CprE 381: Computer Organization and Assembly-Level Programming

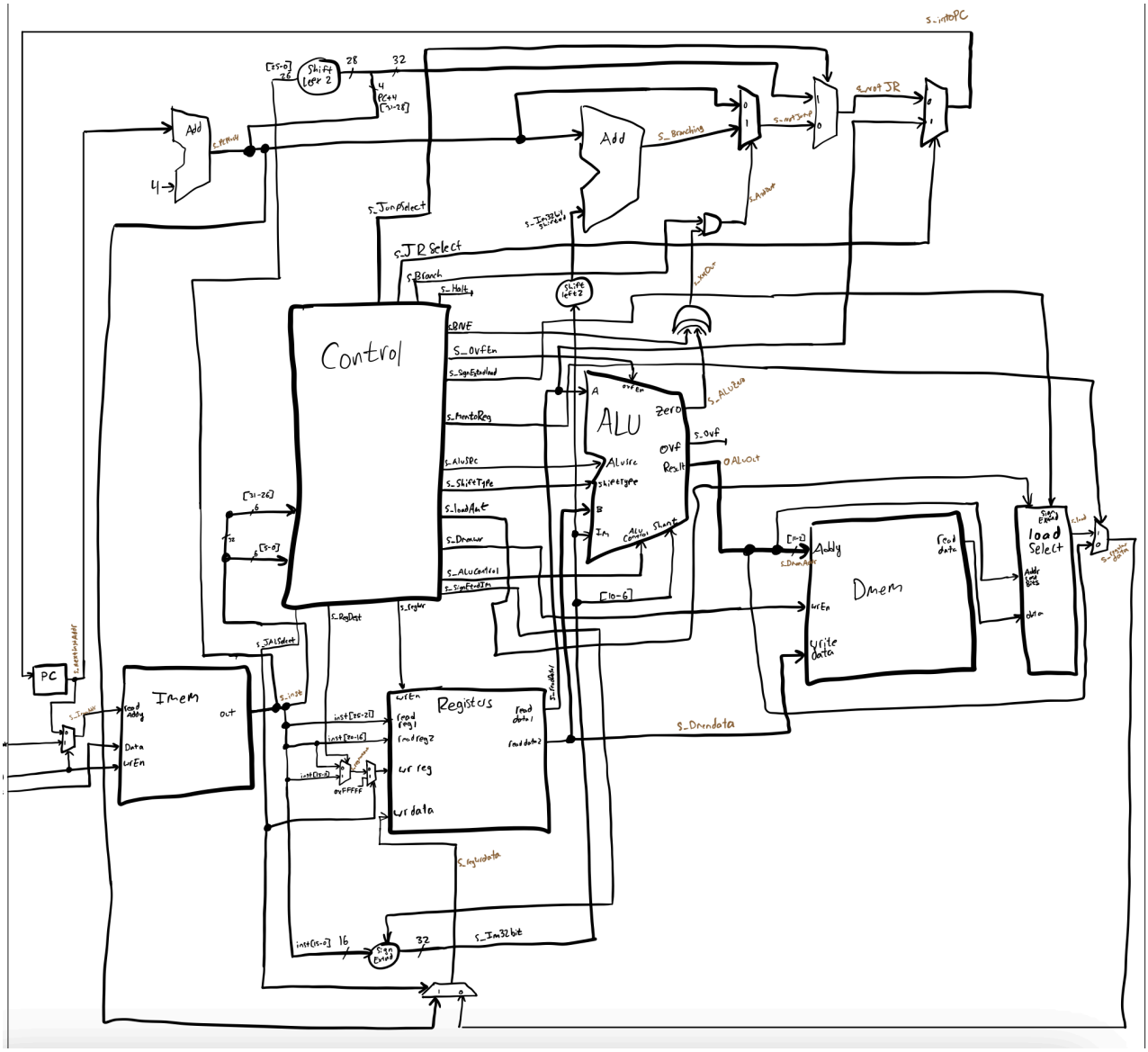
Project Part 1 Report

Team Members: Caden Otis and Christopher Hausner

Project Teams Group #: B_05

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.



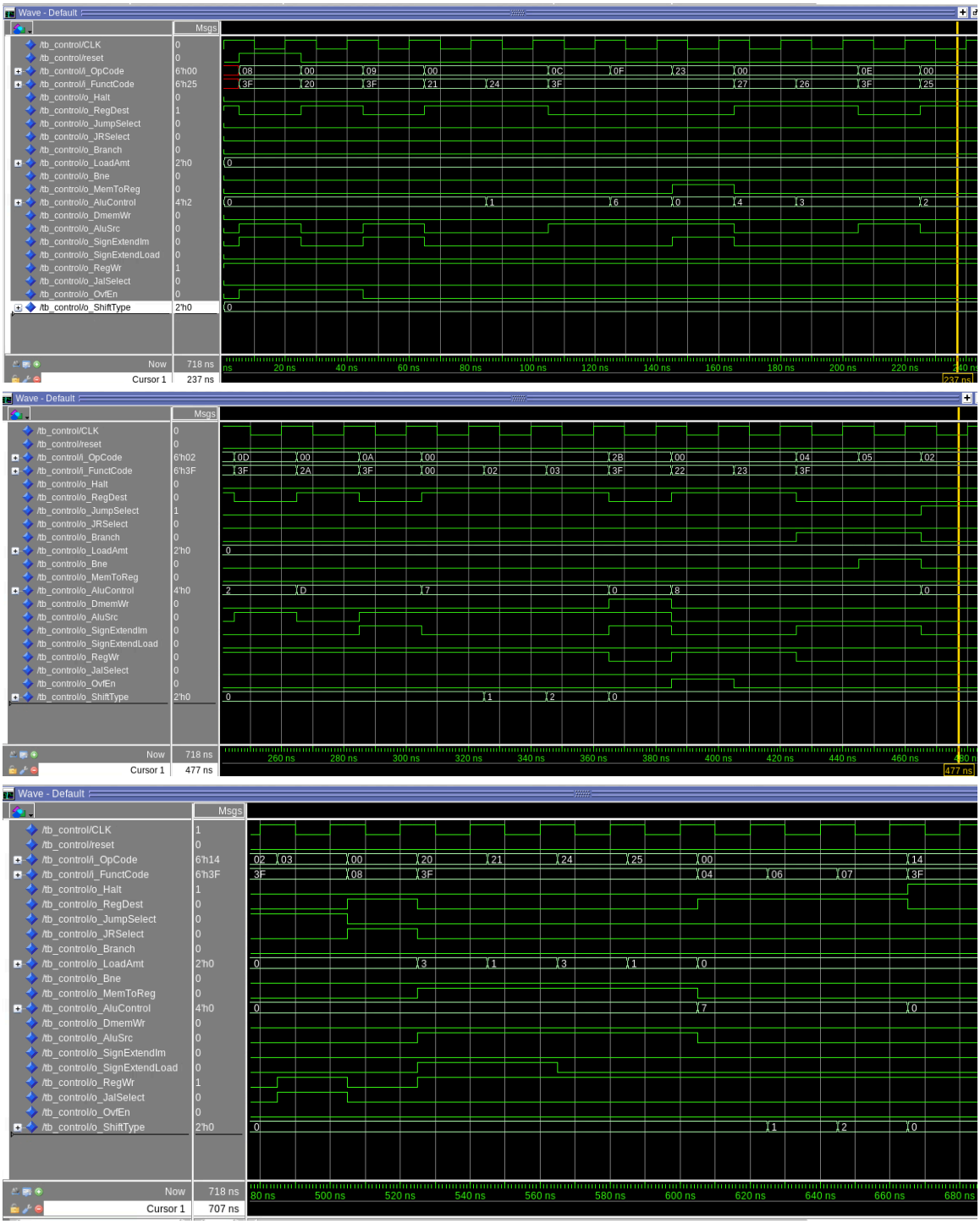
[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your datapath implementation. The end result should be an *N*M* table where each row corresponds to the output of the control logic module for a given instruction.

The spreadsheet will be included in the submission zip folder. Below is a screenshot of our spreadsheet, indicating how our control unit module will decode all required MIPS instructions.

Instruction	Opcode (Binary)	Funct (Binary)	Control Signals											
			SignSelectIM(0=unsigned;1= sign)	LoadAmt	MemtoReg	s_DMemWr *[MemWrite from text]*	s_RegWr *[RegWrite from text]*	JalSelect	JrSelect	JumpSelect	Branch	BNE	RegDst
xor	"000000"	"100110"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
xori	"001110"	"------"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	0 *(rt)*
or	"000000"	"100101"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
ori	"001101"	"------"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	0 *(rt)*
slt	"000000"	"101010"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
slti	"001010"	"------"	1	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	0 *(rt)*
sll	"000000"	"000000"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
srl	"000000"	"000010"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
sra	"000000"	"000011"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
sw	"101011"	"------"	1	0	0 (no read from mem)	1	0	0	0	0	0	0	DC (0)
sub	"000000"	"100010"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
subu	"000000"	"100011"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
beq	"000100"	"------"	1	0	0 (no read from mem)	0 (no write to mem)	0	0	0	0	1	0	DC (0)
bne	"000101"	"------"	1	0	0 (no read from mem)	0 (no write to mem)	0	0	0	0	1	1	DC (0)
j	"000010"	"------"	0	0	0 (no read from mem)	0 (no write to mem)	0	0	0	1	0	0	DC (0)
jal	"000011"	"------"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	1	0	1	0	0	DC (0)
jr	"000000"	"001000"	0	0	0 (no read from mem)	0 (no write to mem)	0	0	1	0	0	0	DC (0)
lb	"100000"	"------"	0	11	1	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	0 *(rt)*
lh	"100001"	"------"	0	1	1	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	0 *(rt)*
lbu	"100100"	"------"	0	11	1	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	0 *(rt)*
lhu	"100101"	"------"	0	1	1	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	0 *(rt)*
sllv	"000000"	"000100"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
srlv	"000000"	"000110"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)
srav	"000000"	"000111"	0	0	0 (no read from mem)	0 (no write to mem)	1 (Writes to reg)	0	0	0	0	0	1 (rd)

[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).

To create the control logic module, we thought it would be best to use a dataflow style architecture to construct the module. By using dataflow, we can make the decoding of the opcode and function code easy by just using a lot of with-select-when statements to determine the state of each control signal. By following the instruction order from our spreadsheet in Part 2 (a.i) to create our testbench, we saw that the resulting waveforms match the expected control signals. Below is the output that we got from our testbench. This matches what we would expect from our spreadsheet.

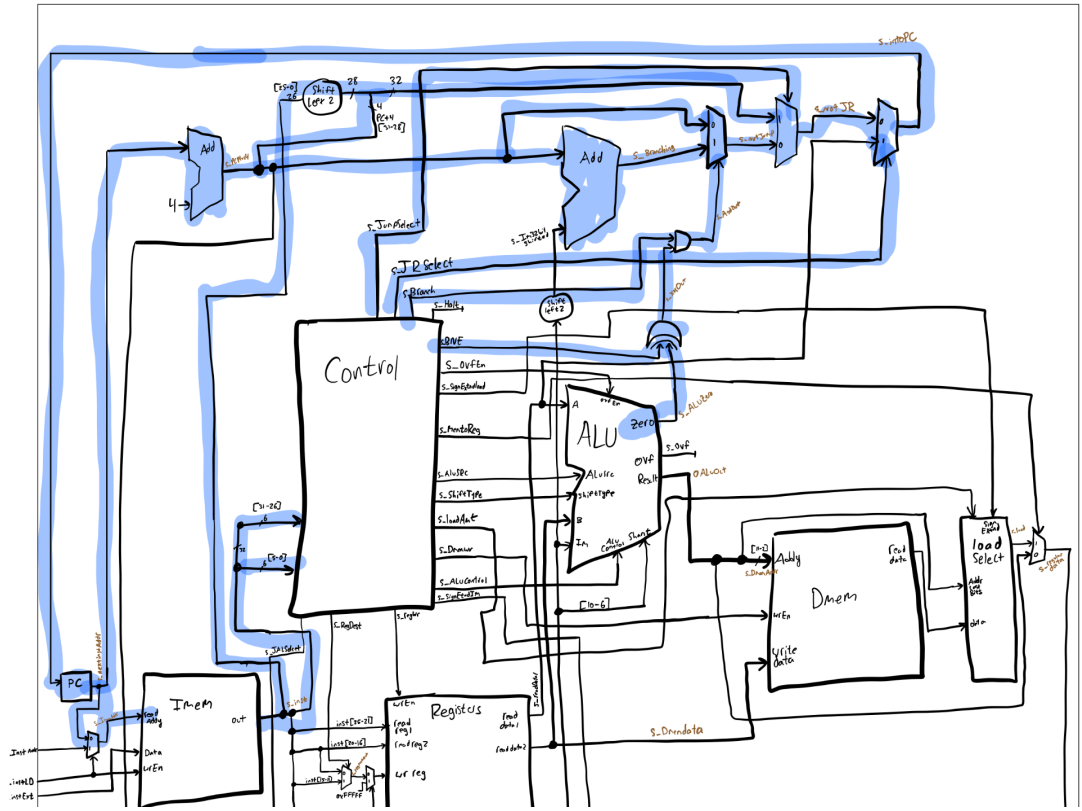


[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

Our instruction fetch logic must support **branching** (beq, bne) and **jumping** (j, jal, jr) instructions. Since the control unit will output control signals that determine the next

instruction the PC outputs, we need to ensure that our control logic module can correctly decode the branching and jumping instructions to set the correct control signals (s_Branch, s_JumpSelect, s_JRSelect, s_JalSelect), which we already did at the end of part 1.

[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?

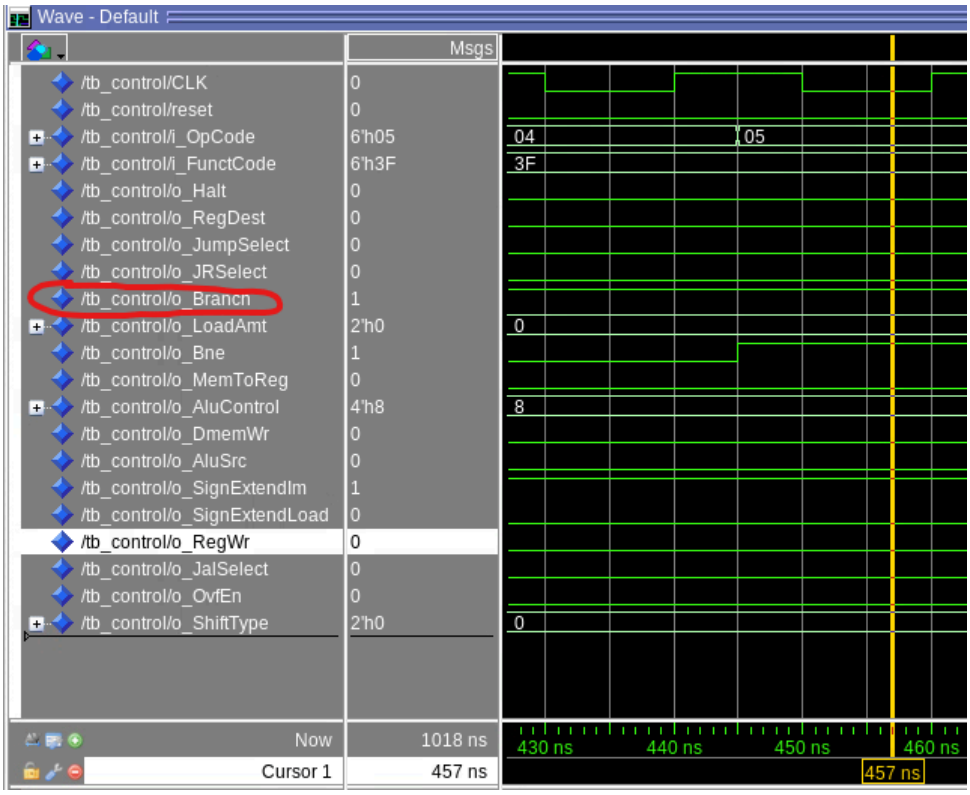


The highlighted section is for the fetch logic. Out of the PC, it goes into Imem (as long as we aren't loading the Imem). From Imem it goes both into the control, and the instruction goes up to the jump MUX to be used if we are jumping. With the way Jump works in MIPS, we will shift it left 2 bits and append the MSB of the jump instruction for the address. Then if the s_Jumpselect mux is 1 we will perform a jump. Moving down the line, we also have a branching unit. Branching will add/subtract to the pc position and is PC relative (unlike jump). If we are branching the Branching control selector will be 1. This will be ANDed with the XOR of both BNE and the zero flag, essentially being 1 if its BNE and not zero, or BEQ and zero. This will correctly select if we are branching. Lastly, we have our JR mux. If we are performing a JR, we will read the value from the register and jump to that value. If not, the s_JRselect will be zero and we will perform the other option that is fed through (Jump, Branch, or Normal). This is all we need for our fetch logic!

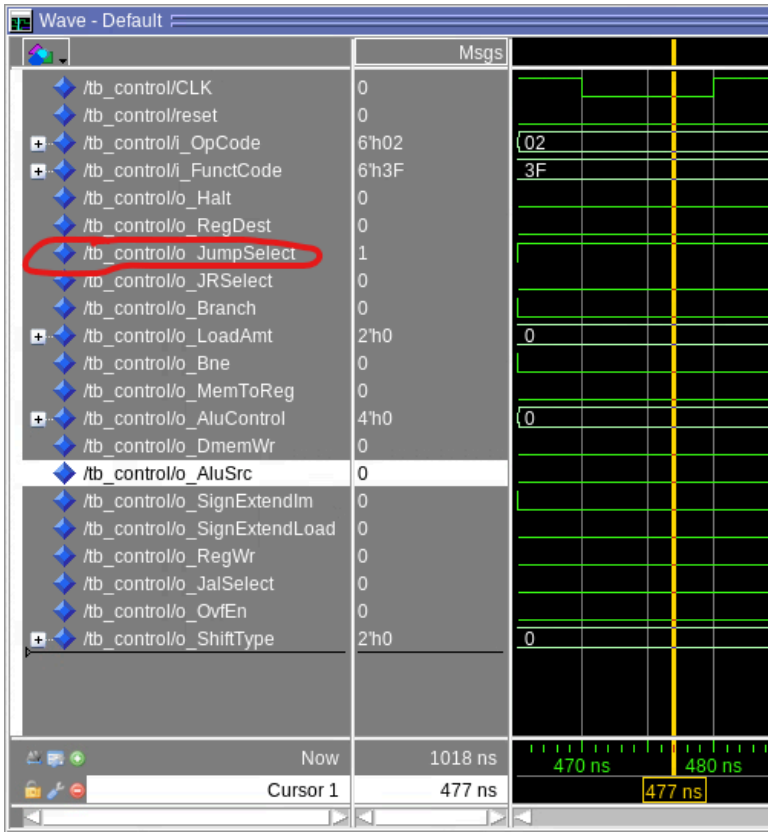
[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use QuestaSim to test your design thoroughly to make sure it is working as expected. Describe how the

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

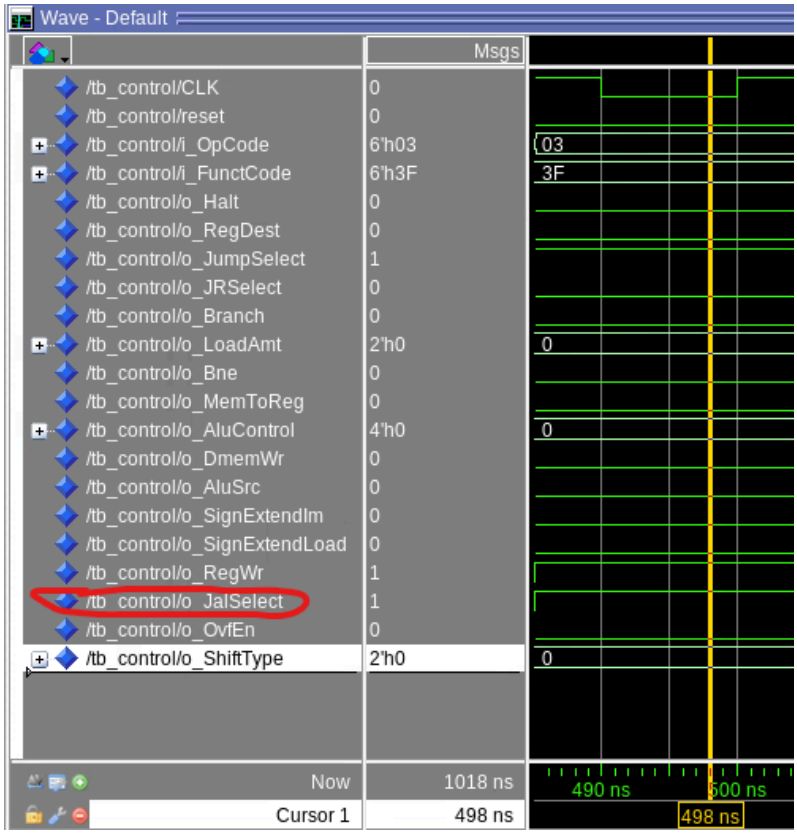
Using with-select-when statements to easily decode the opcode and function code, our control logic module can correctly set s_Branch, s_JumpSelect, s_JRSelect, s_JalSelect control signals to accurately update the next instruction in the PC. In the below waveform, it is showing the output of the control logic module when it has received a beq and a bne instruction. Our control unit was able to correctly decode the two branch instructions and set the s_Branch control signal to be 1. A screenshot of the waveform generating this result is shown below.



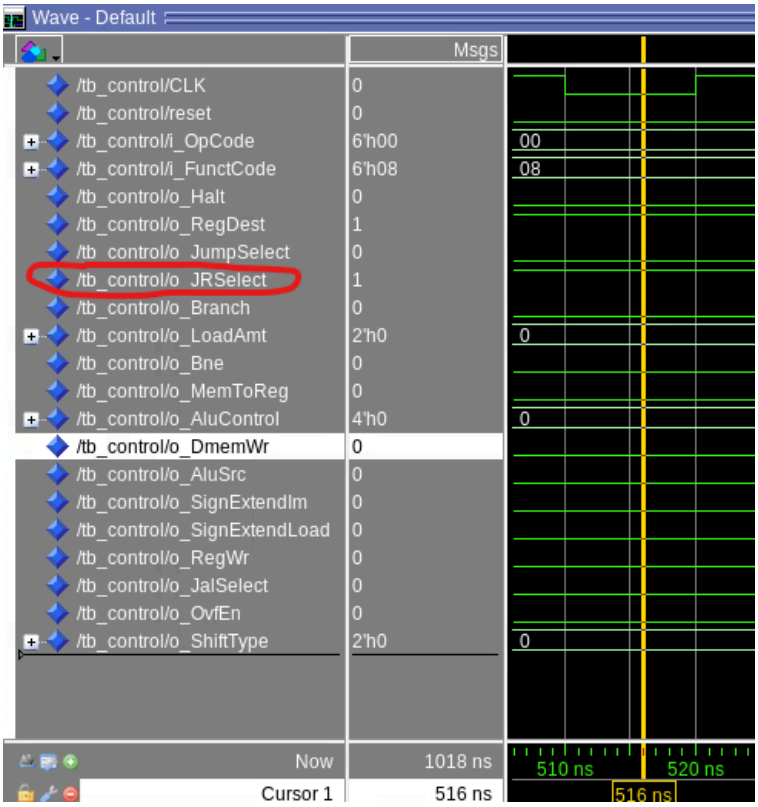
For the j instruction, our control unit was also able to decode the instruction and correctly set the s_JumpSelect control signal to 1. A screenshot of the waveform generating this result is shown below.



For the jal instruction, our control unit was also able to decode the instruction and correctly set the s_JalSelect control signal to 1. A screenshot of the waveform generating this result is shown below.



For the j instruction, our control unit was also able to decode the instruction and correctly set the s_JRSelect control signal to 1. A screenshot of the waveform generating this result is shown below.



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

Both the srl and the sra instructions will shift a value in a register to the right by an amount specified with an immediate value. However, the srl instruction will fill in the vacant upper bits with zeros, therefore zero extending the shifted value. Inversely, the sra instruction will fill in the vacant upper bits with the most significant bit of the shifted value, therefore sign extending the shifted value.

MIPS doesn't have an sla instruction because the sll instruction already preserves sign bits naturally. When the sll instruction is used, it will shift the most significant bits to the left and fill in the vacant lower significant bits with zeros. During this process, the upper bits from the non-shifted value are still present in the now shifted value. However, the opposite happens in the sra and srl instructions, where the most significant bits get shifted to the right and turned into least significant bits. Because of this, it is more crucial to think about how to replace those now vacant upper bits. This is why there are zero-extended and sign-extended (srl and sra) instructions for shifting right, but only one instruction (sll) for shifting left.

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

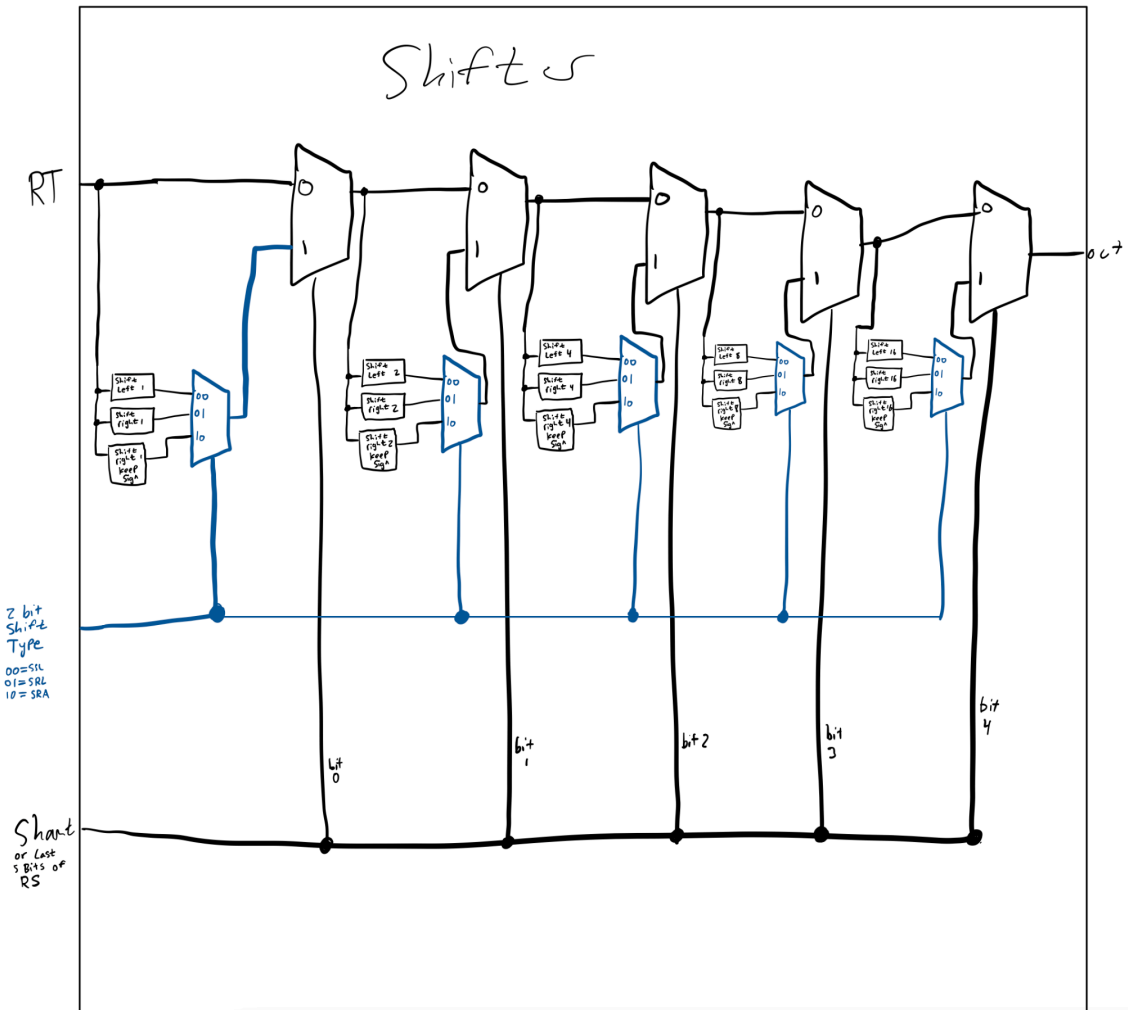
In our shifter module, which handles the srl and sra shifting operations, it takes in 3 inputs: i_RT, i_ShiftType, and i_Shamt. The i_RT input is a 32-bit input that takes in the value to shift. The i_ShiftType input is a 2-bit control input that determines whether to do a srl or sra shift (will also be used to do a sll shift later on). The i_Shamt input is a 5-bit input that acts as a control signal to determine how much to shift the value in i_RT. Our shifter will then output the final shifted value as a 32-bit output.

To actually perform the shifting operation within our shifter module, the value in i_RT will go through 5 stages. The first stage will shift i_RT by 1 to the right for both srl and sra operations, where there will be one signal that has the value shifted to the right and the most significant bit will be replaced with a zero, and there will be another signal with the value shifted to the right and the most significant bit will be replaced with itself. These two values will be the inputs into a 2:1 mux, and the select input will be i_ShiftType to choose either the srl- or sra-shifted value. That result, and the i_RT value will go into another 2:1 mux, with the select input being the least significant bit in i_Shamt. This mux will output the shifted value if the select bit is 1. There are then 4 more stages of 2 2:1 muxes that our shifter module goes through, where each stage shifts by an increasing power of 2 (1, 2, 4, 8, 16), allowing for shifts up to 31 bits.

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

To allow our shifter module to support left shifting operations, which will just be the sll instruction, we can add a third shifting type to the 5 shifting stages of our shifter module. To do this, the i_ShiftType input will now encode three instructions: srl, sra, and sll. At the beginning of each stage, there will be an additional signal that will perform a left shift. We then need to switch the first 2:1 mux into a 4:1 mux so that the three shifted values can be inputs into the mux, and then the i_ShiftType select input will pick between the 3 shifted values based on the three shifting instructions. The other functionality of the shifter module will stay the same, allowing us to support the srl, sra, and sll instructions.

The schematic below describes our shifter module with its five-stage shifting functionality:

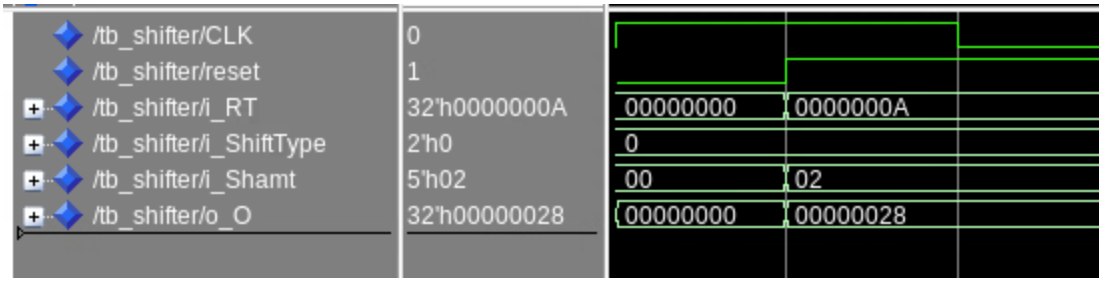


[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.

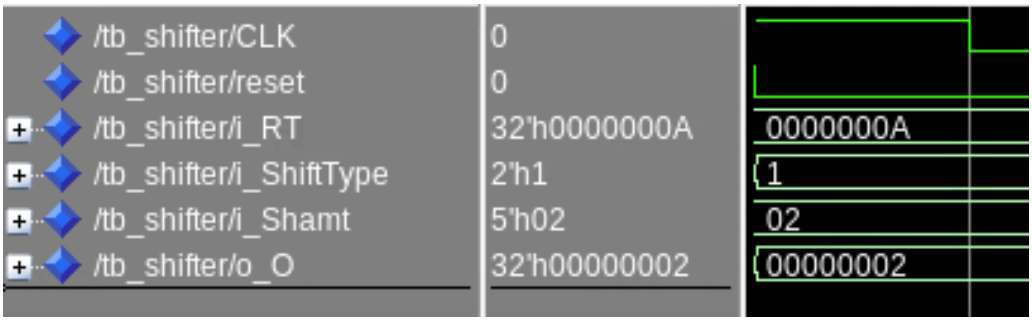
When encoding the shift type in i_ShiftType, the following is the encoding we use to determine what shift operation to perform:

- 00 → sll
- 01 → srl
- 10 → sra

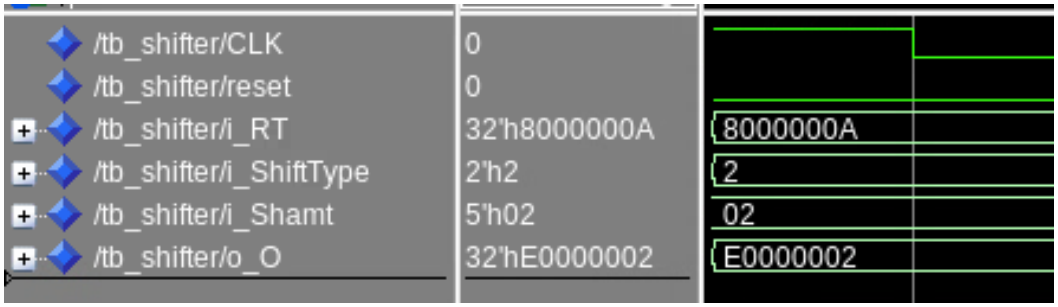
Using this encoding, we can look at the first test that tests if our shifter module works with the sll instruction. In the test, it wants to shift the value 0xA to the left by 2, so the inputs will be i_RT = 0xA, i_ShiftType = 0b00, i_Shamt=0b00010. If done correctly, when the value 0xA is shifted to the left by two, that results in the value 0x28. The waveform below matches this expected behavior.



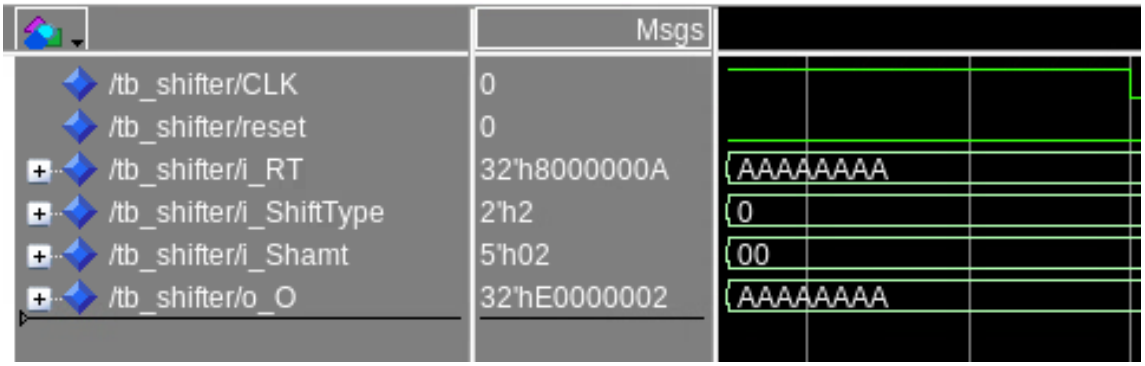
We will now look at the test that checks if our shifter module works with the srl instruction. In the test, it wants to shift the value 0xA to the right by 2 and then zero-extended, so the inputs will be i_RT = 0xA, i_ShiftType = 0b01, i_Shamt=0b00010. If done correctly, when the value 0xA is shifted to the right by two, zeros will fill in the lower bits and result in the value 0x2. The waveform below matches this expected behavior.



We will now look at the test that checks if our shifter module works with the sra instruction. In the test, it wants to shift the value 0xA to the right by 2 and then sign-extended, so the inputs will be i_RT = 0xA, i_ShiftType = 0b10, i_Shamt=0b00010. If done correctly, when the value 0xA is shifted to the right by two, zeros will still fill in the lower bits and result in the value 0x2. The waveform below matches this expected behavior.

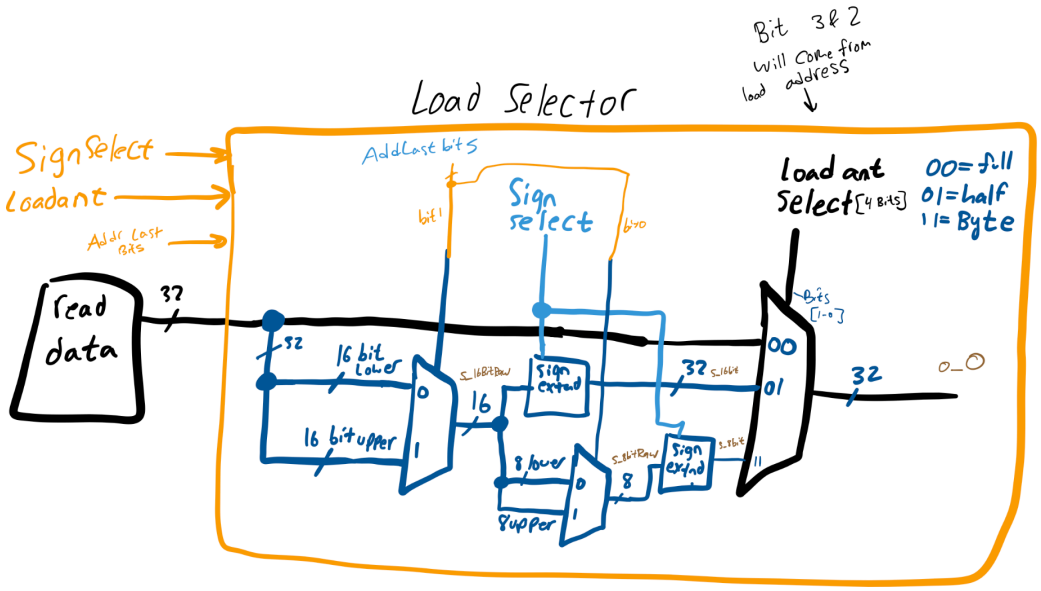


To fully make sure our shifter module works correctly, we made a test that checks if the value in i_RT won't be shifted at all if the input i_Shamt = 0. So if i_RT = 0xAAAAAAAA, then o_O should be 0xAAAAAAAA. The waveform below matches this expected behavior.



[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.

One component we added was a LoadSelector. See image below.



The purpose of this is for when we want to do lh, lhu, lb or lbu. Traditionally for all other instructions, we can just select the raw output of Dmem which is LoadAmtSelect option 00. It loads the full 32 bits. For these specific instructions however we want to be able to load just a byte or a halfword and set the rest to zero. This will select upper or lower 16 bits and sign extend it correctly based on a control signal. If this isn't enough then we are accessing a byte and for that we have another mux. Then the 8 bits are sign extended. We can select from 00 (32 bits), 01 (16 bits) or 11 (8 bits). Addlastbits comes from the ALU result of the last 2 bits. If we are attempting to address a word, these will end in 00. However when addressing a half word this will be 10, and for a bye will be (01) or (11). That is taken to choose which bits to send through the Mux levels. One design choice was if we want the shifter to be inside of the ALU. Ultimately we found it to be simpler to be done inside of the ALU, and so we added that to the ALU and added the control signals needed. We did this because having the shifter outside could have been more confusing and cluttered our design. Potentially our design could be improved by putting it outside

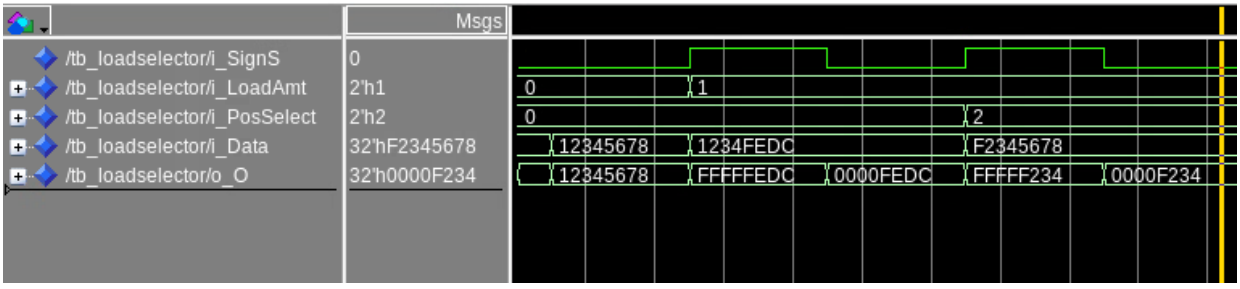
of the ALU and not making it 3 in 1 in the design we have with muxes, but it does the job and it does it correctly.

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

To make sure that our LoadSelector module works properly, we first wanted to test its ability to correctly load unsigned and signed halfwords. To verify this, we created the following tests:

- **Test 1**: load the full amount of the value 0x12345678 (**lw instruction**). So if i_Data = 0x12345678 and i_LoadAmt = 00 (load 32 bits), then o_O should be 0x12345678
- **Test 2**: load the lower half amount of the value 0x1234FEDC and sign extend the remaining bits (**lh instruction**). So if i_Data = 0x1234FEDC, i_LoadAmt = 01 (load 16 bits), and i_SignS = 1, then o_O should be 0xFFFFFEDC
- **Test 3**: load the lower half amount of the value 0x1234FEDC without sign extending the remaining bits (**lhu instruction**). So if i_Data = 0x1234FEDC, i_LoadAmt = 01 (load 16 bits), and i_SignS = 0, then o_O should be 0x0000FEDC
- **Test 4**: load the upper half amount of the value 0xF2345678 and sign extend the remaining bits (**lh instruction**). So if i_Data = 0xF2345678 , i_LoadAmt = 01 (load 16 bits), and i_SignS = 1, then o_O should be 0xFFFFF234
- **Test 5**: load the upper half amount of the value 0xF2345678 and not sign extend the remaining bits (**lhu instruction**). So if i_Data = 0xF2345678 , i_LoadAmt = 01 (load 16 bits), and i_SignS = 0, then o_O should be 0x0000F234

The waveform below shows that its behavior matches the expected results from the tests outlined above

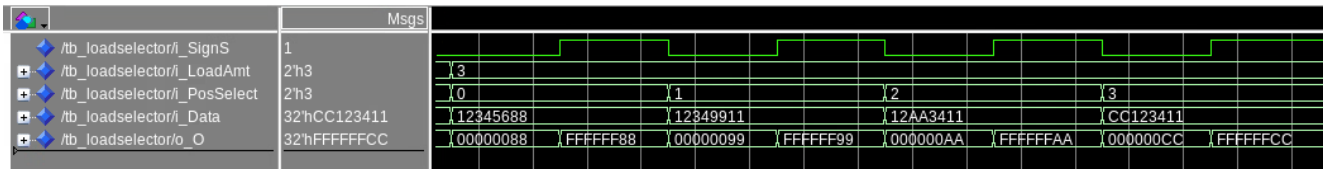


To further make sure that our LoadSelector module works properly, we now need to test its ability to correctly load unsigned and signed byte values. To verify this, we created the following tests:

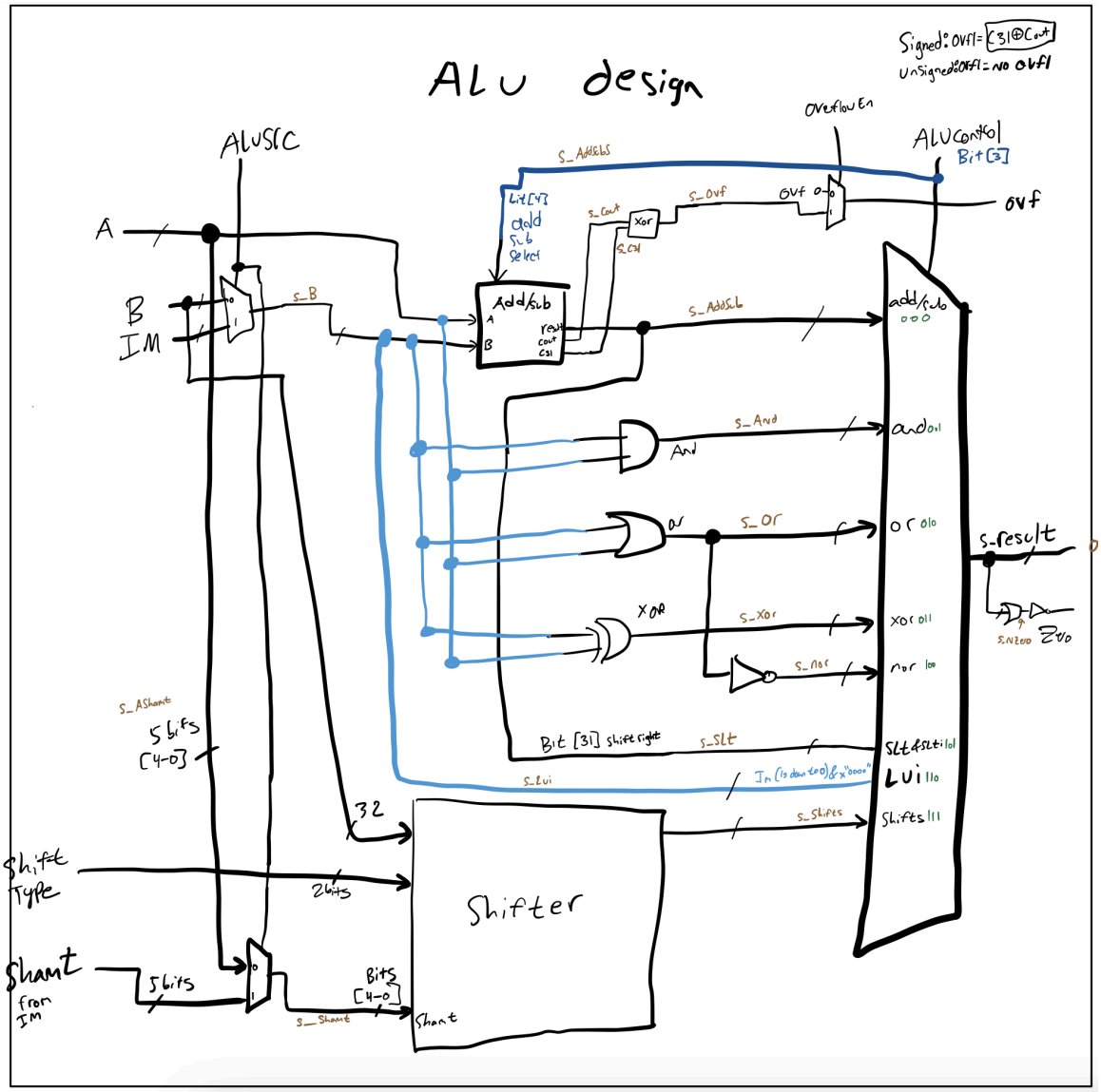
- **Test 1**: load the unsigned bottom byte of the value 0x12345688 (**lbu instruction**). So if i_Data = 0x12345688, i_LoadAmt = 11 (load 8 bits), i_SignS = 0, then o_O should be 0x00000088

- **Test 2**: load the signed bottom byte of the value 0x12345688 (**lb instruction**). So if i_Data = 0x12345688, i_LoadAmt = 11 (load 8 bits), i_SignS = 1, then o_O should be 0xFFFFF88
- **Test 3**: load the unsigned second last bottom byte of the value 0x12349911 (**lbu instruction**). So if i_Data = 0x12349911, i_LoadAmt = 11 (load 8 bits), i_SignS = 0, then o_O should be 0x00000099
- **Test 4**: load the signed second last bottom byte of the value 0x12349911 (**lb instruction**). So if i_Data = 0x12349911, i_LoadAmt = 11 (load 8 bits), i_SignS = 1, then o_O should be 0xFFFFFF99
- **Test 5**: load the unsigned third last bottom byte of the value 0x12AA3411 (**lbu instruction**). So if i_Data = 0x12AA3411, i_LoadAmt = 11 (load 8 bits), i_SignS = 0, then o_O should be 0x000000AA
- **Test 6**: load the signed third last bottom byte of the value 0x12AA3411 (**lb instruction**). So if i_Data = 0x12AA3411, i_LoadAmt = 11 (load 8 bits), i_SignS = 1, then o_O should be 0xFFFFFFAA
- **Test 7**: load the unsigned last bottom byte of the value 0xCC123411 (**lbu instruction**). So if i_Data = 0xCC123411, i_LoadAmt = 11 (load 8 bits), i_SignS = 0, then o_O should be 0x000000CC
- **Test 8**: load the signed last bottom byte of the value 0xCC123411 (**lb instruction**). So if i_Data = 0xCC123411, i_LoadAmt = 11 (load 8 bits), i_SignS = 1, then o_O should be 0xFFFFFFCC

The waveform below shows that its behavior matches the expected results from the tests outlined above



[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?

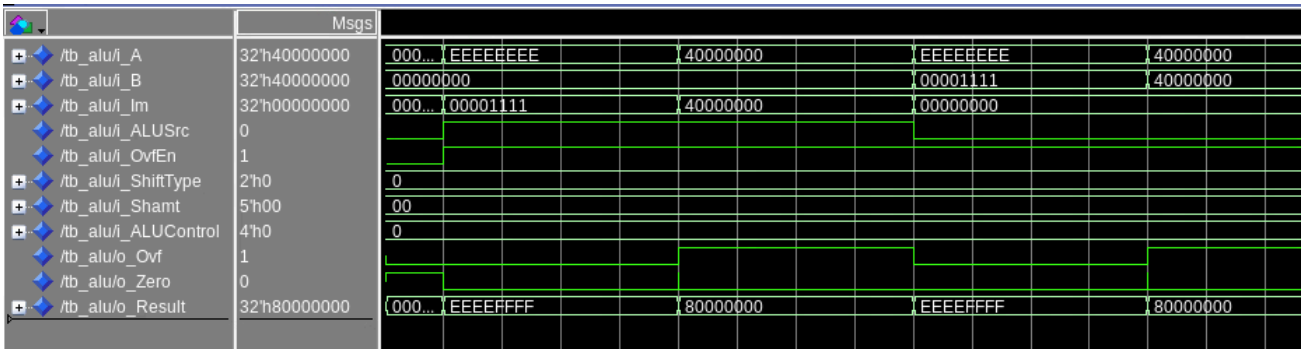


This is our ALU design. We had to make some choices, such as to put our Shifter inside of our ALU. Because of this we also had to have IM and B value feed into our ALU and select inside of it. The way we did it ALUSrc is multi purposed to select B or IM and Shamt to feed into the shifter. Because of this if ALUSrc is 1, and we don't feed B into the ALU before the mux selector, we will get the IM value which is wrong. Easiest fix was to feed IM and B into the alu. (B is the rt value from our Register to clarify). Then from here our ALUControl will select which operation we will perform. Overflow is calculated from our adder/subtractor where we added the c31 output. The c31 output is Xor'ed with the Cout bit. When this is 1 we have overflow. However, some instructions should not have overflow. For this we added a OvflEN control that will enable the output of the Overflow. If this is 0, we hard code the output of Ovfl to 0 because we cant have any. For the Zero flag, we take the output and OR all of the bits together. Then we negate this and we have our zero value! Finally, SLT and SLTI uses our subtractor. If our value in bit 31, aka the signed bit, is zero then we know that it was not less than. If our number is negative then we know it was less than. (A is less than B, A is less than IM depending

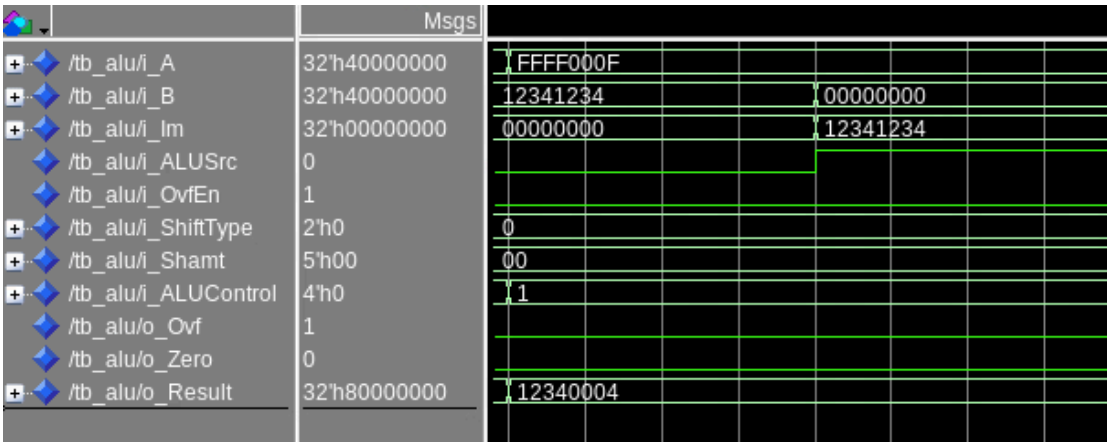
on slt or slti). One last note is our ALUControl MSB is the selector for adding (0) or subtracting (1). ALU control is 4 bits wide.

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

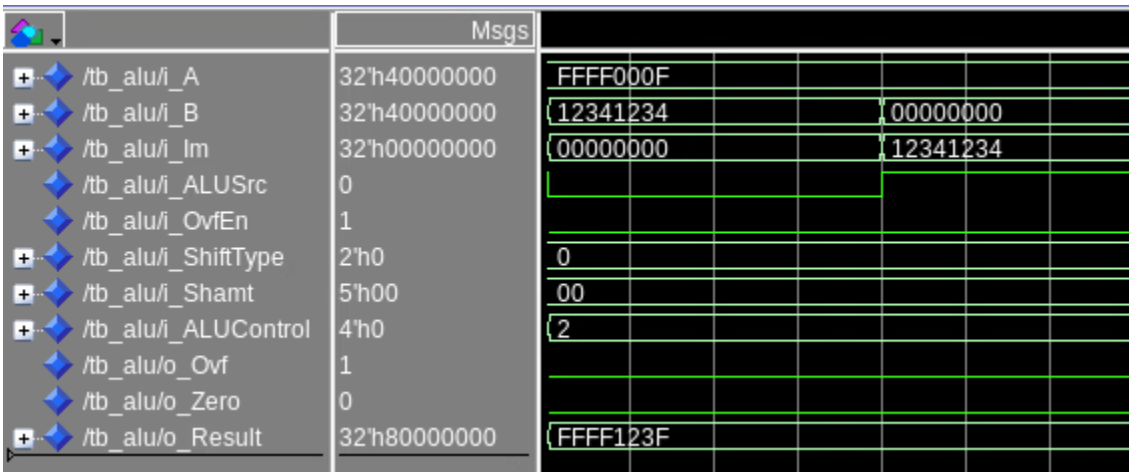
In our ALU design, if it gets the control signal ALUControl = 0x0, then it will know to perform an addition operation, either doing one of the add, addi, addu, or addiu instructions. Further, the control signal ALUSrc can be used to determine if the ALU needs to do an immediate operation or not. If ALUSrc = 1, then an immediate value will enter the adder/subtractor and addi or addiu will be performed. Otherwise, add or addu will be performed with the values from the two read registers. The waveform below demonstrates the four add instructions (add, addi, addu, or addiu) being performed by the ALU.



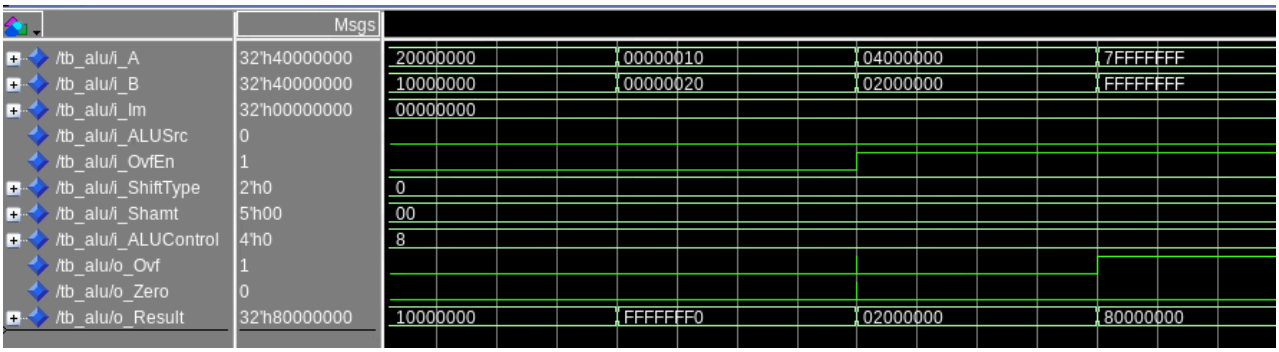
In our ALU design, if it gets the control signal ALUControl = 0x1, then it will know to perform a bitwise AND operation. Further, the control signal ALUSrc can be used to determine if the ALU needs to do an immediate operation or not. If ALUSrc = 1, then an immediate value will enter the adder/subtractor and the andi instruction will be performed. Otherwise, the and instruction will be performed with the values from the two read registers. The waveform below demonstrates the two and instructions (and and andi) being performed by the ALU.



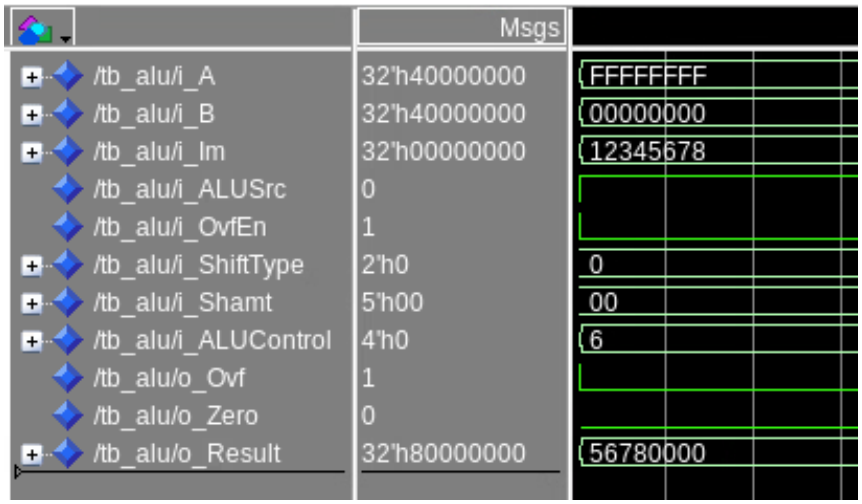
In our ALU design, if it gets the control signal ALUControl = 0x2, then it will know to perform a bitwise OR operation. Further, the control signal ALUSrc can be used to determine if the ALU needs to do an immediate operation or not. If ALUSrc = 1, then an immediate value will enter the adder/subtractor and the ori instruction will be performed. Otherwise, the or instruction will be performed with the values from the two read registers. The waveform below demonstrates the two and instructions (or and ori) being performed by the ALU.



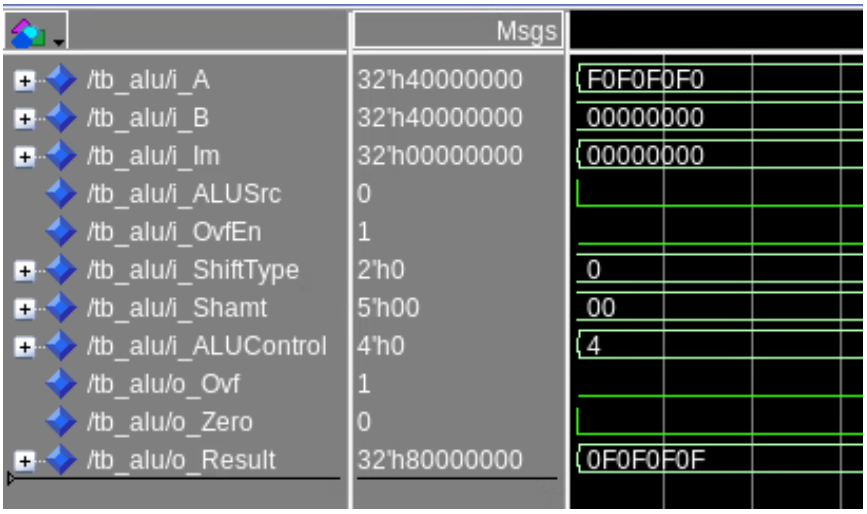
In our ALU design, if it gets the control signal ALUControl = 0x8, then it will know to perform a subtraction operation (sub, subu). Further, the control signal OvfEN can be used to determine which of the two subtraction instructions to use. If OvfEN = 1, then overflow is allowed and the sub instruction will be performed. Otherwise, overflow is not allowed and the subu instruction will be performed. The waveform below demonstrates the two and instructions (sub and subu) being performed by the ALU.



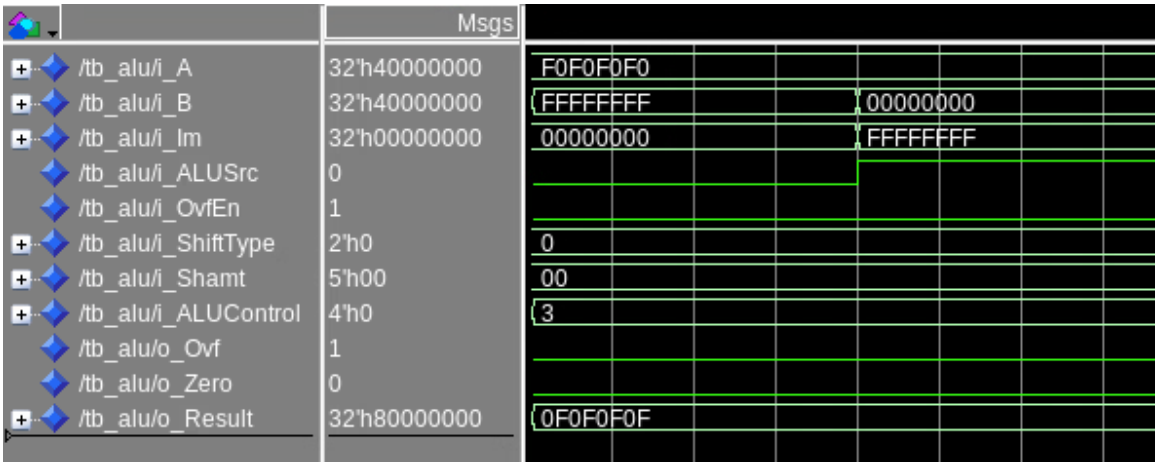
In our ALU design, if it gets the control signal ALUControl = 0x6, then it will know to perform the lui instruction. Further, the control signal ALUSrc should be set to 1 so that the immediate value can enter the ALU and then be loaded into a register. The waveform below demonstrates the lui instruction being performed by the ALU.



In our ALU design, if it gets the control signal ALUControl = 0x4, then it will know to perform the nor instruction. Further, the control signal ALUSrc should be set to 0 so that the value from two registers can be nor'ed together, and the control signal OvfEN = 0 to not allow for any overflow. The waveform below demonstrates the nor instruction being performed by the ALU.

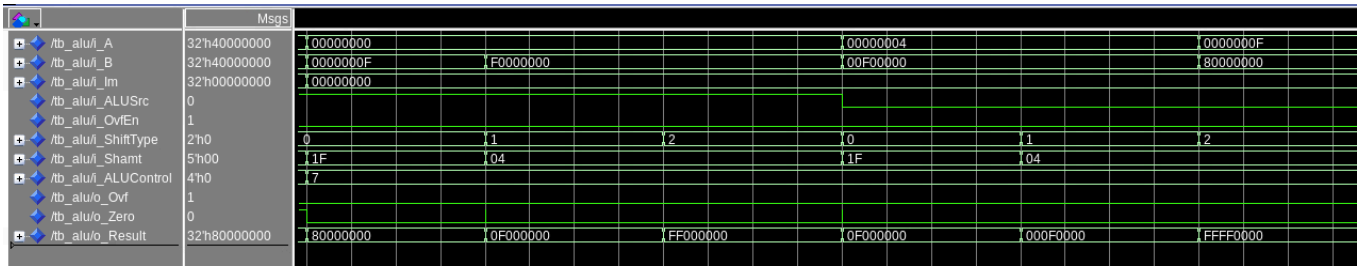


In our ALU design, if it gets the control signal ALUControl = 0x3, then it will know to perform either the xor or xori instruction. Further, the control signal ALUSrc can be used to determine which of the two instructions to use, and the control signal OvfEN = 0 to not allow for any overflow. If ALUSrc = 0, then the ALU will perform the xor instruction and the value from two registers will be xor'ed together. Otherwise, the ALU will perform the xori instruction and a register value and the immediate value will be xor'ed together The waveform below demonstrates the xor and xori instructions being performed by the ALU.



In our ALU design, if it gets the control signal ALUControl = 0xD, then it will know to perform either the slt or slti instruction. Further, the control signal ALUSrc can be used to determine which of the two instructions to use, and the control signal OvfEN = 0 to not allow for any overflow. If ALUSrc = 0, then the ALU will perform the slt instruction and the value from two registers will be compared to see if the value from the first register is less than the value from the second register. Otherwise, the ALU will perform the slti instruction and the value from the first register will be checked if it is less than the immediate value. The waveform below demonstrates the slt and slti instructions being performed by the ALU.

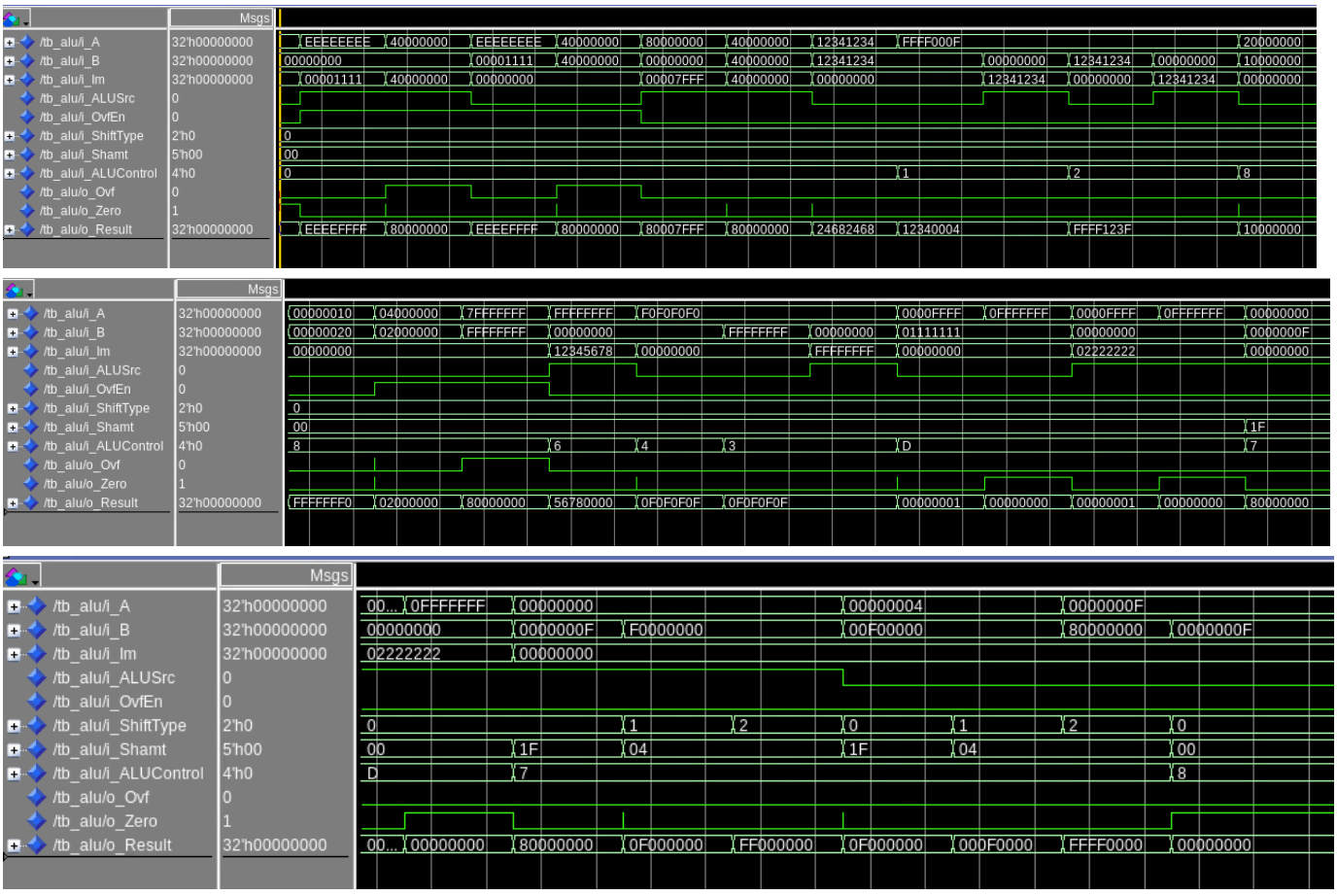
In our ALU design, if it gets the control signal ALUControl = 0x7, then it will know to perform either the sll, srl, sra, sllv, srlv, or srav instruction. Further, the control signal OvfEN should be set to 0 to not allow for any overflow. If ALUSrc = 1, then the ALU will use an immediate value and perform either the sll, srl, or sra instruction. Otherwise, it will perform either the sllv, srlv, or srav instruction. The exact instruction is encoded by the i_ShiftType input (encoding talked about in Part 2 (c.i.4)). The i_Shamt input is then used to determine how much to shift the value in the register by. The waveform below demonstrates the sll, srl, sra, sllv, srlv, or srav instructions being performed by the ALU.



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

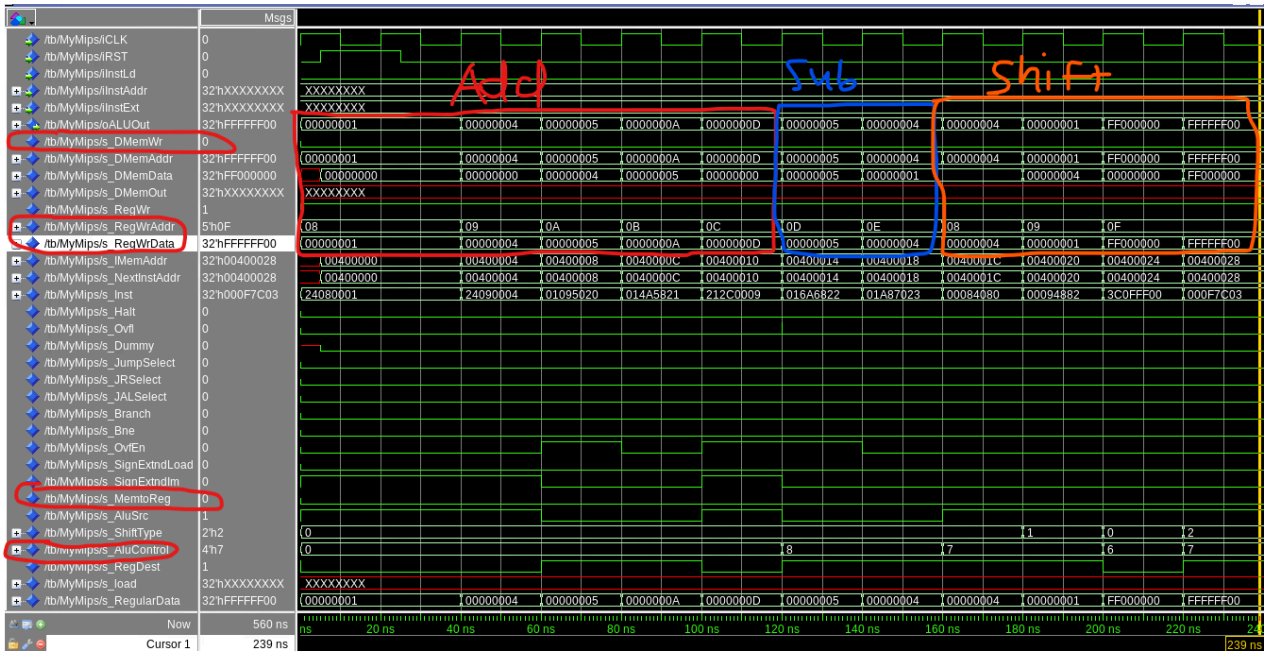
Our test plan for our ALU module is comprehensive because it tests every logical/arithmetic MIPS instruction at least once, with some instructions being tested multiple times with different values and also seeing how they react to overflow. Our test plan has thoroughly tested our ALU to ensure that for almost any possible sequence of inputs, the ALU will output the correct, expected results.

We came up with our test plan in Part 2 (c.v) that tested every logical/arithmetic MIPS instruction at least once, so below is a summary of the resulting waveform from our test plan that matches what we were expecting



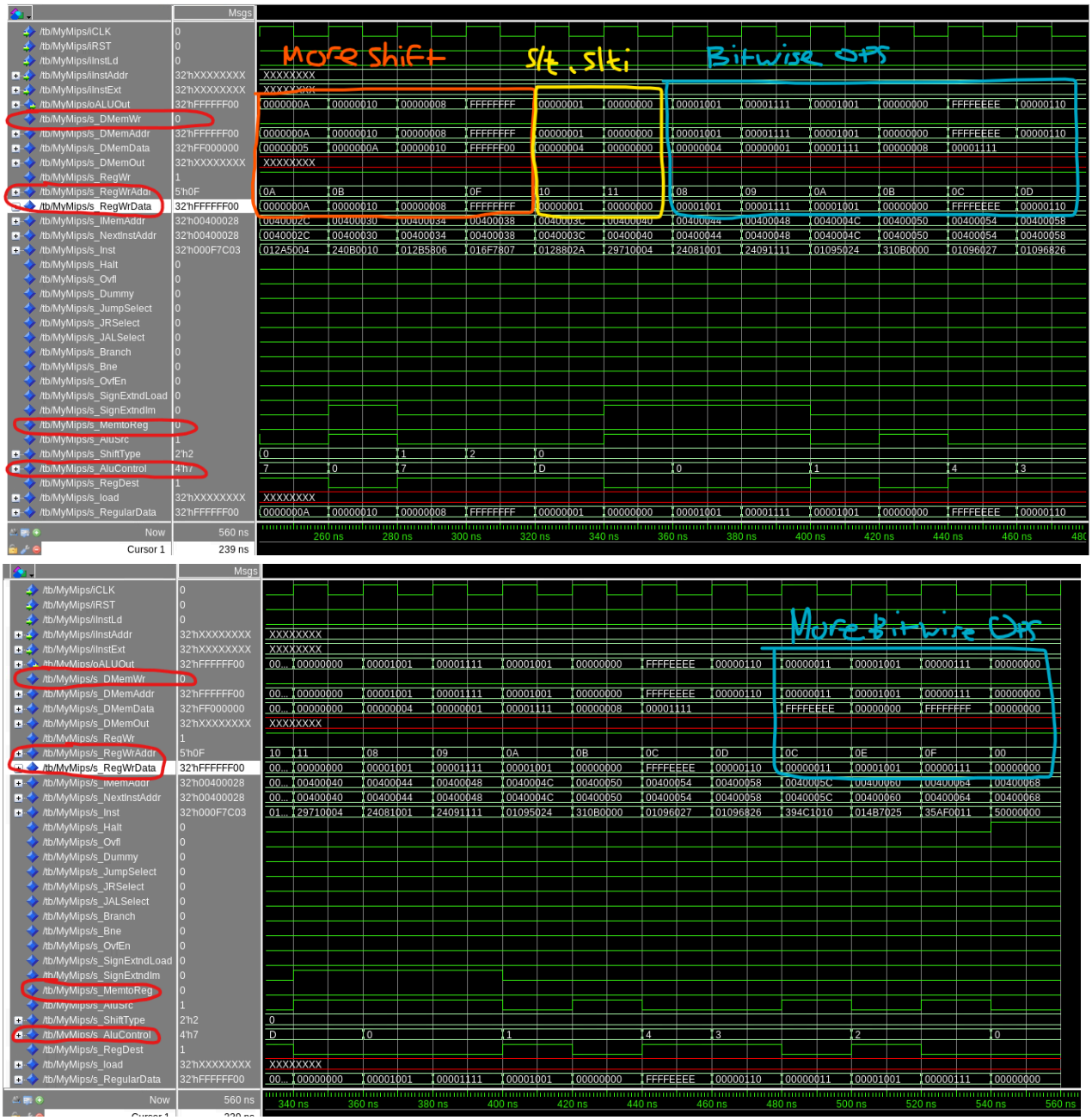
[Part 3] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

For our test Proj1_base_test.s, we were required to make use of every arithmetic/logical instruction once. For every instruction, the control unit was able to correctly decode each instruction and set the ALUControl signal to the correct operation for the ALU to perform. This then caused the correct address to be inputted into Dmem, where data memory would never write values into memory (DmemWr was always zero) since arithmetic and logical instructions don't require that functionality. Then the output from the ALU was used as input to the register file to write the new value to the correct register (RegWrData and RegWrAddr). Below is the resulting waveform from our Proj1_base_test.s test file.



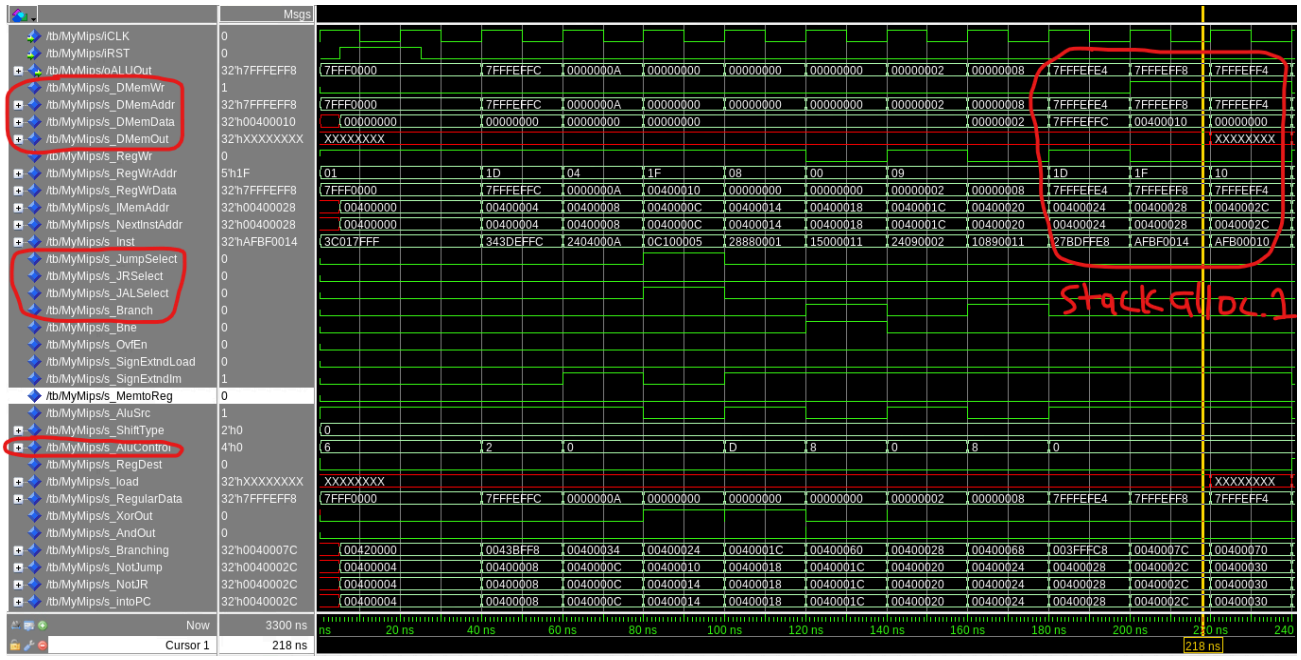
/tb/MyMips/iInstAddr	32'hXXXXXXXX	XXXXXXXX									
/tb/MyMips/iInstExt	32'hXXXXXXXX	XXXXXXXX									
/tb/MyMips/oALUOut	32'hFFFFFF00	00000001	00000004	00000005	0000000A	0000000D	00000005	00000004	00000004	00000001	FF000000	FFFFFF00
/tb/MyMips/s_DMemWr	0											
/tb/MyMips/s_DMemAddr	32'hFFFFFF00	00000001	00000004	00000005	0000000A	0000000D	00000005	00000004	00000004	00000001	FF000000	FFFFFF00
/tb/MyMips/s_DMemData	32'hFF000000	00000000	00000000	00000004	00000005	00000000	00000005	00000001		00000004	00000000	FF000000
/tb/MyMips/s_DMemOut	32'hXXXXXXXX	XXXXXXXX										
/tb/MyMips/s_RegWr	1											
/tb/MyMips/s_RegWrAddr	5'h0F	08	09	0A	0B	0C	0D	0E	08	09	0F	
/tb/MyMips/s_RegWrData	32'hFFFFFF00	00000001	00000004	00000005	0000000A	0000000D	00000005	00000004	00000004	00000001	FF000000	FFFFFF00
/tb/MyMips/s_IMemAddr	32'h00400028	00400000	00400004	00400008	0040000C	00400010	00400014	00400018	0040001C	00400020	00400024	00400028
/tb/MyMips/s_NextInstAddr	32'h00400028	00400000	00400004	00400008	0040000C	00400010	00400014	00400018	0040001C	00400020	00400024	00400028
/tb/MyMips/s_Inst	32'h000F7C03	24080001	24090004	01095020	014A5821	212C0009	016A6822	01A87023	00084080	00094882	3C0FFF00	000F7C03
/tb/MyMips/s_Halt	0											
/tb/MyMips/s_Ovfl	0											
/tb/MyMips/s_Dummy	0											
/tb/MyMips/s_JumpSelect	0											
/tb/MyMips/s_JRSelect	0											
/tb/MyMips/s_JALSelect	0											
/tb/MyMips/s_Branch	0											
/tb/MyMips/s_Bne	0											
/tb/MyMips/s_OvfEn	0											
/tb/MyMips/s_SignExtndLoad	0											
/tb/MyMips/s_SignExtndIm	0											
/tb/MyMips/s_MemtoReg	0											
/tb/MyMips/s_AluSrc	1											
/tb/MyMips/s_ShiftType	2'h2	0								1	0	2
/tb/MyMips/s_AluControl	4'h7	0					8		7		6	7
/tb/MyMips/s_RegDest	1											
/tb/MyMips/s_load	32'hXXXXXXXX	XXXXXXXX										
/tb/MyMips/s_RegularData	32'hFFFFFF00	00000001	00000004	00000005	0000000A	0000000D	00000005	00000004	00000004	00000001	FF000000	FFFFFF00

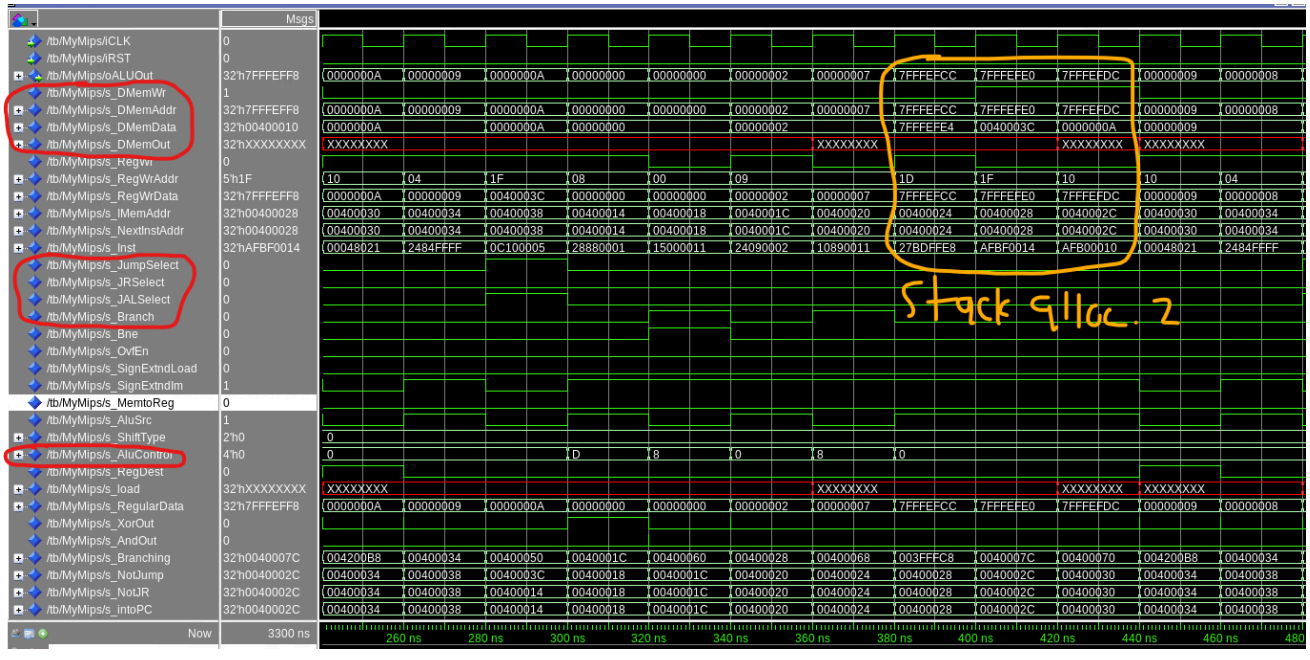
Now: 560 ns
Cursor 1: 239 ns
ns 20 ns 40 ns 60 ns 80 ns 100 ns 120 ns 140 ns 160 ns 180 ns 200 ns 220 ns 24
239 ns

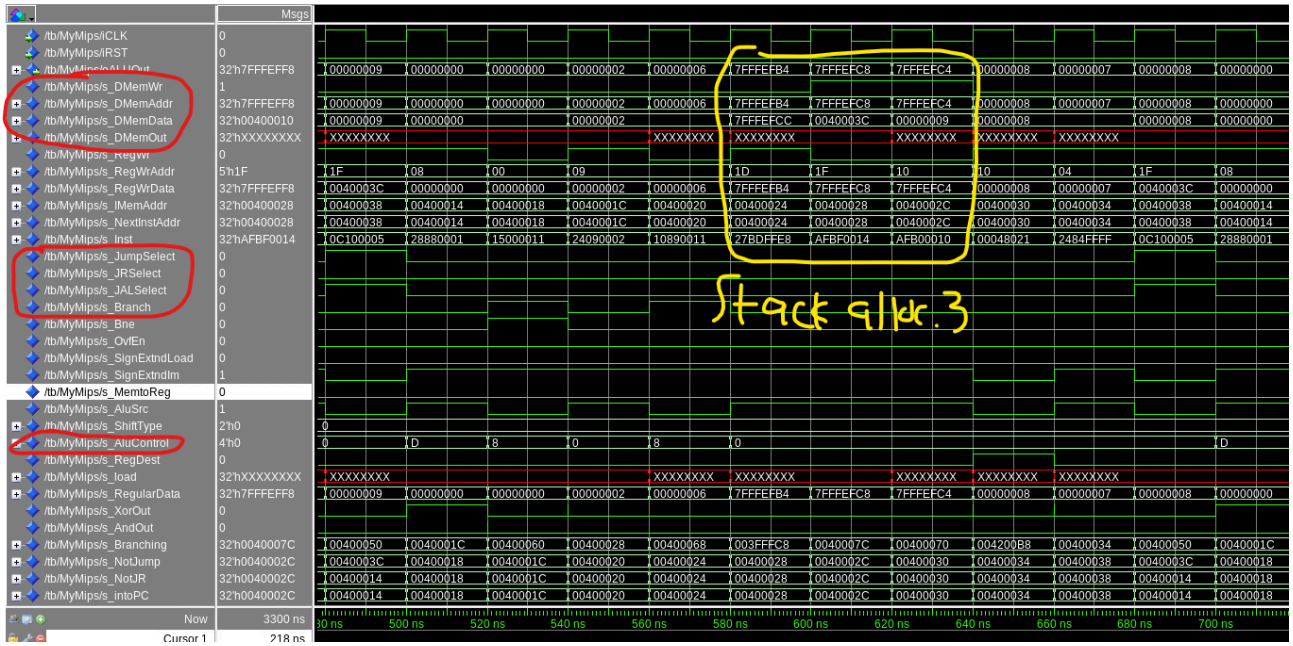


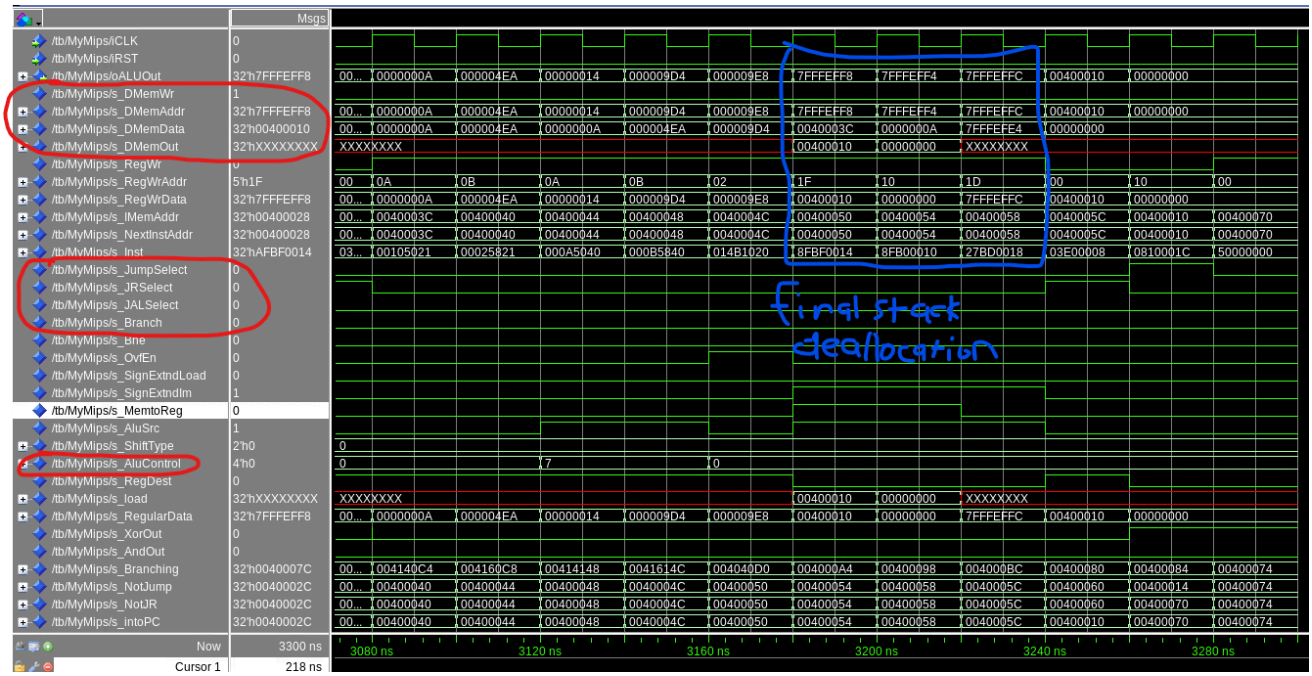
For our test Proj1_cf_test.s, we needed to use all control flow instructions to make a MIPS program that had a call depth of at least 5. Our program had a call depth of 9 and allocated 8 activation records on the stack, where our program is basically recursively calculating the factorial for the integer 10. Our control unit was able to effective decode what each instruction was and set the correct control signals (s_Branch, s_JumpSelect, s_JRSelect, s_JALSelect)for branching and jumping to update the PC with the correct address of the next instruction (beq, bne, j, jal, jr). Also, our control unit was also able to correctly decode each instruction and set the ALUControl signal to the correct operation for the ALU to perform. For storing each activation record in memory, that activation record was properly allocated with each item being stored in Dmem. Below is the

resulting waveform from our Proj1_cf_test.s test file. Due to the length of the waveform and how many dynamic instructions are being run, we will just be showing the first four procedures being called, the first three stacks being saved and allocated to Dmem, and the final deallocation of an activation record to restore the stack back to its original starting address.

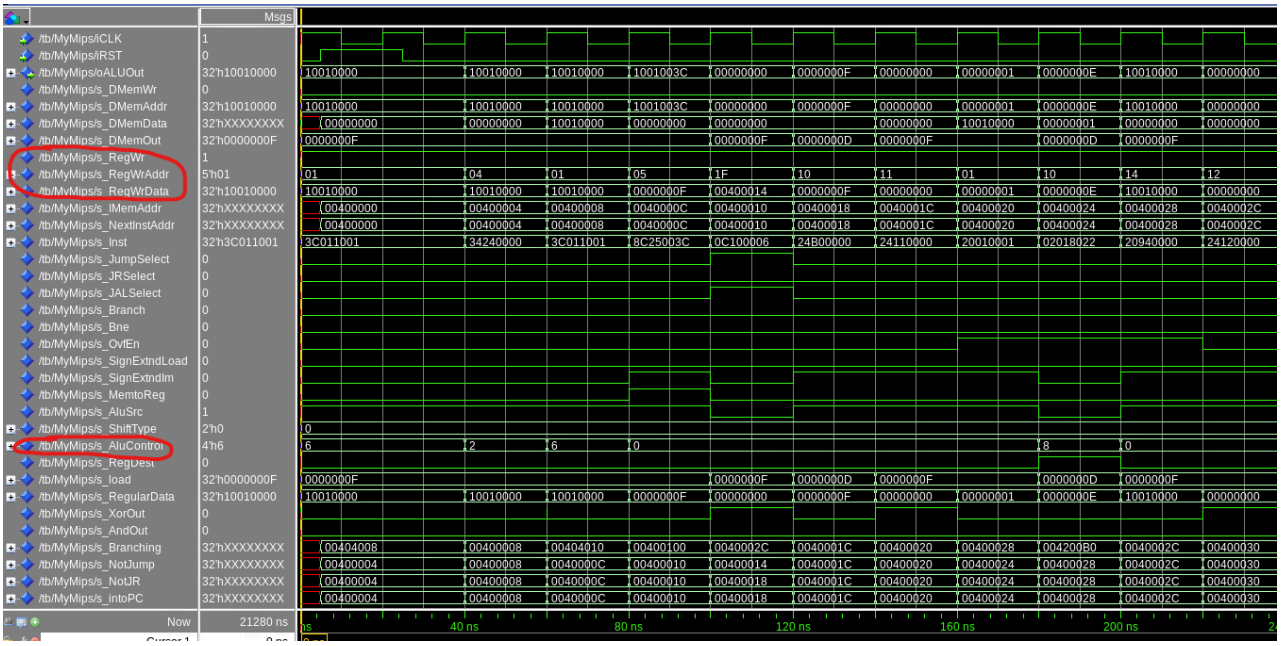


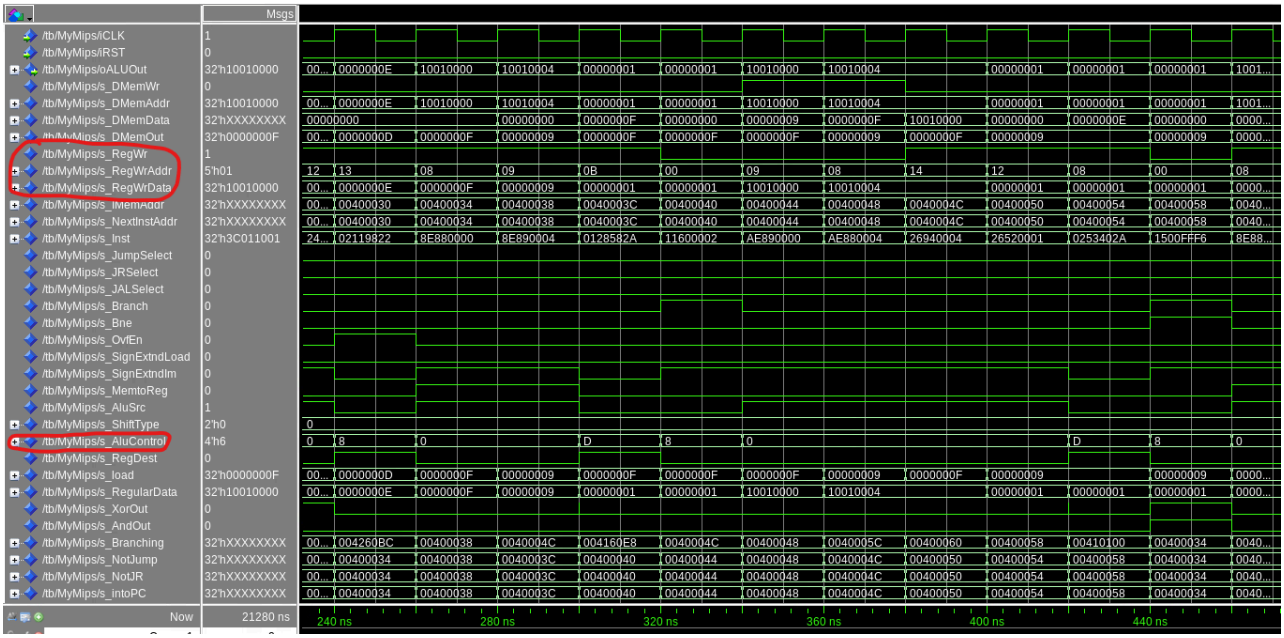


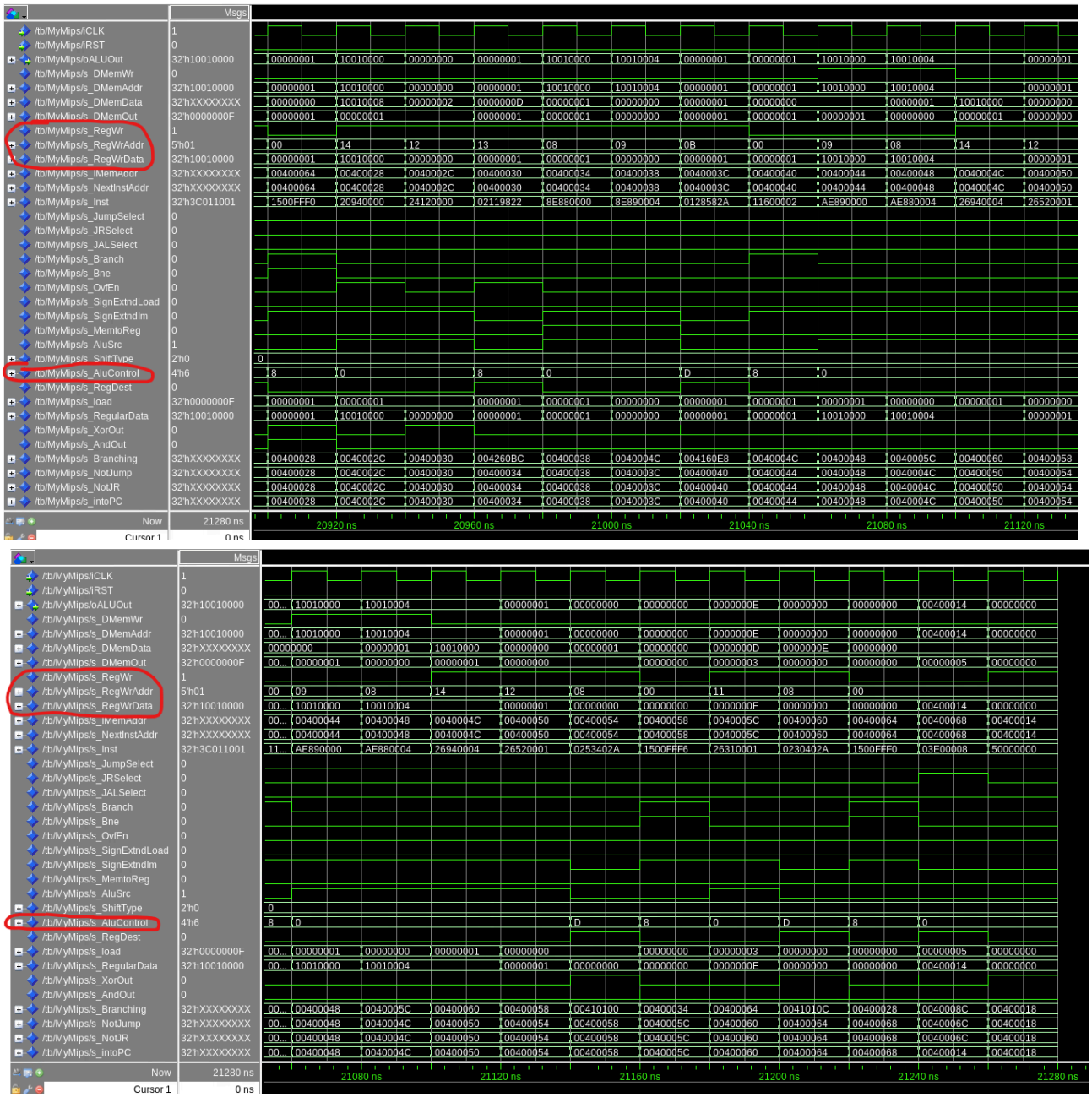




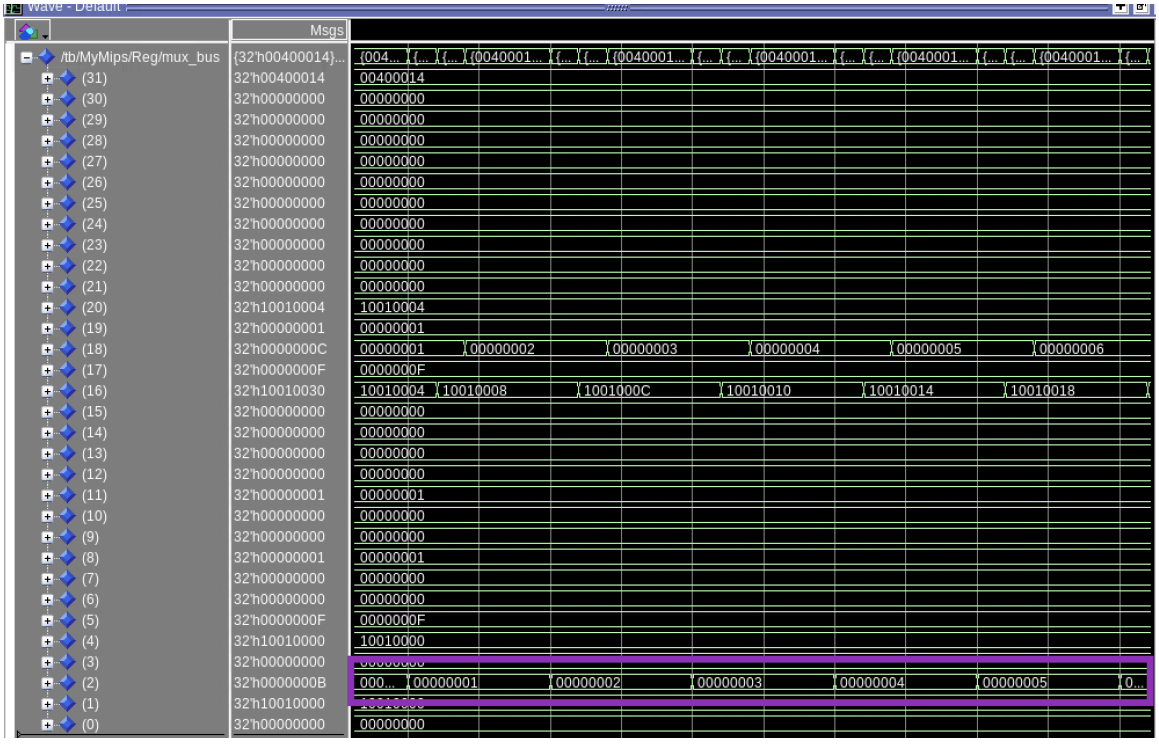
For our test Proj1_bubblesort.s, we needed to use control flow, arithmetic, and memory management instructions to implement the bubble sort algorithm. Once again, our control unit was able to decode the different instructions and send out the correct control signals for updating the PC for the next instruction, having the ALU perform the correct operation. Below is the resulting waveform from our Proj1_bubblesort_test.s test file. Due to the length of the waveform and how many dynamic instructions are being run, we will the beginning and end of the waveform, reflecting the start and end of the bubble sort algorithm.

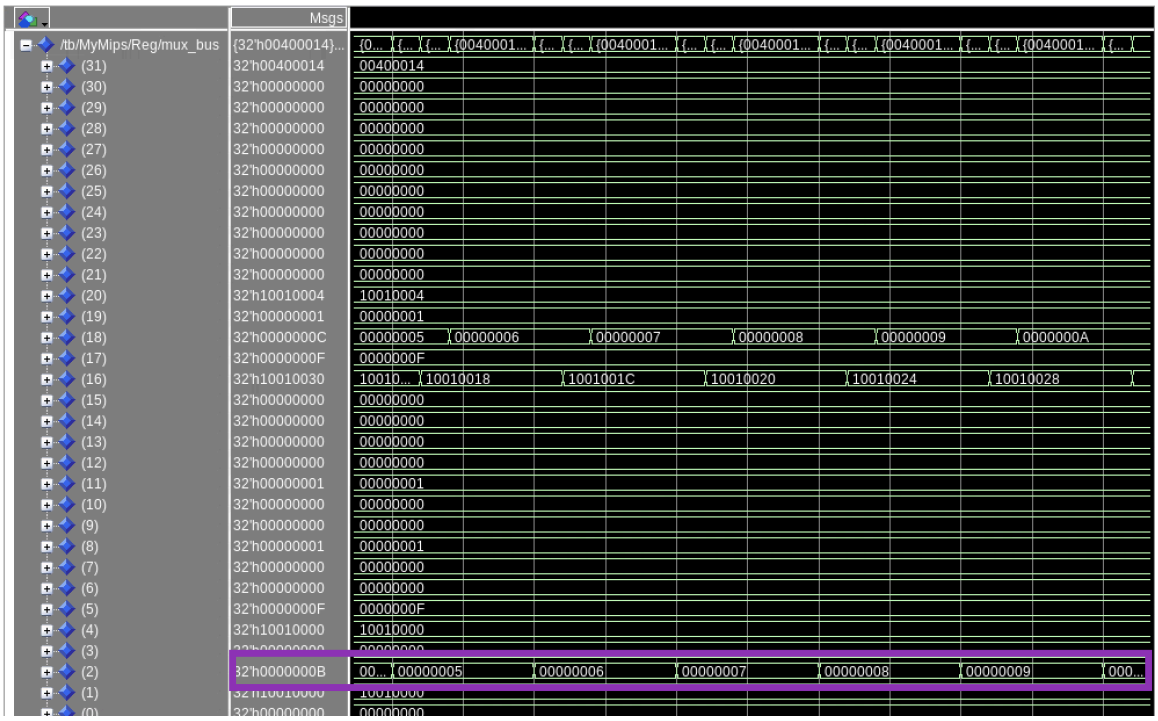

Msgs
/tb/MyMips/iCLK
/tb/MyMips/iRST
/tb/MyMips/oALUOut
/tb/MyMips/s_DMemWr
/tb/MyMips/s_DMemAddr
/tb/MyMips/s_DMemData
/tb/MyMips/s_DMemOut
/tb/MyMips/s_RegWr
/tb/MyMips/s_RegWrAddr
/tb/MyMips/s_RegWrData
/tb/MyMips/s_IMemAddr
/tb/MyMips/s_NextInstAddr
/tb/MyMips/s_Inst
/tb/MyMips/s_JumpSelect
/tb/MyMips/s_JRSelect
/tb/MyMips/s_JALSelect
/tb/MyMips/s_Branch
/tb/MyMips/s_Bne
/tb/MyMips/s_OvfEn
/tb/MyMips/s_SignExtndLoad
/tb/MyMips/s_SignExtndIm
/tb/MyMips/s_MemtoReg
/tb/MyMips/s_AluSrc
/tb/MyMips/s_ShiftType
/tb/MyMips/s_AluControl
/tb/MyMips/s_RegDest
/tb/MyMips/s_load
/tb/MyMips/s_RegularData
/tb/MyMips/s_XorOut
/tb/MyMips/s_AndOut
/tb/MyMips/s_Branching
/tb/MyMips/s_NotJump
/tb/MyMips/s_NotJR
/tb/MyMips/s_intoPC
Now
21280 ns
40 ns
80 ns
120 ns
160 ns
200 ns

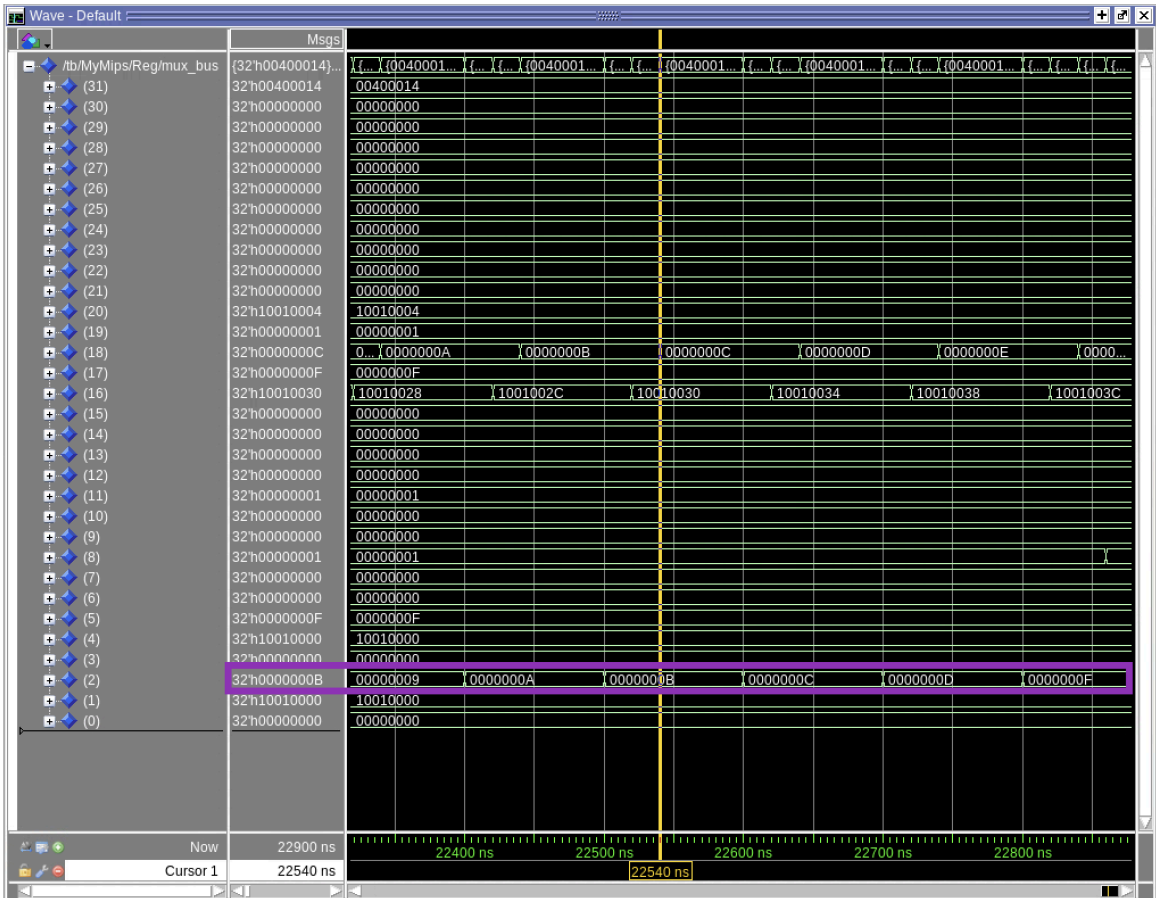

Msgs
/tb/MyMips/iCLK
/tb/MyMips/iRST
/tb/MyMips/oALUOut
/tb/MyMips/s_DMemWr
/tb/MyMips/s_DMemAddr
/tb/MyMips/s_DMemData
/tb/MyMips/s_DMemOut
/tb/MyMips/s_RegWr
/tb/MyMips/s_RegWrAddr
/tb/MyMips/s_RegWrData
/tb/MyMips/s_IMemAddr
/tb/MyMips/s_NextInstAddr
/tb/MyMips/s_Inst
/tb/MyMips/s_JumpSelect
/tb/MyMips/s_JRSelect
/tb/MyMips/s_JALSelect
/tb/MyMips/s_Branch
/tb/MyMips/s_Bne
/tb/MyMips/s_OvfEn
/tb/MyMips/s_SignExtndLoad
/tb/MyMips/s_SignExtndIm
/tb/MyMips/s_MemtoReg
/tb/MyMips/s_AluSrc
/tb/MyMips/s_ShiftType
/tb/MyMips/s_AluControl
/tb/MyMips/s_RegDest
/tb/MyMips/s_load
/tb/MyMips/s_RegularData
/tb/MyMips/s_XorOut
/tb/MyMips/s_AndOut
/tb/MyMips/s_Branching
/tb/MyMips/s_NotJump
/tb/MyMips/s_NotJR
/tb/MyMips/s_intoPC
Now
21280 ns
240 ns
280 ns
320 ns
360 ns
400 ns
440 ns



Below is going to be the end of bubble sort, which we chose to add a for loop to move our array one by one to $v0. This way, we can check the final output of our array and make sure it is correct. As you can see below highlighted in purple, it correctly gets the final values for our array sorted. This was done by just incrementing through the array and loading it from memory, so we know our bubble sort worked correctly!


Wave - Default
Msgs
/tb/MyMips/Reg/mux_bus
{32'h00400014}...
(31) 32'h00400014
(30) 32'h00000000
(29) 32'h00000000
(28) 32'h00000000
(27) 32'h00000000
(26) 32'h00000000
(25) 32'h00000000
(24) 32'h00000000
(23) 32'h00000000
(22) 32'h00000000
(21) 32'h00000000
(20) 32'h10010004
(19) 32'h00000001
(18) 32'h0000000C
(17) 32'h0000000F
(16) 32'h10010030
(15) 32'h00000000
(14) 32'h00000000
(13) 32'h00000000
(12) 32'h00000000
(11) 32'h00000001
(10) 32'h00000000
(9) 32'h00000000
(8) 32'h00000001
(7) 32'h00000000
(6) 32'h00000000
(5) 32'h0000000F
(4) 32'h10010000
(3) 32'h00000000
(2) 32'h0000000B
(1) 32'h10010000
(0) 32'h00000000
00400014
00000000
00000001
00000002
00000003
00000004
00000005
00000006
0000000F
10010004
10010008
1001000C
10010010
10010014
10010018
10010000


Msgs
/tb/MyMips/Reg/mux_bus
{32'h00400014}...
(31) 32'h00400014
(30) 32'h00000000
(29) 32'h00000000
(28) 32'h00000000
(27) 32'h00000000
(26) 32'h00000000
(25) 32'h00000000
(24) 32'h00000000
(23) 32'h00000000
(22) 32'h00000000
(21) 32'h00000000
(20) 32'h10010004
(19) 32'h00000001
(18) 32'h0000000C
(17) 32'h0000000F
(16) 32'h10010030
(15) 32'h00000000
(14) 32'h00000000
(13) 32'h00000000
(12) 32'h00000000
(11) 32'h00000001
(10) 32'h00000000
(9) 32'h00000000
(8) 32'h00000001
(7) 32'h00000000
(6) 32'h00000000
(5) 32'h0000000F
(4) 32'h10010000
(3) 32'h00000000
(2) 32'h0000000B
(1) 32'h10010000
(0) 32'h00000000
00400014
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
00000009
0000000A
0000000F
10010004
10010018
1001001C
10010020
10010024
10010028
10010000



[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.

In this test application, we utilized all of the arithmetic and logical instructions to make sure our processor design could handle the combination of the different arithmetic and logical operations that can be performed in MIPS. Below is the code that we developed for this test case. This code will also be included in our submission.

```
# MIPS file to test all arithmetic and logical instructions with our processor design
.data
.text
.globl main

main:

# Addition instructions
addiu $t0, $zero, 1     # $t0 = 1 = 0x1
```

```
addiu $t1, $zero, 4     # $t1 = 4 = 0x4
add $t2, $t0, $t1       # $t2 = 1 + 4 = 5 = 0x5
addu $t3, $t2, $t2      # $t3 = 5 + 5 = 10 = 0xA
addi $t4, $t1, 9        # $t4 = 4 + 9 = 13 = 0xD

# Subtraction instructions
sub $t5, $t3, $t2       # $t5 = 10 - 5 = 5 = 0x5
subu $t6, $t5, $t0      # $t6 = 5 - 1 = 4 = 0x4

# Shift instructions
sll $t0, $t0, 2         # $t0 = 1 * 2^2 = 1 * 4 = 4 = 0x4
srl $t1, $t1, 2         # $t1 = 4 / 2^2 = 4 / 4 = 1 = 0x1
lui $t7, 0xFF00         # $t7 = 0xFF000000
sra $t7, $t7, 16        # $t7 = 0xFFFFFF00

sllv $t2, $t2, $t1      # $t2 = 5 * 1^2 = 5 * 2 = 10 = 0xA
addiu $t3, $zero, 16    # $t3 = 0 + 16 = 16 = 0x10
srlv $t3, $t3, $t1      # $t3 = 16 / 1^2 = 16 / 2 = 6 = 0x8
srav $t7, $t7, $t3      # $t7 = 0xFFFFFFFF

# Set less than instructions
slt $s0, $t1, $t0       # $s0 = 1 = 0x1
slti $s1, $t3, 4        # $s1 = 0 = 0x0

addiu $t0, $zero, 0x1001  # $t0 = 0x00001001
addiu $t1, $zero, 0x1111  # $t1 = 0x00001111

# Bitwise instructions
and $t2, $t0, $t1       # $t2 = 0x00001001
andi $t3, $t0, 0x0000   # $t3 = 0x0

nor $t4, $t0, $t1       # $t4 = 0xFFFFEEEE
xor $t5, $t0, $t1       # $t5 = 0x00000110
xori $t4, $t2, 0x1010   # $t5 = 0x00000011

or $t6, $t2, $t3        # $t6 = 0x00001001
ori $t7, $t5, 0x11      # $t7 = 0x00000111

halt
```

[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.

When creating a test application that used all of the control flow instructions and had 5+ call depth, we decided to write a recursive function that calculated the factorial in a

modified way (not calculating the factorial of the input correctly) with an input (n = 10) that would give us a call depth of 9, and a total of 8 activation records being allocated and deallocated from the stack. This is the exact recursive function that we wrote in its equivalent C code:

```
int fact (int n)
{
    if (n < 1) return (1);
    else if (n == 2) return (2);
    else return (n * fact(n - 1));
}
```

Below is the MIPS-equivalent code that implements this function and makes up this test application

```
# MIPS file to test all control flow instructions
.data
.text
.globl main


#set stack pointer to be correct
li $sp, 0x7fffeffc

main:
li $a0, 10  # set argument n = 10 = 0xA --> result in a call depth of 9, 8 activation records allocated on the stack
jal Fact    # call the fact procedure for the first time
j Exit

Fact:
# base case 1
slti $t0, $a0, 1       # check if n < 1
bne $t0, $zero, Base1  # enter the first base case if n < 1

# base case 2
addiu $t1, $zero, 2    # $t1 = 2 = 0x2
beq $a0, $t1, Base2    # enter the second base case if n = 2

# recursive case
addiu $sp, $sp, -24   # allocate an activation record
sw $ra, 20($sp)       # save the return address
sw $s0, 16($sp)       # save $s0, which holds n

addu $s0, $zero, $a0  # $s0 = n
```

```
addiu $a0, $a0, -1    # change argument to n - 1
jal Fact              # recursively call fact(n - 1)

addu $t2, $zero, $s0  # $t2 = n
addu $t3, $zero, $v0  # $t3 = fact(n-1)
sll $t2, $t2, 1       # $t2 = n * 1^2 = n * 2
sll $t3, $t3, 1       # $t3 = fact(n-1) * 1^2 = fact(n-1) * 2
add $v0, $t2, $t3     # $v0 = n * fact(n-1) --> final value of $v0 = 0x000009e8

lw $ra, 20($sp)       # restore the return address
lw $s0, 16($sp)       # restore the current value of n
addiu $sp, $sp, 24    # deallocate the activation record
jr $ra

Base1:
addiu $v0, $zero, 1   # return 1
jr $ra

Base2:
addiu $v0, $zero, 2   # return 2
jr $ra

Exit:                 # end the program
halt
```

[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.

This test application will be included in the submission. I will also paste it here for convenience.

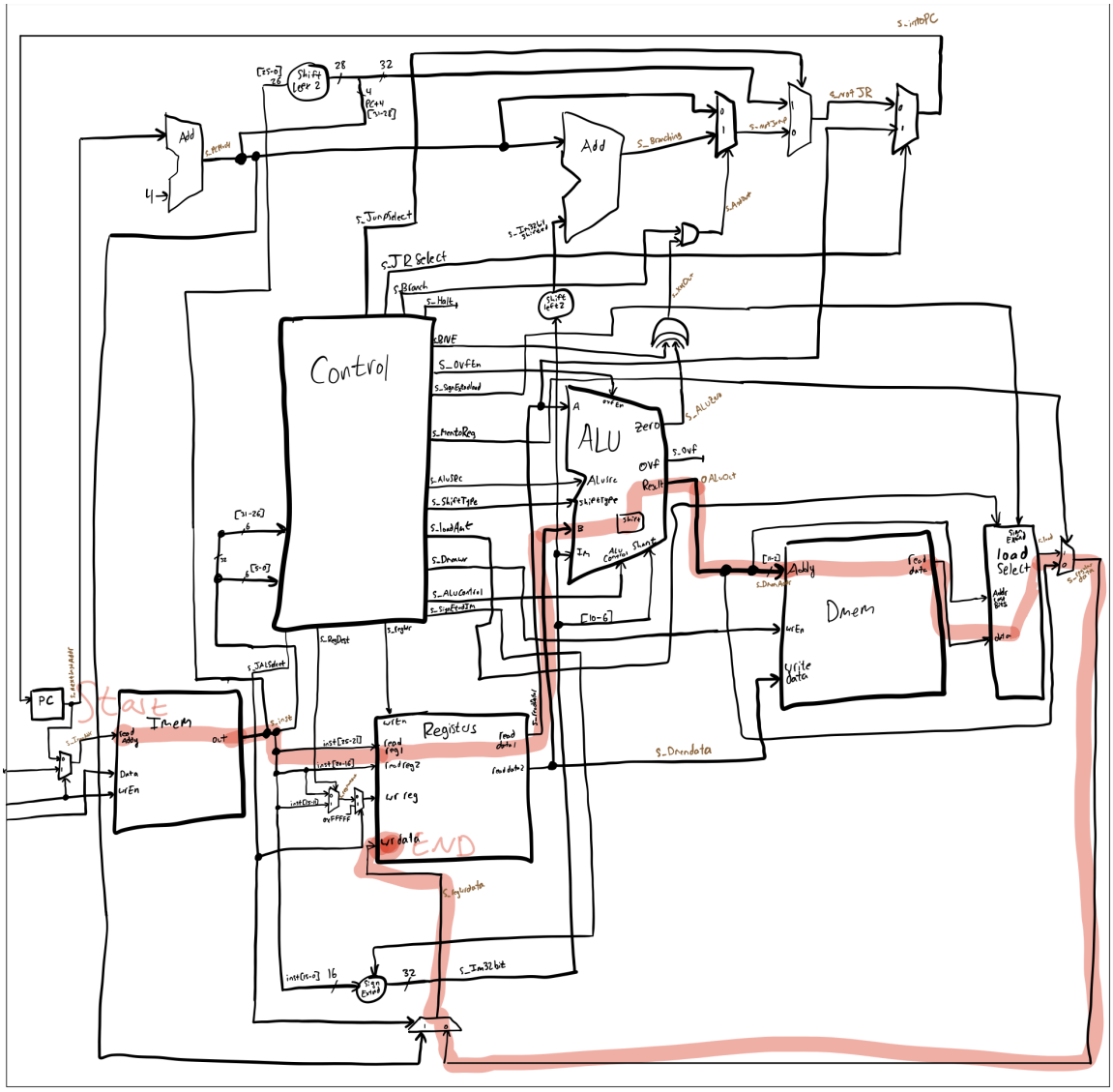
```
.data
#Array
christopher: 0xF, 0x9, 0x8, 0xD, 0x3, 0x1, 0x6, 0xC, 0x4, 0x2, 0x5, 0x7, 0xA, 0xB, 0x0
length: 15
.text
.globl main
main:

la $a0, christopher #Load our address for christopher to be passed into call
lw $a1, length
jal BubbleSort#Call Bubble Sort
j display#one by one load into a register to show questasim works
```

```
BubbleSort:
addiu $s0, $a1, 0#Get the length
addiu $s1, $zero, 0#Our counter for the outer loop.
subi $s0, $s0, 1#Sub 1 for array loop purposes


ForLoopOuter:
addi $s4, $a0, 0#Set our array start to S4
addiu $s2, $zero, 0#Each time outer loop is called we would reset the inner loop.
sub $s3, $s0, $s1#Our inner loop stop value changes.


ForLoopInner:
lw $t0, ($s4)#load array @s2 position in array
lw $t1, 4($s4)#load array @s2+1
#Compare S2 to S2+1
slt $t3, $t1, $t0#if S2 is Greater Perform a swap else move on with loop.
beq $t3, $zero, NoSwap#If 0 we dont need to swap

#Swap out elements
sw $t1, ($s4)
sw $t0, 4($s4)

NoSwap:
#inner loop cond
addiu $s4, $s4, 4#increment array pos
addiu $s2, $s2, 1#incremeout our counter for Inner
slt $t0, $s2, $s3 #if we are less ,this is 1.
bne $t0, $zero, ForLoopInner#if 1, call the loop.


#outerloop cond
addiu $s1, $s1, 1#incremeout our counter for Outer
slt $t0, $s1, $s0 #if we are less ,this is 1.
bne $t0, $zero, ForLoopOuter#if 1, call the loop.
jr $ra#return

display:
#la $a0, christopher #Load our address for christopher to be passed into call
#lw $a1, length
la $s0, christopher#Load array again
lw $s1, length#Load length again
addiu $s2, $zero, 0#Counter

displayLoop: #$v0 displays final valyes one by one
lw $v0, ($s0) #load array @position in array
```

```
addiu $s0, $s0, 4#increment array pos
addiu $s2, $s2, 1#incremeout our counter for Inner
slt $t0, $s2, $s1 #if we are less ,this is 1.
bne $t0, $zero, displayLoop#if 1, call the loop.


halt
```

[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?

Max Frequence: **20.78 mhz**



Our critical path is starting at Imem, going into the registers, and then into the ALU component to Shift left logical. Our design has the Shifter in the ALU component, and then the output of that maps into Dmem, then to our load selector (used to select lw or lb. Not needed here but still passes through) and then into 2 control logic mux and ends up back into the write of the register file. Essentially, shifting is our critical path.

One component that we could change to improve the frequency is the **shifter component** within the ALU. The shifter is one of the components that we designed that takes the longest time to execute within the critical data path. To improve the shifter, we could split the shifter functionality into three separate components instead of just having one large shifter component that includes all of the shifting functionality. By splitting the shifting component up into several components, the critical path won't have to go through as many MUXes every single time a shift operation needs to be performed within the circuit. By changing and optimizing our shifter, we can compensate for the amount of propagation delay the instruction and data memory components have, which will result in a better slack time and better frequency performance.