COEN 4710: VHDL Project 2

4/23/2019

Dr. Ababei

Logan Wedel

Jeremy Horky

Daniel Whipple

I. Introduction

For this project, several components that are crucial to a MIPS 32-bit ISA, including a register bank, an ALU and control unit, a 32-bit adder, a memory unit, and a single cycle control unit, are to be designed and described in VHDL, and then tested thoroughly to ensure functionality and correctness. Each of these components have different required design criteria and functionality to the datapath. The register bank holds 32 registers of 32 bits. The ALU is able to perform the ADD, SUB, AND, OR, SLL, SRL, and SLT operations, and the ALU control unit facilitates this. The 32-bit adder is a simple connection of 32 full adder components and allows two 32-bit numbers to be added or subtracted. The memory unit is a 64x32 SRAM unit that is able to be preloaded with data. Finally, the single cycle control unit reads a 6-bit opcode and generates all necessary control signals to perform the specified operation. Three more simple components will be created, including a logical shifter, a simple multiplexer, and a program counter.

After thorough testing, all the components specified above appear to be working according to the design requirements. From these components, a simple processor will be able to be constructed. There are no bonus requirements completed for this project.

II. Design Process

The focus of how decisions were made in designing the MIPS control path were based on how the main controller could control each component with its signals. Instead of directly implementing the ALU with the 6-bit function and 2-bit ALU op, an ALU control component was implemented to help improve the debugging process and determine the ALU control input for the ALU rather than having the ALU self-determine the action of the ALU. While this may increase the data path run time, it improved the readability and control of the circuit and can later be implemented within the ALU.

The ALU has numerous operations to account for, leading to the design choice of making the core functionality of the ALU a case statement that activated based off changes in the data inputs and ALU_control process, allowing for a much simpler implementation. Important operations of the ALU include add and sub, both which used a 32-bit adder. By setting the second data value as a 2’s compliment, the adder can be used to subtract the two data values. Other important operations include the sll and srl which used a shifter, another new component added to this control path. The shifter determined the srl (101) and sll (100) operation to use depending on the ALU_control value that was already determining which ALU operation to perform. This allowed the shifter to select the correct shift without requiring a new signal from the main controller. While the shifter wasn’t implemented into the ALU, it is fully functional and will be implemented in the full data path.

Since the Instruction and Data memory are very similar components, only one had to be designed. The Instruction memory unit had address input to read and determine the output, but only reading was enabled while writing was disabled. Since no new data is being added, the datra

has a 0 input. This creates a problem in the full data path that this circuit can only run programs that are preloaded, but it will be enough for initial testing. Adding new instructions can be implemented into a more complicated Instruction memory unit.

Another unique choice for this project was adding a new mux and jump signal from the controller path as it made an easier jump operation, rather than branching forward a step through the ALU as it would have been implemented. This increased the complexity of the hardware but decreased the jump operation run time.

III. Design Specifications

Table I: control signals

	ADD	SUB	AND	OR	SLL	SRL	SLT	JR	ADDI	ORI	LUI	LW	SW	BEQ	JAL
Reg_dest	1	1	1	1	1	1	1	X	1	X	0	0	X	X	X
jump	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
branch	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Mem_read	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
Mem_reg	0	0	0	0	0	0	0	X	0	X	1	1	X	X	0
ALU_op	10	10	10	10	10	10	10	00	10	10	10	00	00	01	00
Mem_write	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
ALU_src	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
Req_write	1	1	1	1	1	1	1	0	1	0	1	1	0	0	0

Figure I: Datapath

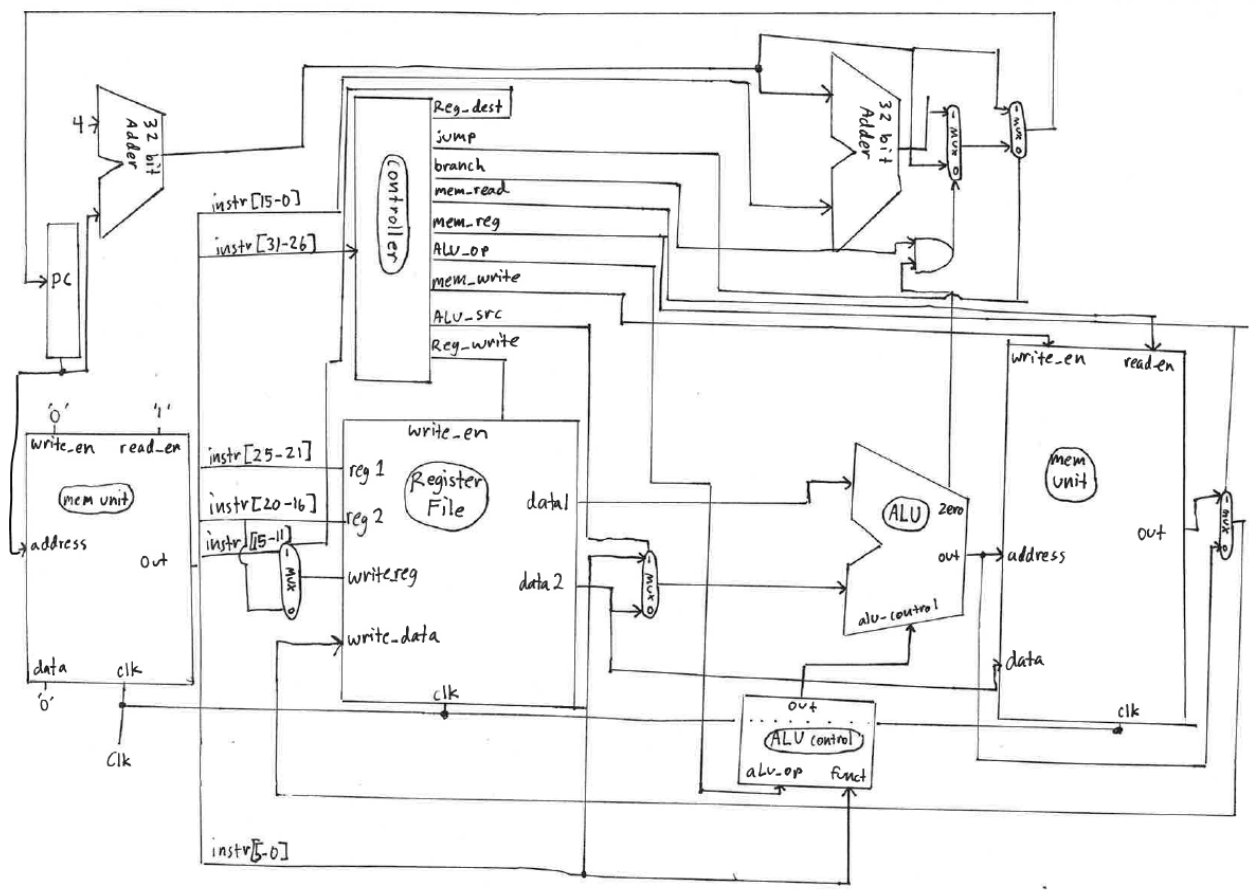


Figure II: Adder Schematic

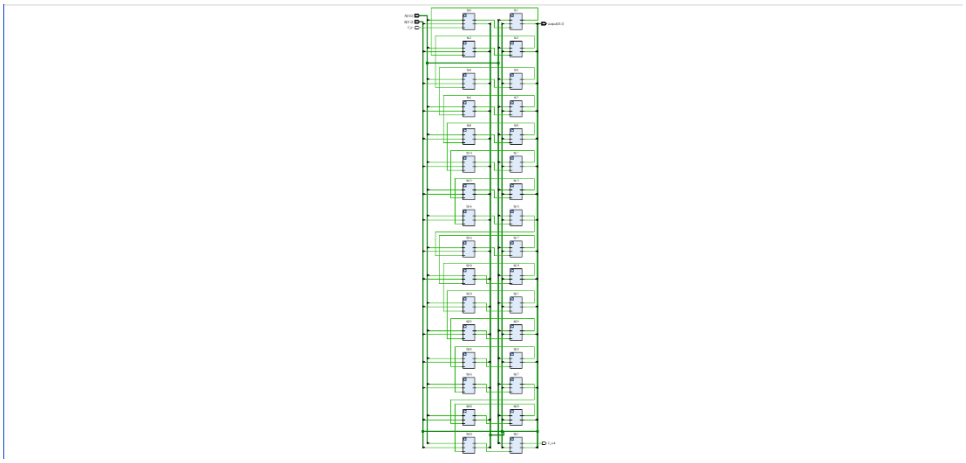


Figure III: ALU Schematic

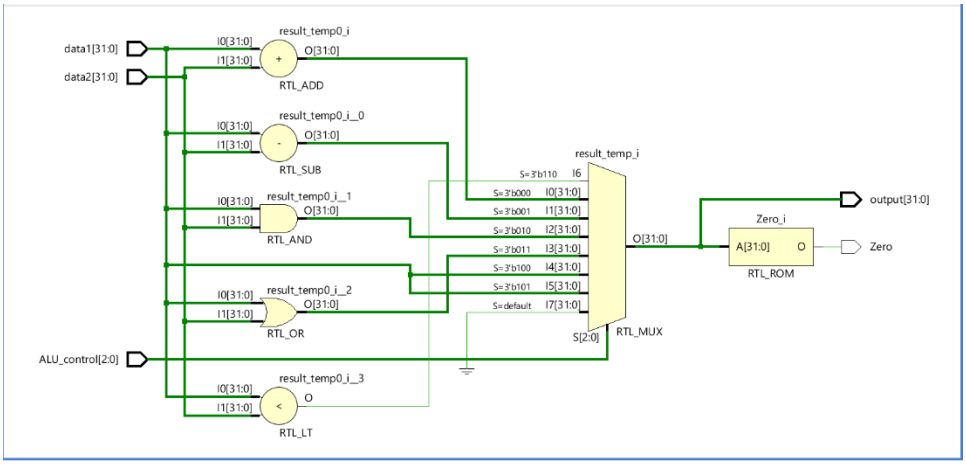


Figure IV: ALU Control Schematic

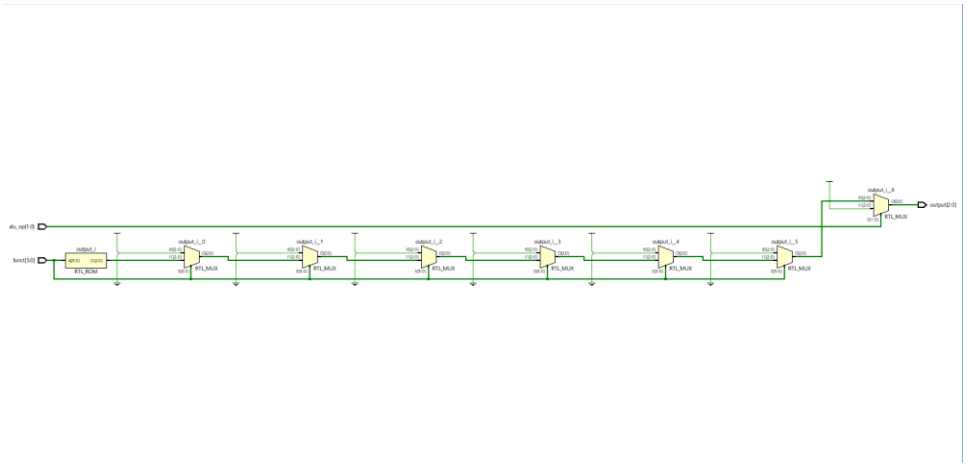


Figure V: Control Unit Schematic

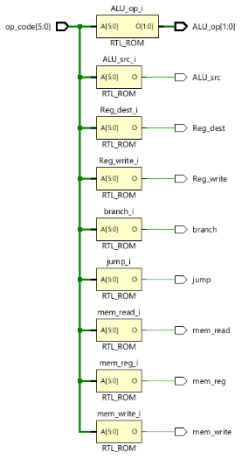


Figure VI: Memory Unit Schematic

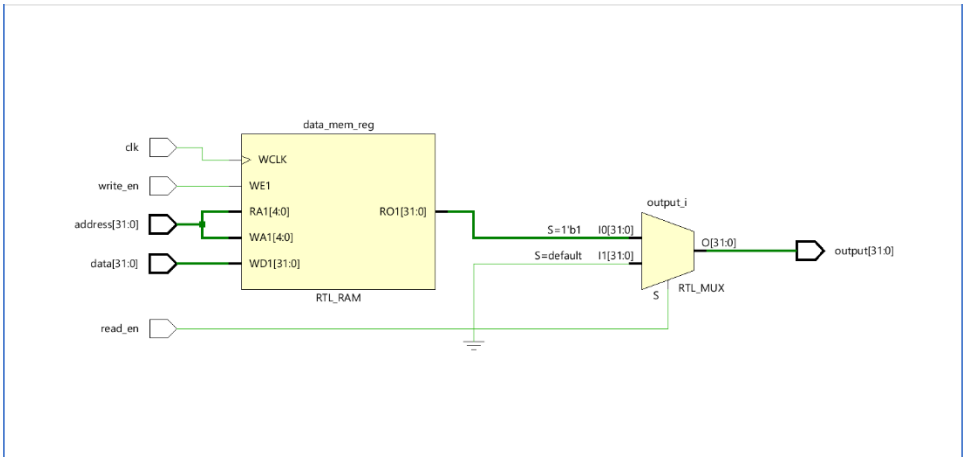


Figure VII: Register File Schematic

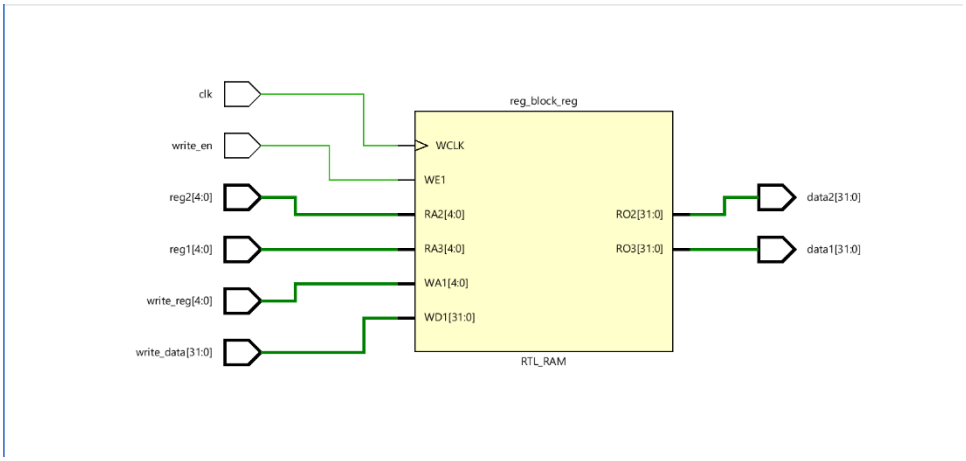
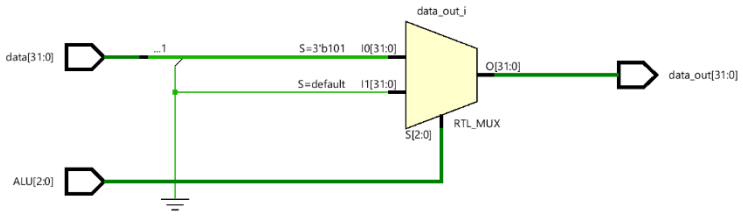
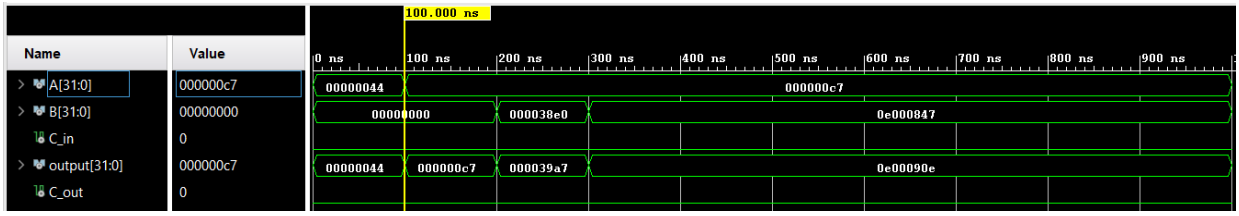


Figure VIII: Shifter Schematic



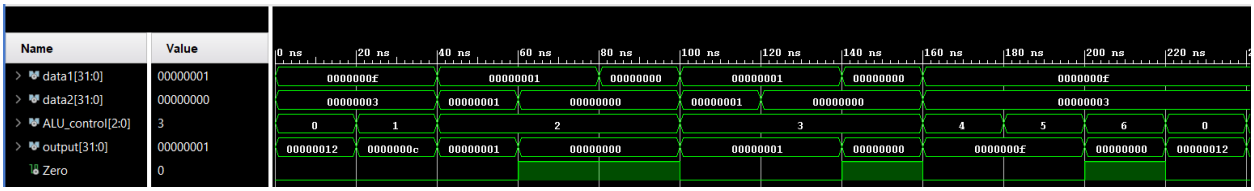
IV. Testing and Verification

Adder:

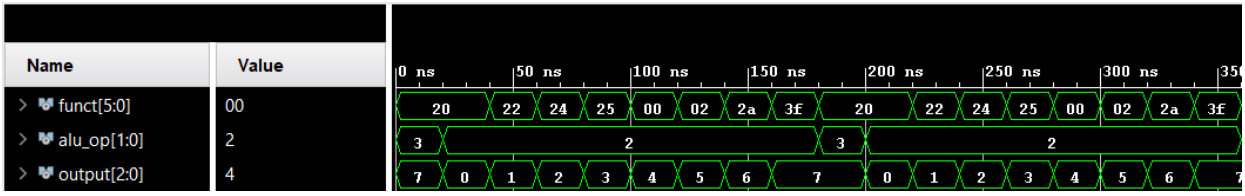


The adder test bench tests a few addition operations. The adder first tested the ‘a' input with ’00000000000000000000000001000100‘ which set the output to hex 44 . Setting ’a’ to ’00000000000000000000000011000111’ appropriately change the output to hex c7. Both ‘a’ values are added to a ’b’ value of zero, resulting in an output that matches the ’a’ values as expected. Then ’b’ is set to ’ 00000000000000000011100011100000’ or 38E0 hex resulting in c7+38E0 which appropriately results in an output of 39A7. Changing ’b’ to ’00001110000000000000100001000111’ or E000847 sets the output correctly to E00090E.

ALU and Control:

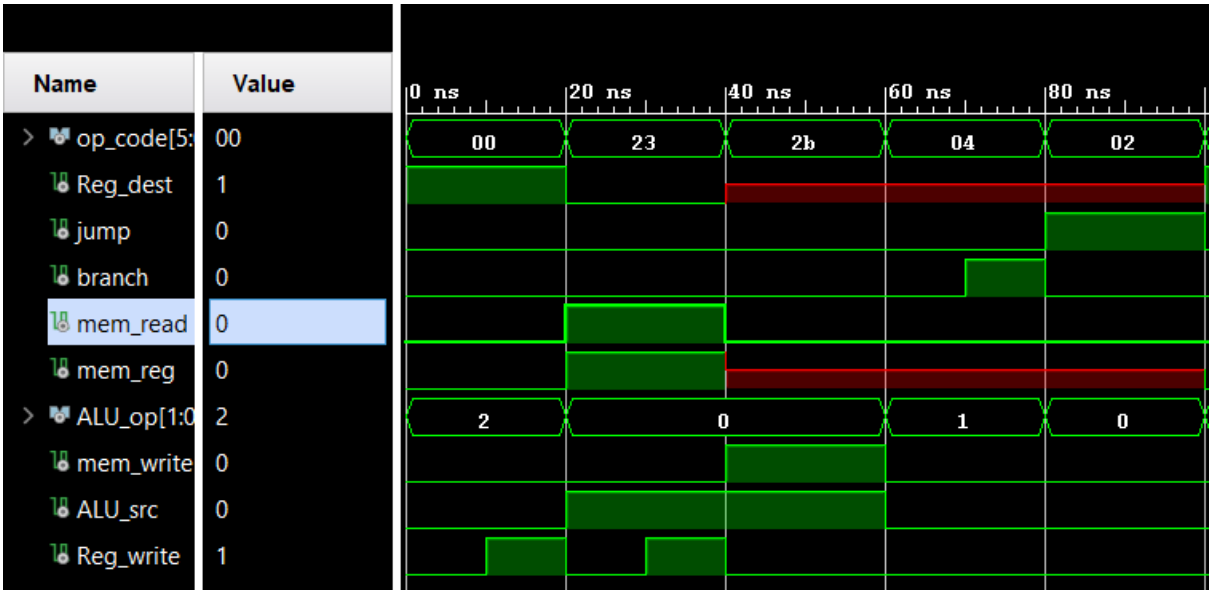


The ALU test bench checks each operation of the ALU and whether it performs the operation appropriately. The ALU tests runs through all of the tests with data1 being ’00000000000000000000000000001111’ or hex F and data2 being ’ 00000000000000000000000000000011’ or hex 3. An ALU_control 0 sets the ALU to perform addition of data1 and data2 resting in an output of hex 12. ALU_control of 1 performs a subtraction resulting in –c. ALU_control of 2 performs an and operation by checking if data1 and data2 are both ‘1’ and outputs ‘1’ if yes and ‘0’ if no. For this test there are three cases: when data1 and data2 are set to ‘1’, when data1 is set to ‘1’ ad data2 is set to ‘0’, and when data1 and data2 are set to ‘0’. ALU_control of 3 performs an or operation by checking if data1 or data2 is set to ‘1’ and outputs ‘1’ if yes and ‘0’ if no. For this test there are three cases: when data1 and data2 are set to ‘1’, when data1 is set to ‘1’ ad data2 is set to ‘0’, and when data1 and data2 are set to ‘0’. Since the shifter is not currently implemented into the ALU, there is no change in output for AL control of 4 and 5 (sll and srl). Check the shifter tests to evaluate that functionality. The final ALU_control 6 checks the slt which F<3 should provide false resulting in an output of 0. The Zero output should also be triggered as 1.



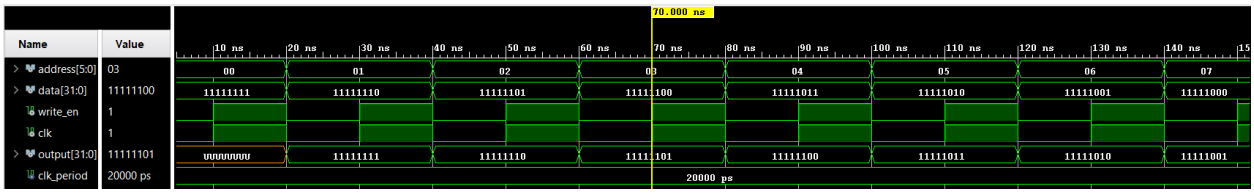
The ALU_control testbench tests if the output signals will indicate the correct ALU operation. First the funct is set to 20 hex for add but the alu_op is set to '11' which should give a nop output. An ALU nop output is 7 hex which is shown. The alu_op is changed to '10' for R type operations and gives a 0 output which indicates add. Setting funct to hex 22 outputs sub which results in the sub output 1. Setting funct to hex 24 outputs the and output value of 2. Setting funct to hex 25 outputs 3 which is the or operation. Setting funct hex 00 outputs 4 which is the sll operation. Setting funct hex 02 outputs 5 which is the srl operation. Setting funct hex 2a outputs 6 which is the slt operation. Setting funct hex 3f outputs 7 again which is appropriate as that is not a real funct input.

Control Unit:



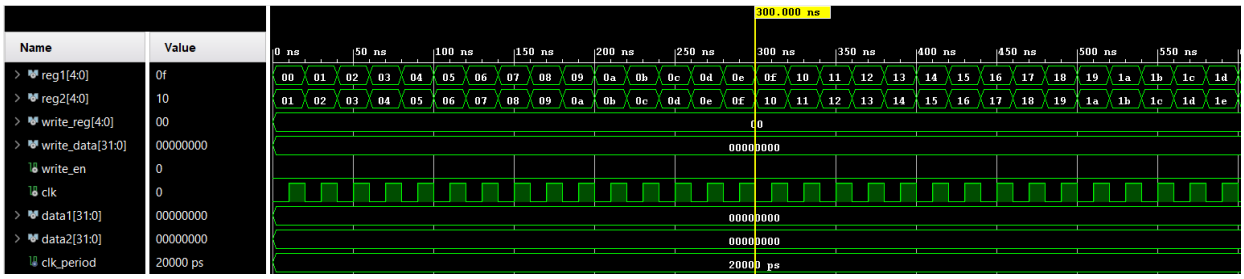
The control unit test bench tests for whether the inputted op_code results in the correct setting of all signals in order to maintain a proper data path. The first op_code hex 00 indicates the R-type operations and triggers Reg_dest, ALU_op vector of '10' and Reg_write after a 10 ns delay. The lw input of hex 23 triggers mem_read, mem_reg, ALU_op '00', ALU_src and Reg_write after a 10 ns delay. The sw input of hex 2b triggers a don't care Reg_dest and mem_reg, ALU_op '00', and active mem_write and ALU_src. The beq input triggers a don't care reg_dest and mem_reg, ALU_op '01' and an active branch with a 10 ns delay. The jump input triggers a don't care Reg_dest and mem_reg, ALU_op '00' and active jump.

Memory Unit:



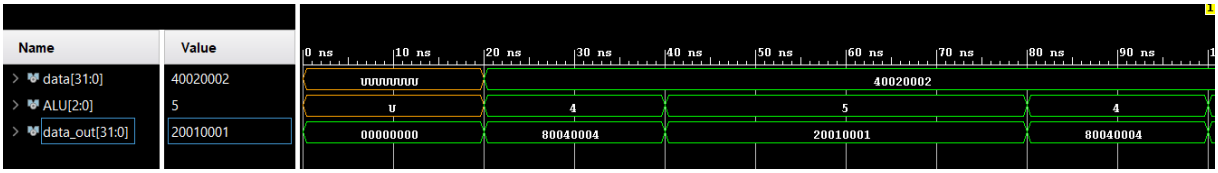
The memory unit test bench tests whether or not data can be written to the memory registers by incrementing the memory address value from ‘0’ to ‘7’ and writing the inputted data to that slot when both write_en is high and the clock is on a rising edge. To ensure the functionality of the write_en signal, the test alternates between high and low every 10 ns, and every 20 ns the memory write address increments and the data being written to the memory changes.

Register File:



The Register File test bench tests that the registers are only written too on the rising edge of a clock and increment through a changing loop of values to confirm the register values properly change. The reg1 values change from 0 to 30 at each clock tick (loop iterates after 20 ns) and the reg2 changes from 1 to 31, always a value greater than reg1. This has shown to be true as every value of reg2 is greater than each value of reg1 at that clock tick. However, the event is triggered at a falling clock edge, not a rising edge.

Shifter:



The shifter test bench checks if the sll and srl commands are properly chosen from the signal given by the ALU and that the operation is properly performed. The test first waits 20 ns to ensure that the ALU value and data values are changing the data_out. Since there is no proper ALU input the data_out gives the error zero valued output. Then ALU is set to hex 4 to check for the sll to shift the data input ’01000000000000100000000000000010’ left. 40020002 is doubled to 80040004 as expected by a shift to the left. The ALU set to 5 shifts the data input to the right which halves 40020002 to 20010001 as expected.

V. Performance Analysis

Table II: Approximate delay per component

Component	CPU Time (s)	Elapsed Time (s)	Approximate Delay
Adder	00:00:07	00:00:11	4 ns
ALU	00:00:09	00:00:15	6 ns
ALU Control	00:00:19	00:00:24	6 ns
Control Unit	00:00:25	00:00:25	0 ns
Memory Unit	00:00:08	00:00:14	6 ns
Register File	00:00:12	00:00:18	6 ns
Shifter	00:00:35	00:00:28	7 ns

The ALU operation will dramatically increase in time when connected in the full data path as it will either be running the Shifter, Adder, or other expressions. Since the ALU has the most intensive job, it is expected that it will be taking longer than other components and likely to be part of the critical data path. It is surprising how long the ALU control takes as compared the control unit as the ALU control only searches for a funct value and returns an output after reaching a match. It is even more important to speed this process up as it precedes the ALU which has already been explained to be a slow process.

VI. Discussion and Conclusion

Upon completing this project, all of the components outlined above have been described and tested. However, there are some aspects that still need to be visited. First, the ALU is not completely functional. Every part of it has been described, but it is missing certain functionalities. The shift logical left and shift logical right functionalities exist in a separate shifter component that was tested individually, however it is disconnected from the ALU at the moment for testing reasons. Both components work on their own but are not integrated yet. Also in the ALU, there is a minor issue with subtraction. For binary subtraction, the typical method is to subtract the two's complement signed form of the subtrahend. However, this caused an error that was not able to be debugged in time. Like the shifters, this functionality works perfectly in the separate adder component, however it has not been integrated into the ALU just yet. Another concern was implementing edge detection that triggered when a rising clock edge was detected. Only methods detecting a falling edge worked so that was implemented rather than rising edge detection. Every other part that was designed works as they should, and the ALU works other than those few aforementioned integration issues. Everything is perfectly primed to be connected together to form the full datapath detailed in figure 1.

Overall, this project helped to demonstrate the various components that are part of the 32-

bit MIPS ISA and has helped to create the basics of a simple datapath. From here, it will not be difficult to simulate the datapath, as all that needs to be done is connecting the outputs of some of the designed components to the inputs of other components. Most of the components that were designed for this project work perfectly, however there is still room for improvement.