CMPE 260 Laboratory Exercise 5

Pipelined MIPS Processor

Aden Crimmins
Performed: April 13, 2021
Submitted: May 4, 2021

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

Lecture Section: 2
Professor: Mr. Cliver

By submitting this report, you attest that you neither have given nor have received any assistance (including writing, collecting data, plotting figures, tables or graphs, or using previous student reports as a reference), and you further acknowledge that giving or receiving such assistance will result in a failing grade for this course.

Your Signature: *Aden Crimmins*

Abstract

The purpose of this exercise was to combine each of the components of the MIPS processor made in past exercises, into one Pipelined MIPS processor. The outputs of each stage were designed to be captured into a register which passed each output on the same clock signal to make the design follow a pipelined operation. The Pipelined processor was first tested using a instruction set that ran each operation followed by an instruction set that created the first 10 values in the Fibonacci sequence. This exercise was successful in creating a full MIPS processor, designing and running test benches, and reinforcing VHDL practices.

Design Methodology

The design was created using the Xilinx Vivado Design Suite and implemented on the Basys3 FPGA. The components were separated into different stages within the processor. There was the Fetch Stage, Decode Stage, Execute Stage, Memory Stage, and Writeback Stage utilized to form the entire MIPS processor.

The Fetch stage contained the instruction memory which was populated with the instructions that were being run for one of the two test benches. On each clock cycle, while it was not reset, it would output the next instruction in the memory and increment the count for the following instruction on the next clock cycle. This instruction output then entered a register that delayed the signal by one clock cycle.

The Decode stage took the instruction that was output by the Fetch stage and analyzed the instruction to divide it into its separate components. It obtained the register values that would be used along with the signed immediate value and the register that would be written to. The Decode stage also had a control unit that took in the Opcode and Function portions of the instruction and determined different control bits. RegWrite determined if the register was going to be written back to. MemtoReg acted as the select for the Writeback stage determination between the output of the ALU and the value read from Data Memory. MemWrite was used to indicate if the Data Memory was being written to as part of the instruction. ALUControl dictated what the operation that would be used within the ALU would be, and ALUSrc determined if the second input to the ALU would be the second register or the immediate value. The last bit RegDst determined the destination register for the instruction to be written back to if it needed to store the calculated values.

The Execute stage took in the values for the instruction along with the operation to be performed and returned the resulting value as the ALUOut. This stage also utilized the ALUSrc and RegDst bits to determine the second input and the destination register from the two provided instruction values. If the SW command is used then the data from the second register is used as the data to write to memory while the output of the ALU remains as the Address.

The Data Memory stage takes in the ALU output, the destination register as WriteReg, the write data ad WriteData, and uses the MemWrite bit. The MemWrite bit determines if the data from WriteData is going to be written into memory at the location specified by the ALU output. The ALU output is also branch as a passthrough for the Writeback stage.

The Writeback Stage takes in the ALU output and the data read from memory and uses the MemtoReg bit to select one as the return back into the register file to be written. The WriteReg, and RegWrite values are also connected back to the Register file to specify the register that is being written within the Register file as well as if writing is enabled for the instruction.
The outputs of each of the stages were pushed into registers which delayed them for one clock cycle before releasing them into the next stage. The Control unit values that were passed into each stage but not used continued through the registers as passthrough bits until they were all used by the Writeback stage.

The Fibonacci Sequence was designed to test the functionality of the MIPS processor by calculating a well-known sequence and verifying that the outputs matched. The first ten instructions were used as setup starting the second number(t2) as 1 while also setting t3 and t4 as 1 and three respectively. The registers t3 and t4 were used as the address in the memory of where each value would be stored. The register t1 holds the first of the two numbers used throughout the sequence but it did not need to be initialized as it was already set to 0. After a few empty instructions the first two values were stored and t4 was incremented by 1 as only one storage indicator was needed from this point on. The next portion of code is repeated as part of the loop ending once it gets the tenth value. The first part of the loop adds t1 and t2 and stored it into t2, followed by subtracting t1 from t2 and storing it into t1. This has the effect of finding the next value and storing it into t2 while replacing the value in t1 with the previous value of t2. Three empty instructions are used to allow propagation time followed by t2 being stored in the next location in memory and t4 incremented once more to get the value for the next memory location. This process is repeated a total of eight times with the last value of 34 being obtained at the end.

Results and Analysis

The testbenches created for this lab tested a variety of cases of each of the input values for the functionality of the MIPS Processor using different operations to ensure full functionality of each stage and their components. Figures 1 and 2 show the behavioral waveform for first testbench.

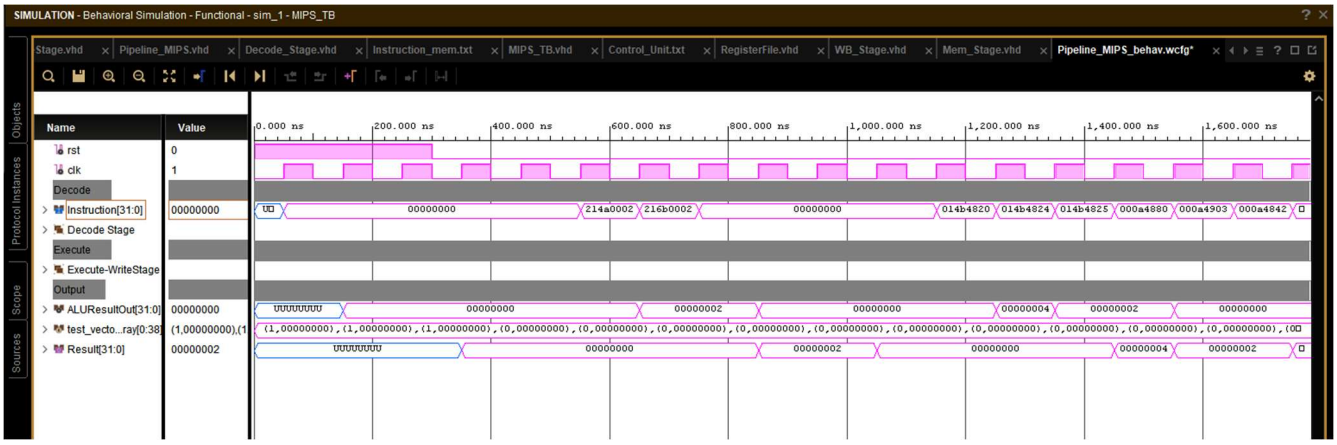

Figure 1: Operations Test Behavioral Simulation(1)
The First half of the waveform for the Behavioral Simulation is shown in Figure 1 in order to make the values visible.

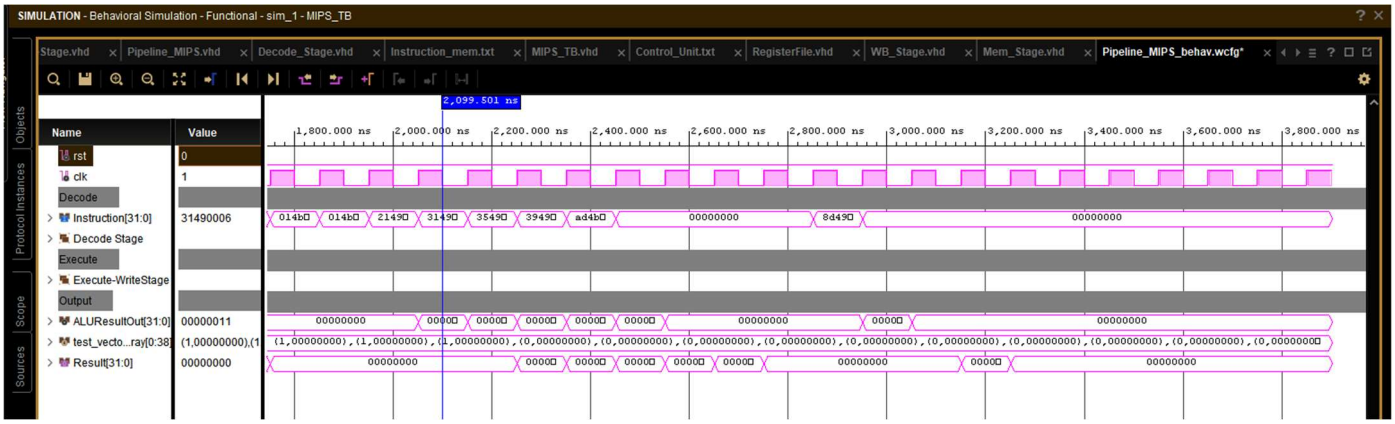

Figure 2: Operations Test Behavioral Simulation(2)

The test bench starts by testing the reset functionality and showing that the instructions do not continue to increment as each cycle passes. After this the registers t2 and t3 are filled with the value of 2 by adding it to their starting values of 0. A waiting time of 4 empty instructions is added to make sure the register values have been written before testing other functionality within the code. It tests adding t2 and t3 followed by an AND and an OR, each with the respective output of 4, 2, and 2. Next it tests each of the shift commands, sll, sra, and srl by shifting them with values of 2, 4, and 1, each time with the ALUResult showing the correct values. The SUB and XOR functions are tested next to finish the register type instructions with the correct outputs showing. The Immediate type instructions are tested in the order of ADDI, ANDI, ORI, and XORI. Each one provided with an immediate value and producing the correct output. SW and LW are tested last with time in between each one to allow for the data to be written before being read.

The second test bench was designed to generate the Fibonacci sequence storing each value. The resulting waveforms from the Behavioral simulation using this instruction set is shown

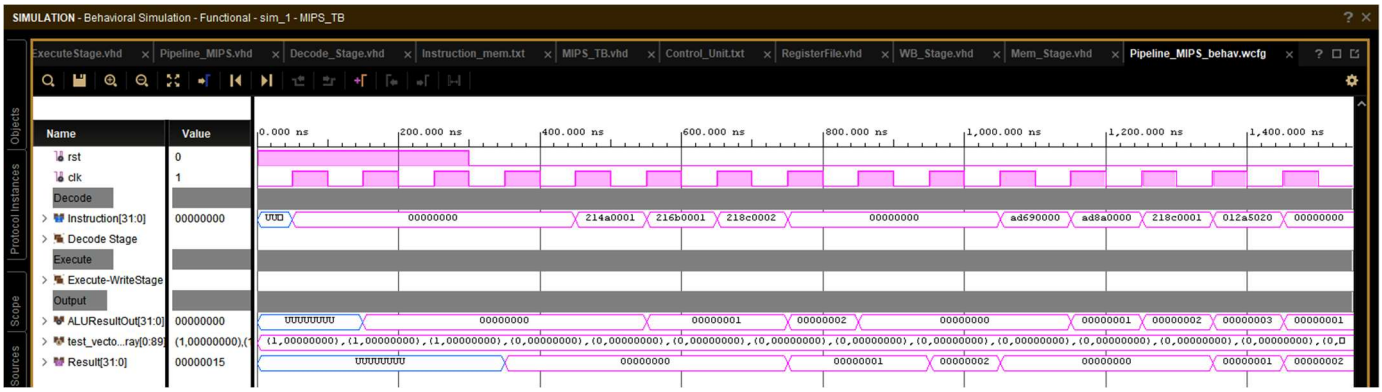

Figure 3: Fibonacci Behavioral Simulation(1)

The waveform was divided into six different parts in order to ensure the values are visible in the report. This is the first part.

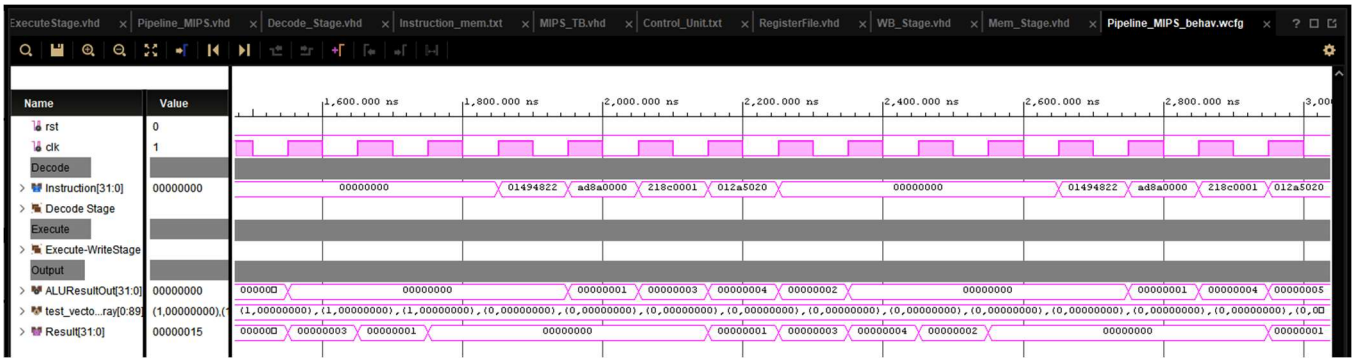

Figure 4: Fibonacci Behavioral Simulation(2)

The waveform was divided into six different parts in order to ensure the values are visible in the report. This is the second part.

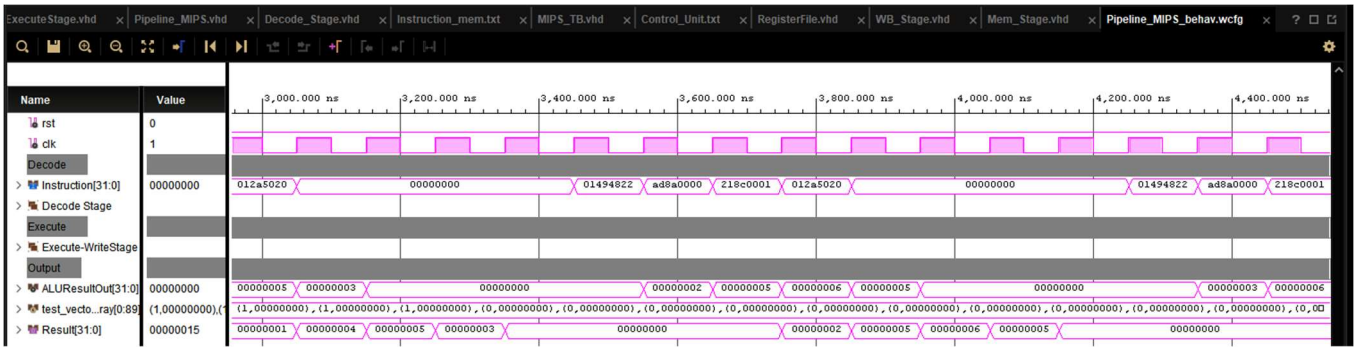

Figure 5: Fibonacci Behavioral Simulation(3)

The waveform was divided into six different parts in order to ensure the values are visible in the report. This is the third part.

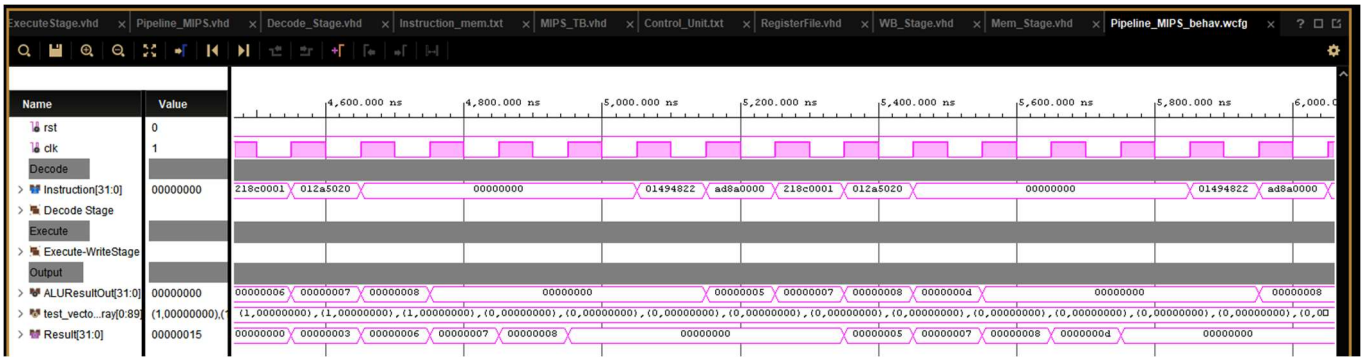

Figure 6: Fibonacci Behavioral Simulation(4)

The waveform was divided into six different parts in order to ensure the values are visible in the report. This is the fourth part.

Figure 7: Fibonacci Behavioral Simulation(5)

The waveform was divided into six different parts in order to ensure the values are visible in the report. This is the fifth part

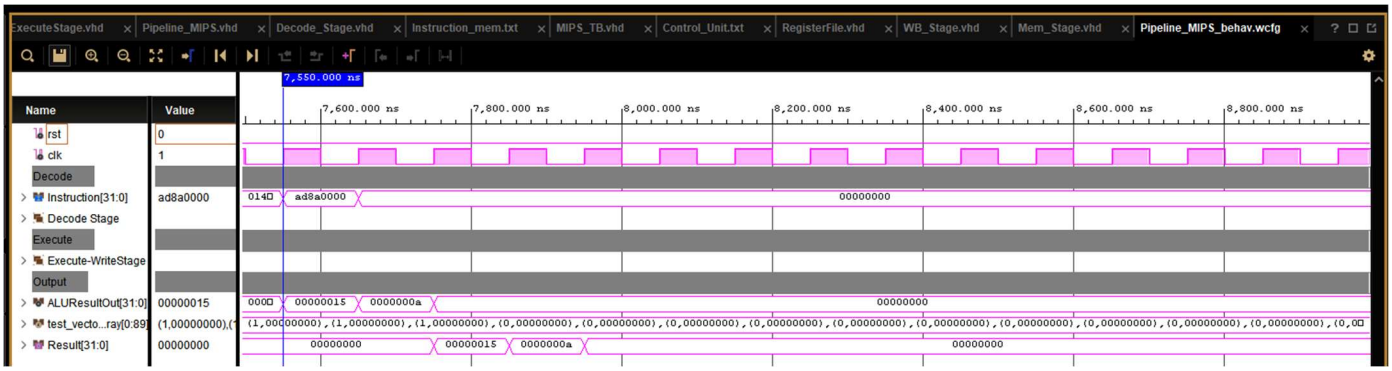

Figure 8: Fibonacci Behavioral Simulation(6)

The waveform was divided into six different parts in order to ensure the values are visible in the report. This is the sixth part. The sequence is shown to have the correct output with the larger of the two number values appearing before each of the 3-cycle timing breaks, and the second largest coming directly after. The numbers in between the two show the location it is being written first followed by the next location to be written. Each of the numbers match the sequence of 0, 1, 1, 2, 3, 5, 8, 13, 21, then 34. This shows that the correct outputs were obtained by the instruction set and they were calculated correctly within the MIPS processor.

Conclusion

The Pipelined MIPS Processor created was implemented successfully to complete the functionality to perform the correct operations with the given instructions, outputting the correct values for each set of instructions. Each of these stages were tested in the Xilinx architecture through the Behavioral simulations with the test benches designed for both stages. The simulations showed that both stages were designed correctly. There were minor issues with the project becoming corrupted resulting in the process started from scratch twice during development. This exercise was successful in both the design of the full MIPS Processor, as well as reinforcing the principles necessary to create it including the VHDL functionality and

Instruction writing in hex. It was also successful in providing more practice in the creation of functional test benches to test the created code.