter than the critical path of the single-cycle datapath.

Overall, this lab allowed us to implement and better understand the pipelined architecture introduced in lecture. Additionally, the extra credit portion was a good refresher on working with FPGAs and using the Vivado software suite. With the cache implementation awaiting in the final lab assignment, the sum total of the lab work throughout the semester has served to create an intimate understanding of how computers function.