

CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

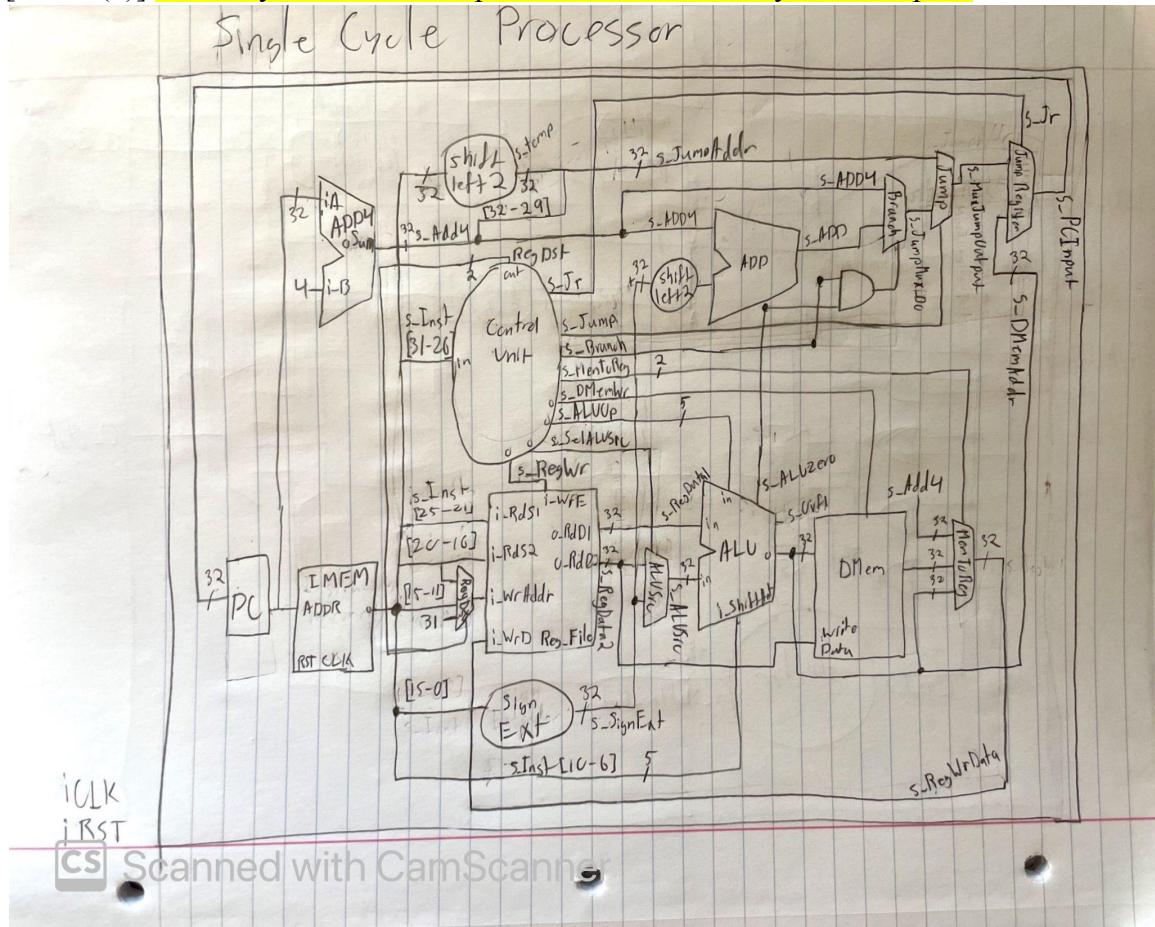
Team Members: Lex Somers

Remington Greatline

Project Teams Group #:2_01

Refer to the highlighted language in the project 1 instruction for the context of the following questions.

[Part 1 (d)] Include your final MIPS processor schematic in your lab report.

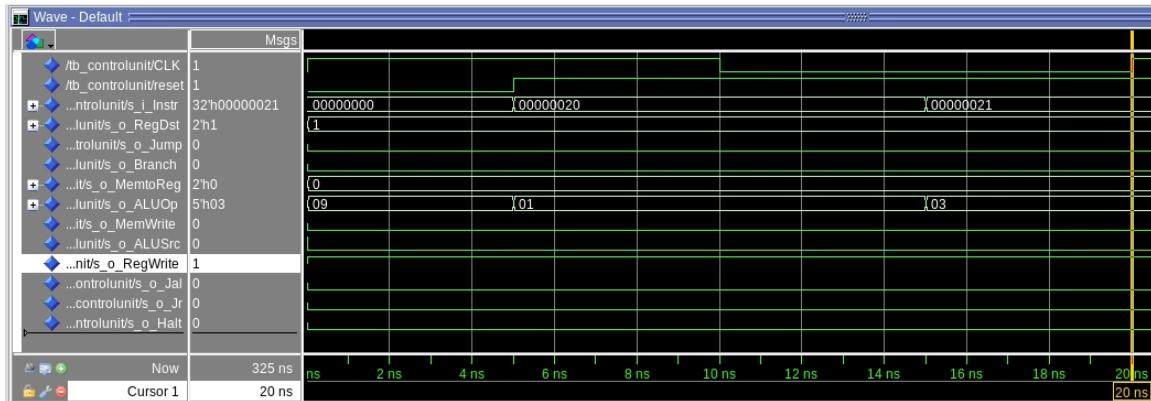


[Part 2 (a.i)] Create a spreadsheet detailing the list of M instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a

separate column for each binary bit. Inside this spreadsheet, create a new column for the N control signals needed by your datapath implementation. The end result should be an $N \times M$ table where each row corresponds to the output of the control logic module for a given instruction.

Instruction	Opcode (Binary)	Func (Binary)	j[16:0]	j[15:0]	MemoReg	MemWrite (W/E Reg)	RegWrite (W/E Reg)	RegSel	Branch (PC+4)	SignExt (extended sign extension on 16 to 32 bit extended)	[Jump]	ALUOp	ALUOp	
R-TYPE														
add	"000000"	"100000"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [does write back to a register]	1 [does read or is destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"00001"	[uses adder/muxor output]
addi	"000001"	"100001"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [does write back to a register]	0 [does not read or is destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"00011"	[uses adder/muxor output]
andi	"001000"	"101000"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"00100"	[uses and of results]
nor	"000000"	"000011"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"000010"	[uses nor of inputs]
xor	"000000"	"000011"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"000010"	[uses xor of inputs]
or	"000000"	"000011"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"000010"	[uses or of inputs]
slt	"000000"	"101010"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"01100"	[uses less than result]
sll	"000000"	"000000"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"01001"	[uses shift result]
srl	"000000"	"000010"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"010010"	[uses shifter result]
sub	"000000"	"100001"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"000000"	[uses subtractor output]
subu	"000000"	"100011"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	1 [uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	0 [uses register as second input]	"01000"	[uses adder/muxor output]
jr	"000000"	"001000"	1	0	0 [does not matter]	0 [does NOT write to memory]	0 [does not matter]	0 [does not matter]	0 [no jump]	1 [does not matter but choose one because more sign extension]	1 [jump]	0 [uses register as second input]	"000000"	[no ALUOp]
L-TYPE														
addi	"001000"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [addiu uses it as destination register rather than rd]	0 [use PC+4 no branch]	1 [addiu sign extended]	0 [no jump]	1 [uses immediate as second input]	"00001"	[uses adder/muxor output]
addiu	"000101"	"-----"	0	0	0 [addiu does NOT read from memory]	0 [addiu does NOT write to memory]	1 [writes back to a register]	0 [addiu uses it as destination register rather than rd]	0 [use PC+4 no branch]	1 [addiu sign extended]	0 [no jump]	1 [uses immediate as second input]	"00011"	[uses adder/muxor output]
andi	"001010"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rd as destination register rather than rd]	0 [use PC+4 no branch]	0 [zero extended]	0 [no jump]	1 [uses immediate as second input]	"00100"	[uses and of results]
ori	"000110"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rd as destination register rather than rd]	0 [use PC+4 no branch]	0 [zero extended]	0 [no jump]	1 [uses immediate as second input]	"001010"	[uses or of inputs]
lui	"000101"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rd as destination register rather than rd]	0 [use PC+4 no branch]	0 [zero extended]	0 [no jump]	1 [uses immediate as second input]	"01010"	[uses immediate of inputs]
lui	"000111"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rd as destination register rather than rd]	0 [use PC+4 no branch]	1 [sign extended]	0 [no jump]	1 [uses immediate as second input]	"01100"	[uses less than result]
lui	"000111"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rd as destination register rather than rd]	0 [use PC+4 no branch]	1 [sign extended]	0 [no jump]	1 [uses immediate as second input]	"01101"	[uses shifter result]
beq	"000000"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to memory]	0 [does not matter]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	1 [uses immediate as second input]	"000001"	[uses zero, OUT process]
bne	"000001"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to memory]	0 [does not matter]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	1 [uses immediate as second input]	"000010"	[uses zero, OUT process]
bgte	"001010"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to memory]	0 [does not matter]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	1 [uses immediate as second input]	"000000"	[uses immediate of inputs]
bgz	"000011"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to memory]	0 [does not matter]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	1 [uses immediate as second input]	"000000"	[uses immediate of inputs]
blez	"000010"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to memory]	0 [does not matter]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	1 [uses immediate as second input]	"000010"	[uses zero, OUT process]
bleq	"000000"	"-----"	0	0	0	0	0	0	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	1 [uses immediate as second input]	"000010"	[uses zero, OUT process]
bltz	"010101"	"-----"	0	0	0	0	0	0	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	1 [uses immediate as second input]	"000000"	[uses immediate of inputs]
blge	"010100"	"-----"	0	0	0	0	0	0	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extension]	0 [no jump]	1 [uses immediate as second input]	"000000"	[uses immediate of inputs]
lw	"000011"	"-----"	0	0	1 [reads from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rd as destination register rather than rd]	0 [use PC+4 no branch]	1 [sign extended]	0 [no jump]	0 [uses immediate as second input]	"000001"	[uses address/muxor output]
sw	"000101"	"-----"	0	0	0 [does not matter]	1 [writes to memory]	0 [does NOT write to a register]	0 [does not matter]	0 [use PC+4 no branch]	1 [sign extended]	0 [no jump]	0 [uses immediate as second input]	"000001"	[uses address/muxor output]
J-TYPE														
j	"000010"	"-----"	0	0	0 [does not matter]	0 [does not matter]	0 [does not matter]	0 [does not matter]	0 [does not matter]	0 [does not matter but choose one because more sign extension]	1 [jump]	0 [does not matter]	"000000"	[does not matter]
jal	"000011"	"-----"	0	1	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [jump]	0 [does not matter]	0 [does not matter]	0 [does not matter but choose one because more sign extension]	1 [jump]	0 [does not matter]	"000000"	[does not matter]
halt	"010100"	"-----"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does not matter]	0 [does not matter]	0 [does not matter]	0 [does not matter but choose one because more sign extension]	0 [no jump]	0 [does not matter]	"000000"	[does not matter]

[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually and show that your output matches the expected control signals from problem 1(a).

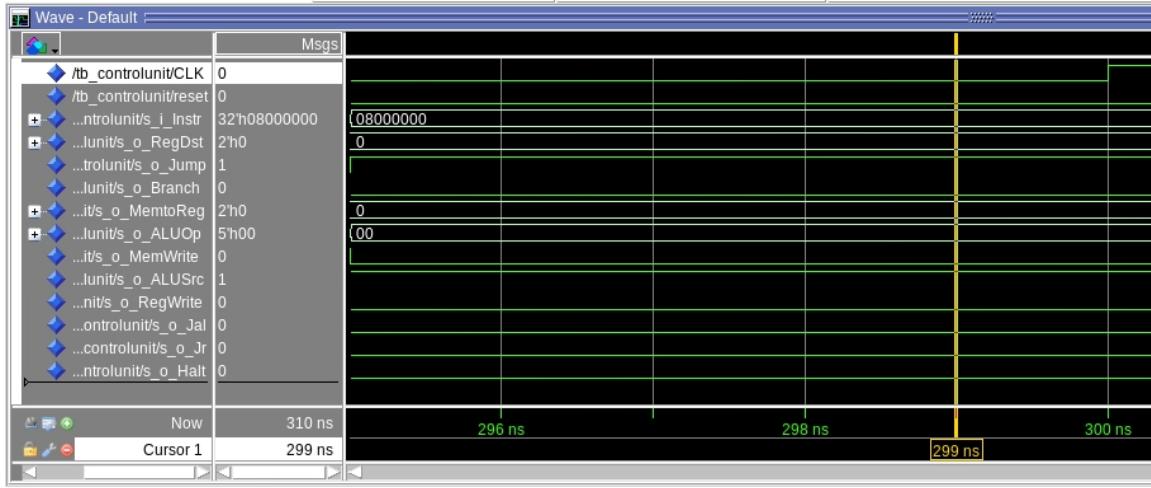


With an instruction of 0x0, the control unit determines that the desired instruction is sll and therefore sets the ALUOp code to 0x9. The following instruction's value is 0x20 which translates opcode 0x1 which corresponds to an addi instruction. Accompanying signals such as o_Jump and o_Branch are set to the appropriate values for these two instructions, that being zero. Note that the “instruction” of our control unit consists of the Opcode concatenated with the function code for ease of determining the appropriate instruction.

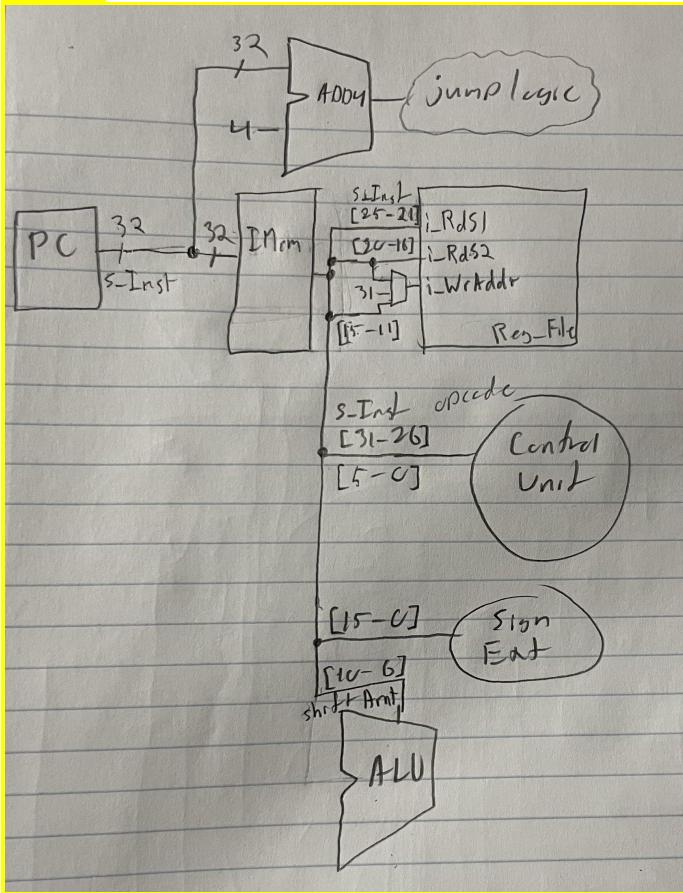
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

Control flow possibilities include jump and branch instructions such as j and beq. Our control unit uses select-when statements that analyze parts of the instruction to determine

control signals for o_Branch, o_Jump, and if necessary o_ALUOp. For example, the jump instruction will set the o_ALUSrc mux to 1, o_Jump to 1, and set the remaining outputs to 0.

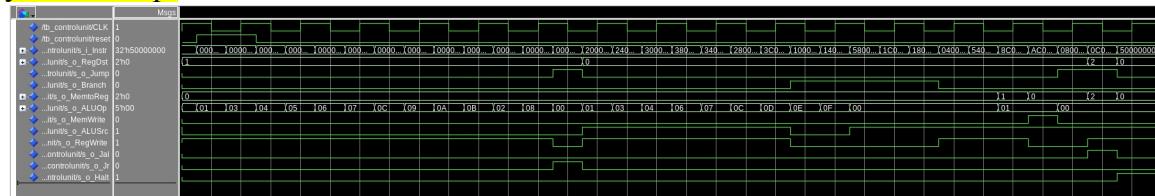


[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?



[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use QuestaSim to test your design thoroughly to make sure it is working as expected. Describe how the

execution of the control flow possibilities corresponds to the QuestaSim waveforms in your writeup.



When fed an Opcode-Function instruction, the control unit analyzes each portion of the instruction and selects each output individually based upon the instruction. This is done with with-select statements and since each output is associated with its own with-select statement, these instructions must consider all outcomes which makes the select when statement appealing to use and fairly straightforward to implement without the need of process statements.

[Part 2 (c.i.1)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts.
Why does MIPS not have a `sla` instruction?

`srl` or `sll` are both logical shifts, meaning, they do not care about the sign of the number they are shifting. `sra` on the other hand does care about the sign of the number it is shifting. So, in `srl`, the bits added from the shift will be 0s. In `sra`, the bits added will ones when bit 31 is a one and zeros when bit 31 is a zero.

The reason there is not a `sla` instruction is because bit 0 does not affect the sign of the number, so shifting arithmetically would not pose any help other than to add a value to the number you're shifting.

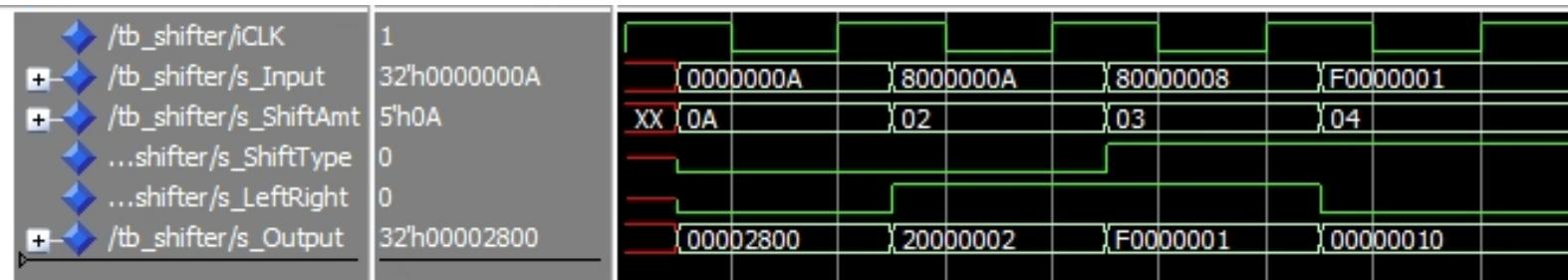
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

In our VHDL code, we used a process statement with if else statements to select what type of shift to perform based on a shift type and shift direction input to the shifter. Once the type of shift and the direction of the shift was selected via the if statements, we then used the built in `shift_right` and `shift_left` functions, specifying a logical or arithmetic shift by using the built in `signed()` function. The amount the value was shifted was specified by the shift amount input to the shifter.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

Our barrel shifter is implemented in a way that it already supports left shifting operations. It can technically perform `sll`, `srl`, `sla`, and `sra`.

[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.



Above shows the output from the shifter test bench. `s_ShiftType` chooses between logical (0) and arithmetic (1) shifts, `s_LeftRight` chooses between a left (0) and right (1) shift, and `s_ShiftAmt` controls the shift amount.

As you can see, when `sll` is performed (Cycle 1) the value `x"A` is shifted left by `x"A` with zeros filling in the lower bits. Likewise, when `srl` is performed (Cycle 2) the input value is shifted left by `x"2` it fills in the upper bits with zeros, despite the most significant bit being a "1".

The next two cycles, `sra` (Cycle 3) and `sla` (Cycle 4), are executed correctly as well. When executing `sra`, it fills in the most significant bits with ones, and when executing `sla`, it shifts the correct amount and fills the bottom bits with zeros, as expected.

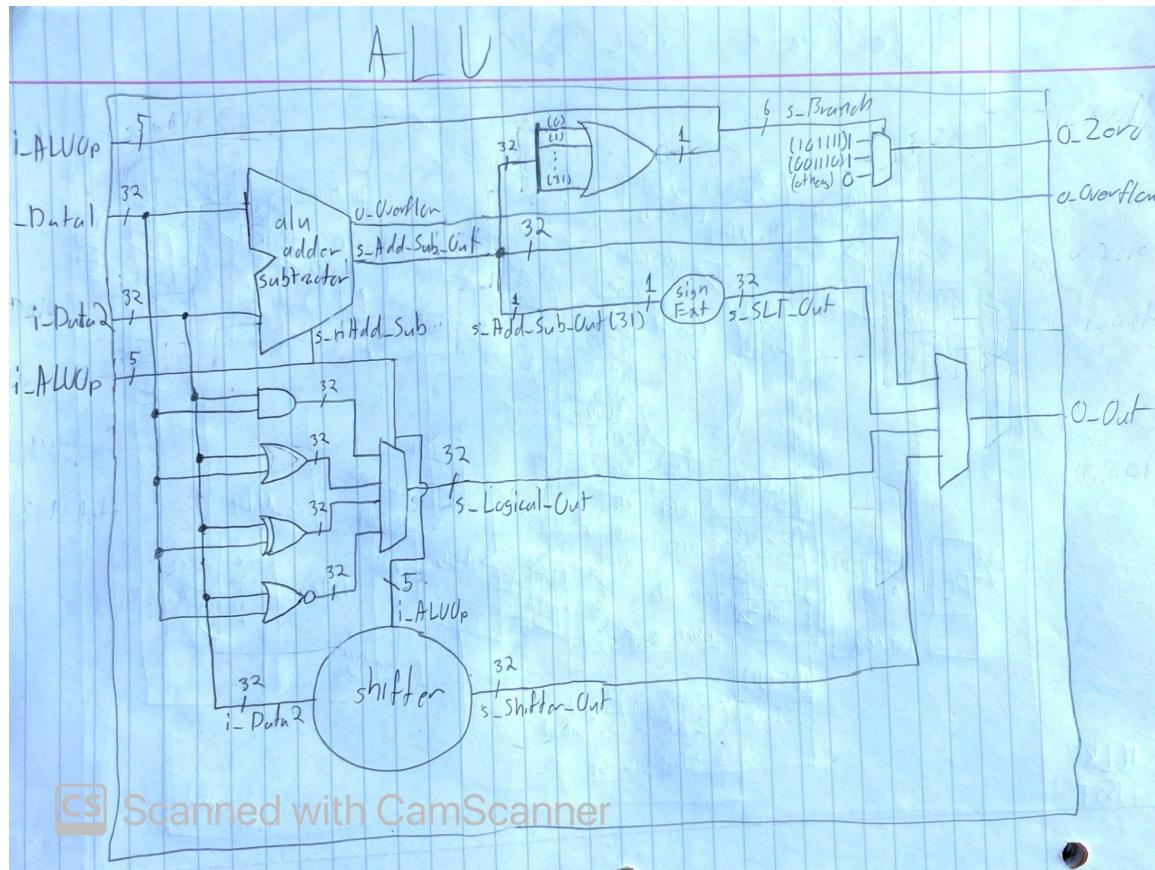
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

When choosing how to design the shifter, we chose to use a process statement and use the built-in VHDL `shift_right()`, `shift_left()`, `signed()`, and `unsigned()` functions. This made it very easy to implement the shifter because we were able to merely choose what operation to do using a few if statements with `i_ShiftType` and `i_LeftRight` as the conditionals. Although process statements were not recommended, when testing it, it worked just fine, so we proceeded with using it.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

As explained above, the testbench executes the `sll`, `srl`, `sra`, and `sla` operations in order. Each operation corresponds to a new clock cycle, with one full clock cycle being 20ns long. Input values are assigned 1/4th of the way into the full clock cycle for waveform clarity.

[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?



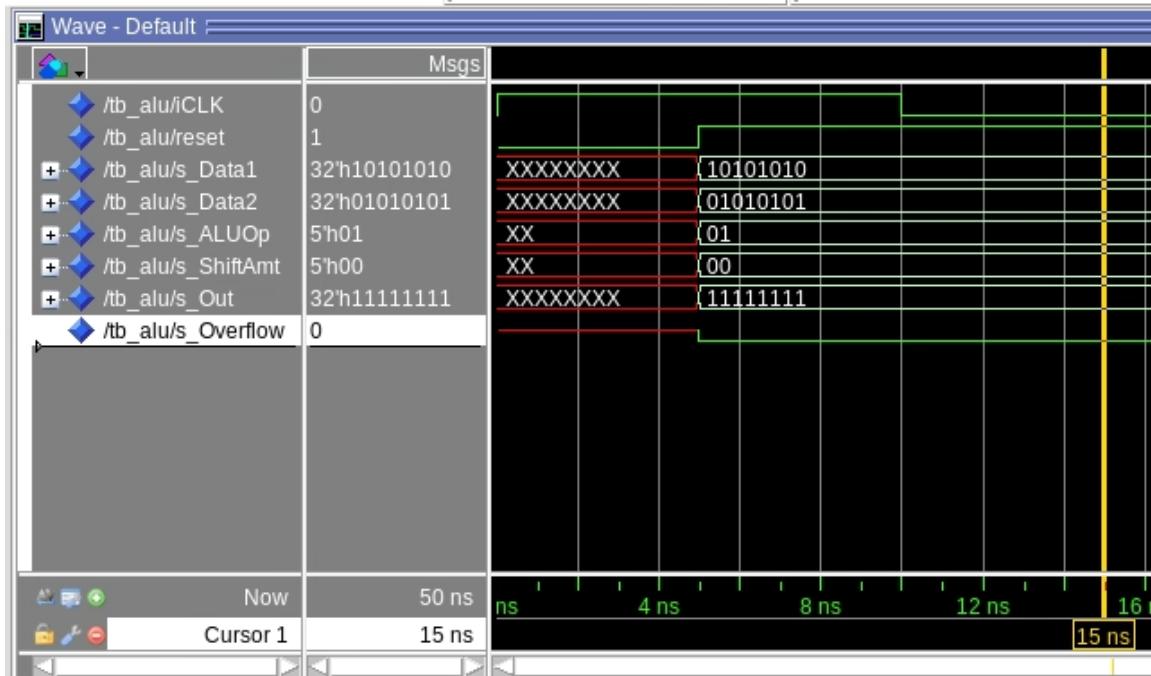
Overflow is calculated by XORing bit 31 of the alu_addersubtractor with the final carry out bit of the alu_addersubtractor output.

Zero is calculated by checking if the alu_addersubtractor output is all zeros using a with select statement.

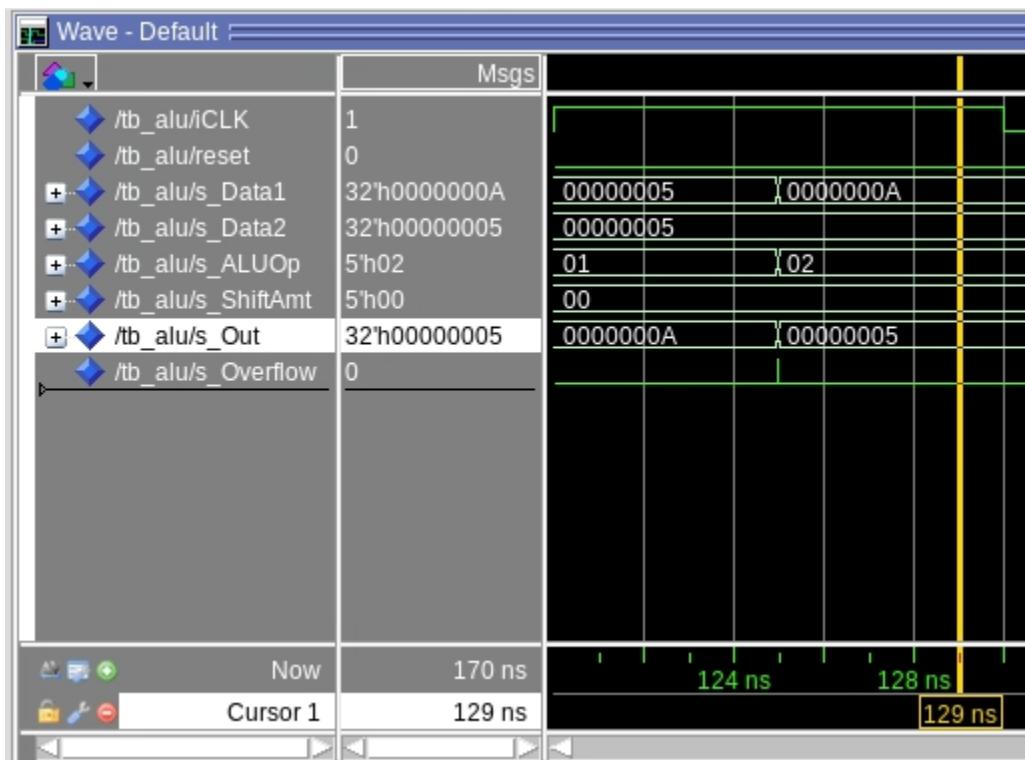
Set on less than is implemented by checking if bit 31 of the alu_addersubtractor output is a 1 (aka if the result of input A minus input B was negative) using a with select statement. If bit 31 was 1, then it sets the output to 1.

[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

The basic design philosophy of the ALU is similar to the Control unit but rather than using an Op-Function code signal to determine its outputs, the ALU primarily relies on the ALUOp value to determine what it will do.

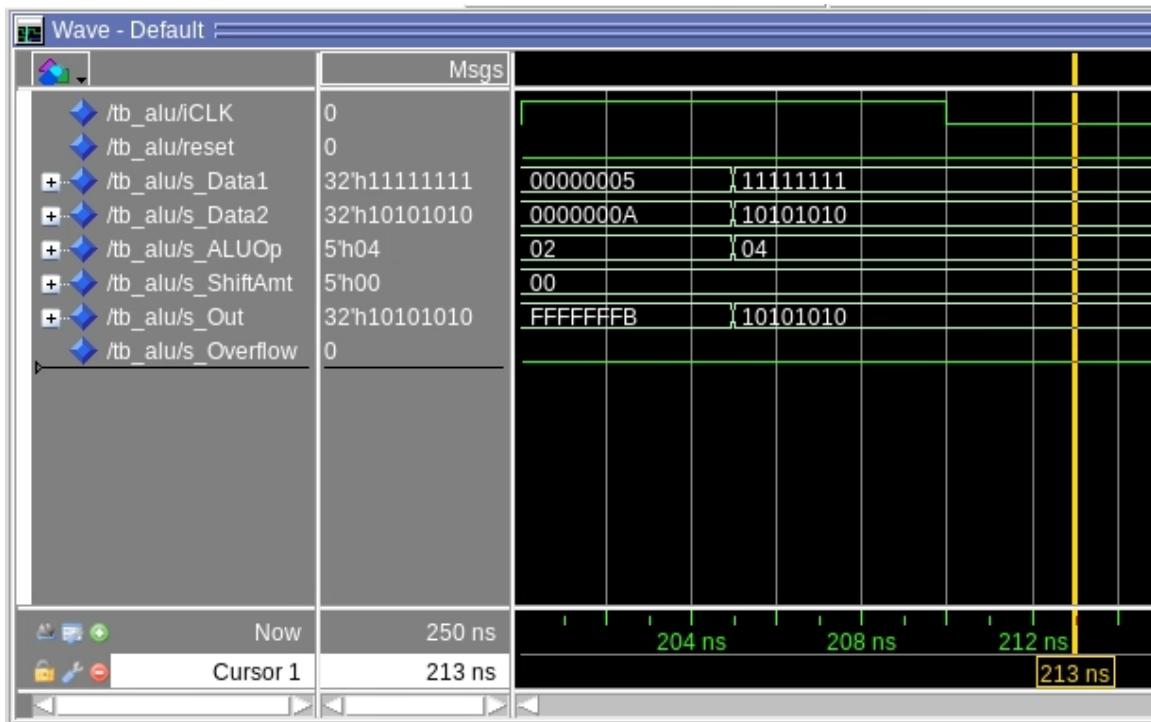


With an ALUOp of 0x01, the ALU determines that the instruction is to add the data inputs together. The ShiftAmt signal is not required for this specific instruction to function. The correct output of an add instruction should be 11111111.

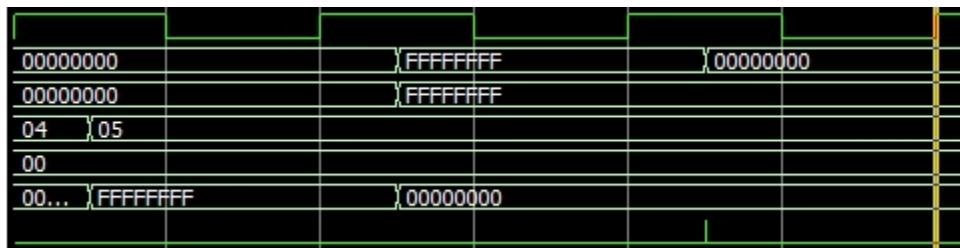


With an ALUOp of 0x02, the ALU determines that the instruction is to subtract data2 from data1. Again, the shift amount is not relevant for this instruction to work.

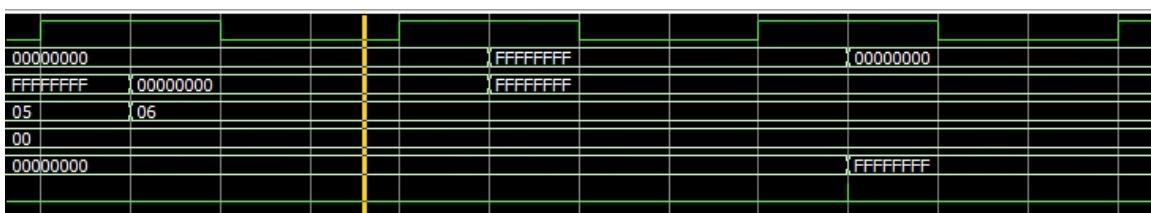
[about:blank#blocked](#)



With an ALUOp of 0x04, the ALU determines that the instruction is to ANDi the values of data1 and data2, the ALU's implementation uses a behavioral AND command to determine the output. Note that the process is similar for an AND instruction but is assumed that one of the data inputs has an immediate value and the other from a register.

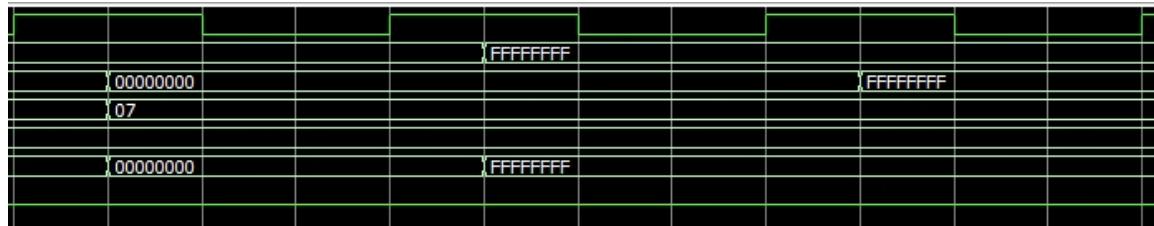


With an ALUOp of 0x05, the ALU determines that the instruction is to NOR the values of data 1 and data 2, the ALU's implementation uses a behavioral NOR command to determine the output.



With an ALUOp of 0x06, the ALU determines that the instruction is to XOR the values of data1 and data2, the ALU's implementation uses a behavioral XOR command to

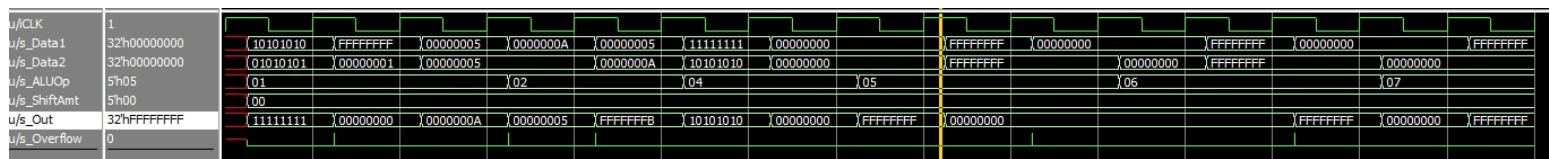
determine the output. Note that the process is similar for an XORi instruction but is assumed that one of the data inputs has an immediate value and the other from a register.



With an ALUOp of 0x07, the ALU determines that the instruction is to OR the values of data1 and data2, the ALU's implementation uses a behavioral OR command to determine the output. Note that the process is similar for an ORi instruction but is assumed that one of the data inputs has an immediate value and the other from a register.

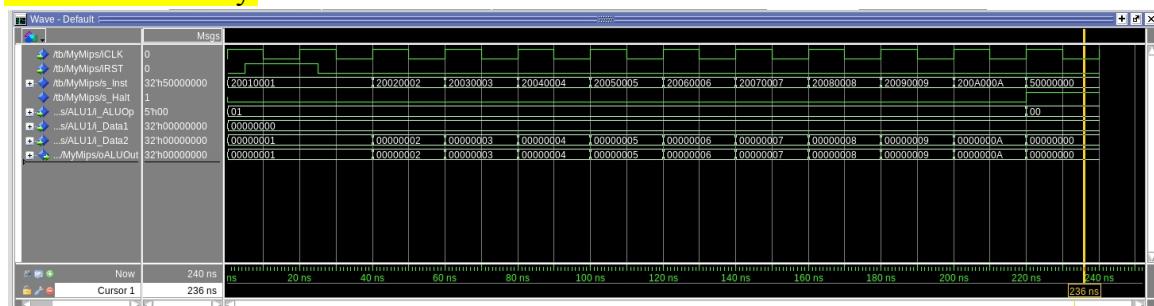
[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

Our tests consist of the following instructions: add, addi, sub, subi, and, andi, nor, xor, xori, or, ori



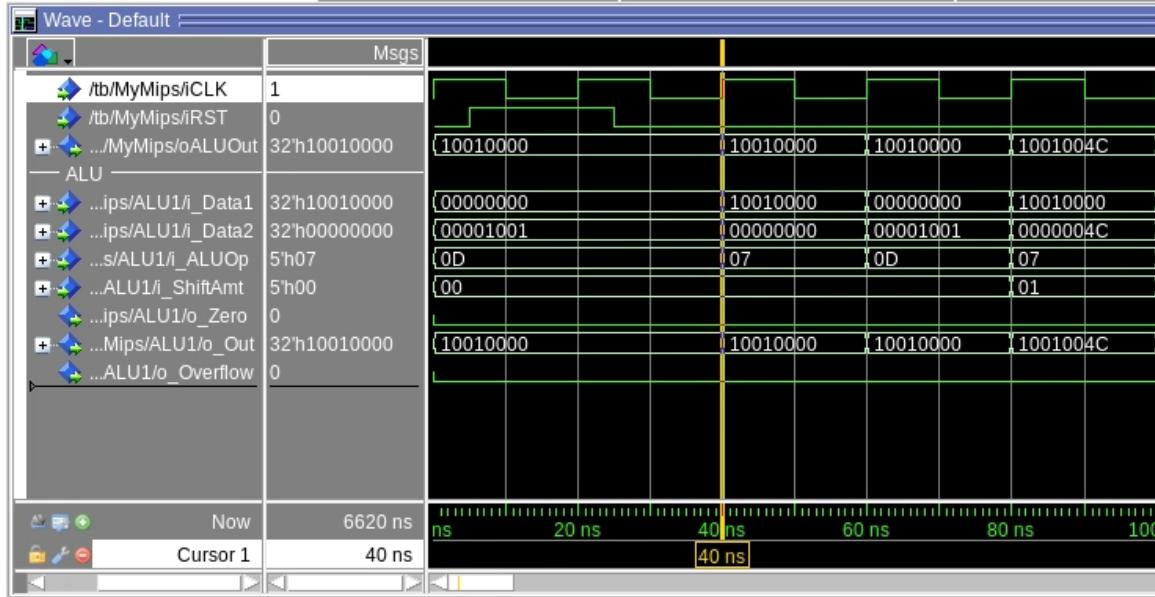
These tests demonstrate that correct values can be derived from the inputs of the ALU given these instructions. The instructions called are both R and I type instructions and demonstrates that the selectc-when logic blocks are not conflicting with each other and work with varying instruction sets.

[Part 3] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

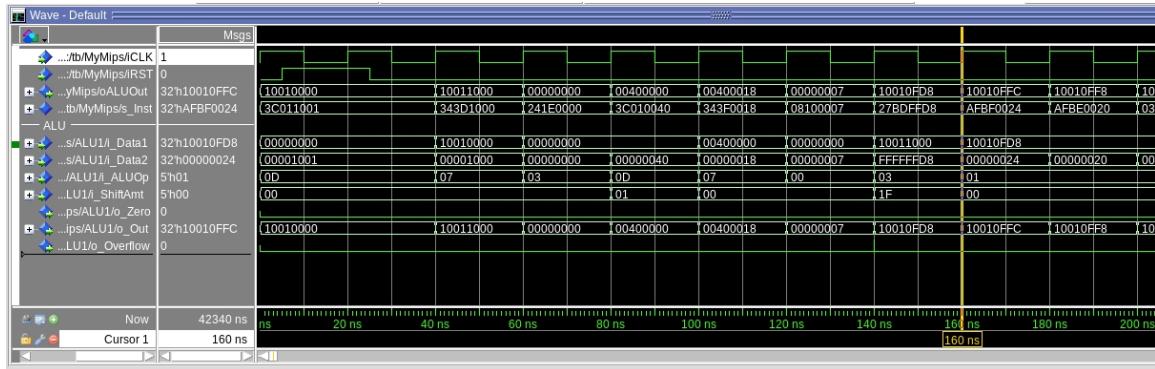


The above test consists of a sequence of addi instructions. The first cycle instruction is fed to the Control Unit and is broken down into a new internal signal consisting of the OpCode and the Function code. Since the OpCode portion of the instruction is 1000, the

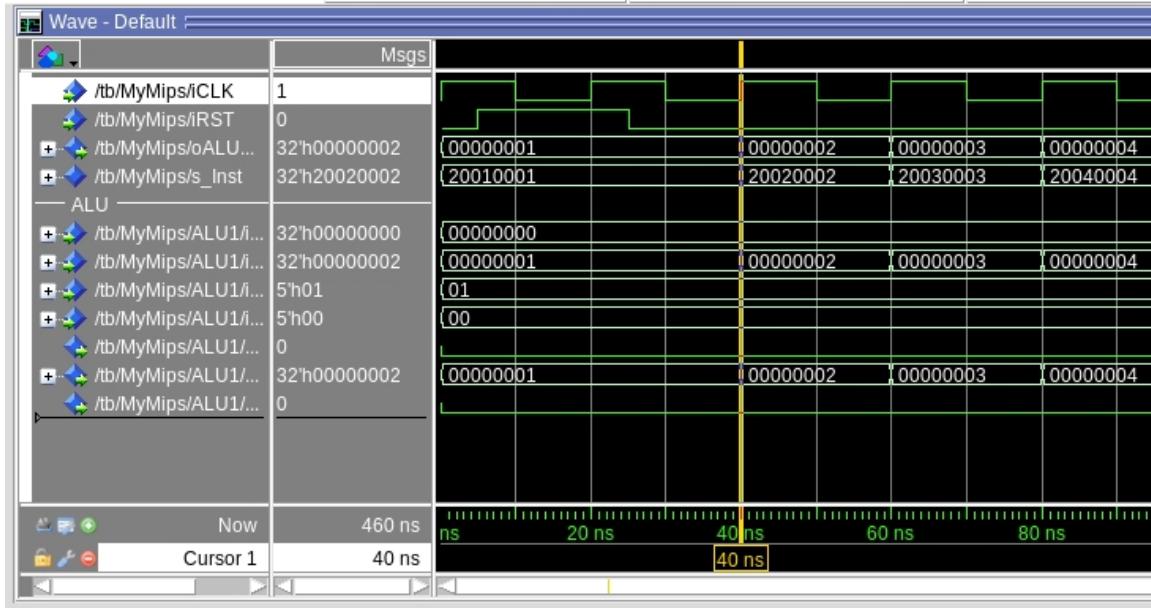
appropriate output signals are set for the various muxes and an ALUOp code of “00001” is forwarded to the ALU indicating to it to use the “addersubtractor” module. Since the instruction remains the same throughout the test, the Control unit does not change its outputs so the purpose of these series of tests really boils down to the reliability of the addersubtractor module.



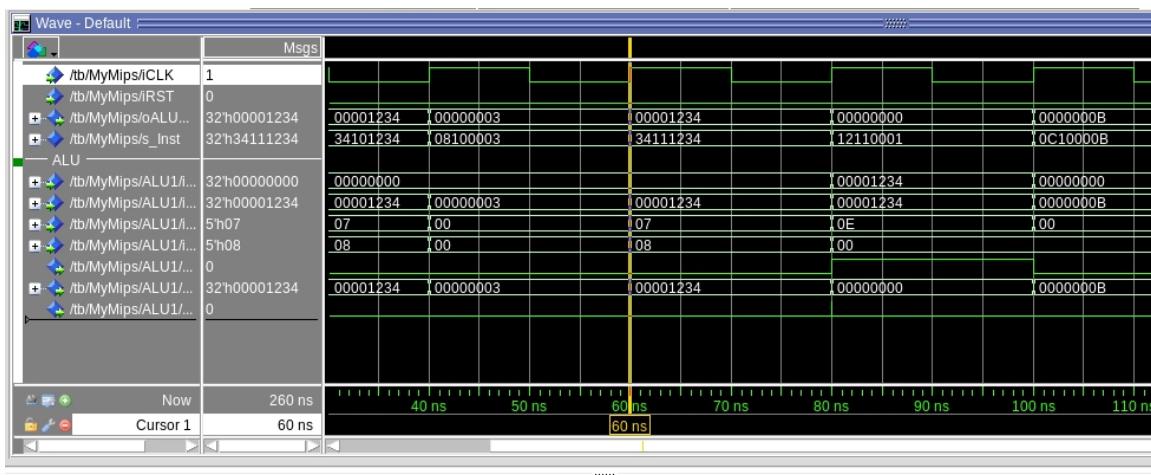
The above “fibonacci” test encounters an error with an jal instruction at MARS instruction number 132 but otherwise the tests work as expected. The first instruction (Beyond the reset cycle) is an “lui” instruction into MIPS register 1. The ALU output shows that the value 0x1001 is correctly loaded into the upper half of the register file as 0x10010000. The following instruction is an “ori” instruction that compares the zero register with the value in register 1. The output is similar to the input due to the logic of ori between zero and said input.



Test “grendel.s” utilizes sw commands throughout its process which is first executed 160 ns into the simulation. The instruction indicates that register 29’s (offset of 36) value is to be stored into memory register 31.



Test “lab3Seq.s” contains of many addi, sub, and add instructions which all have been used in previous instruction tests successively.



Test “simplebranch.s” uses many j, jal, and jr instructions. The j instruction writes the target address to the PC using MUXes, jal saves the next instruction to register 31 then jumps to the defined instruction, and jr jumps to an address that is stored within a specific register.

[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.

[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

[Part 3 (c)] Create and test an application that sorts an array with N elements using the BubbleSort algorithm ([link](#)). Name this file Proj1_bubblesort.s.

[Part 4] Report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematic from part 1. What components would you focus on to improve the frequency?

The critical path is highlighted in red below. Improving the ALU's branch logic performance would be our priority. Other components such as the AND gate and register files would likely be too difficult or impractical to improve further. Our frequency of the processor was about 23.1 MHz

