CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

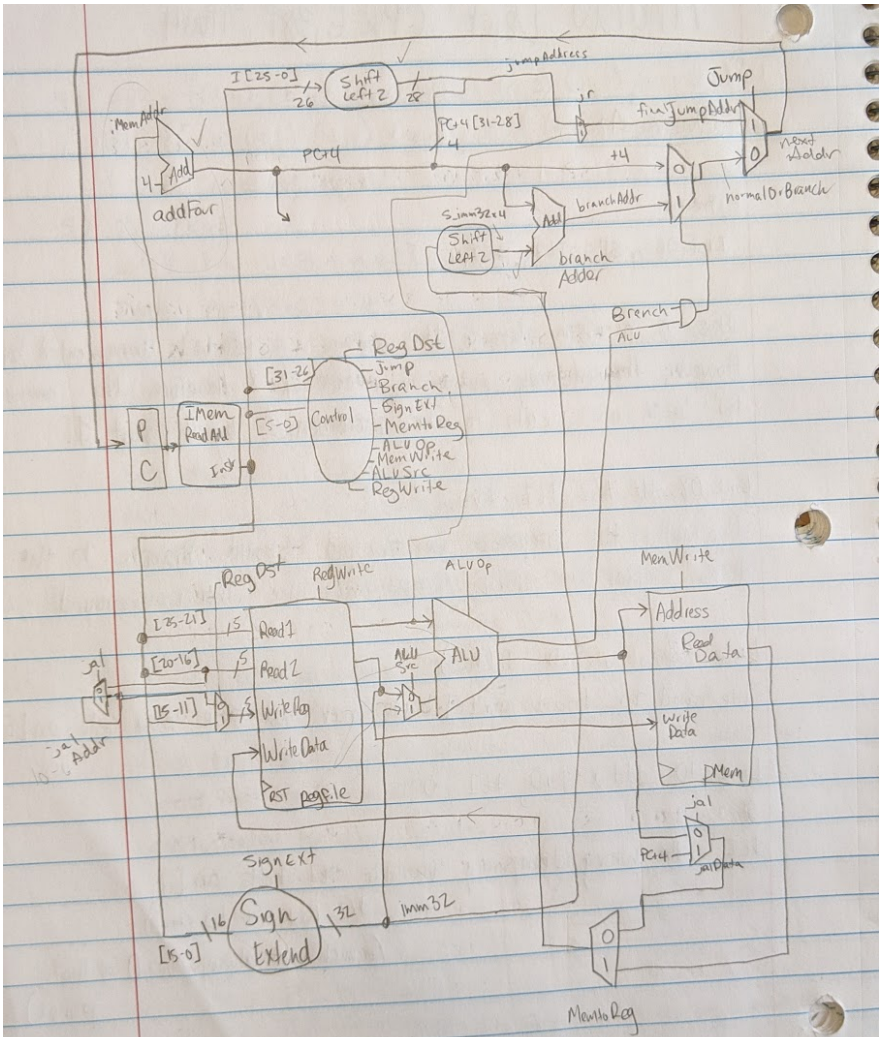
Team Members: _____John Brose_____

_____Andrew Deick_____

_____Rolf Anderson_____

Project Teams Group #:_____2-4_____

[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.*

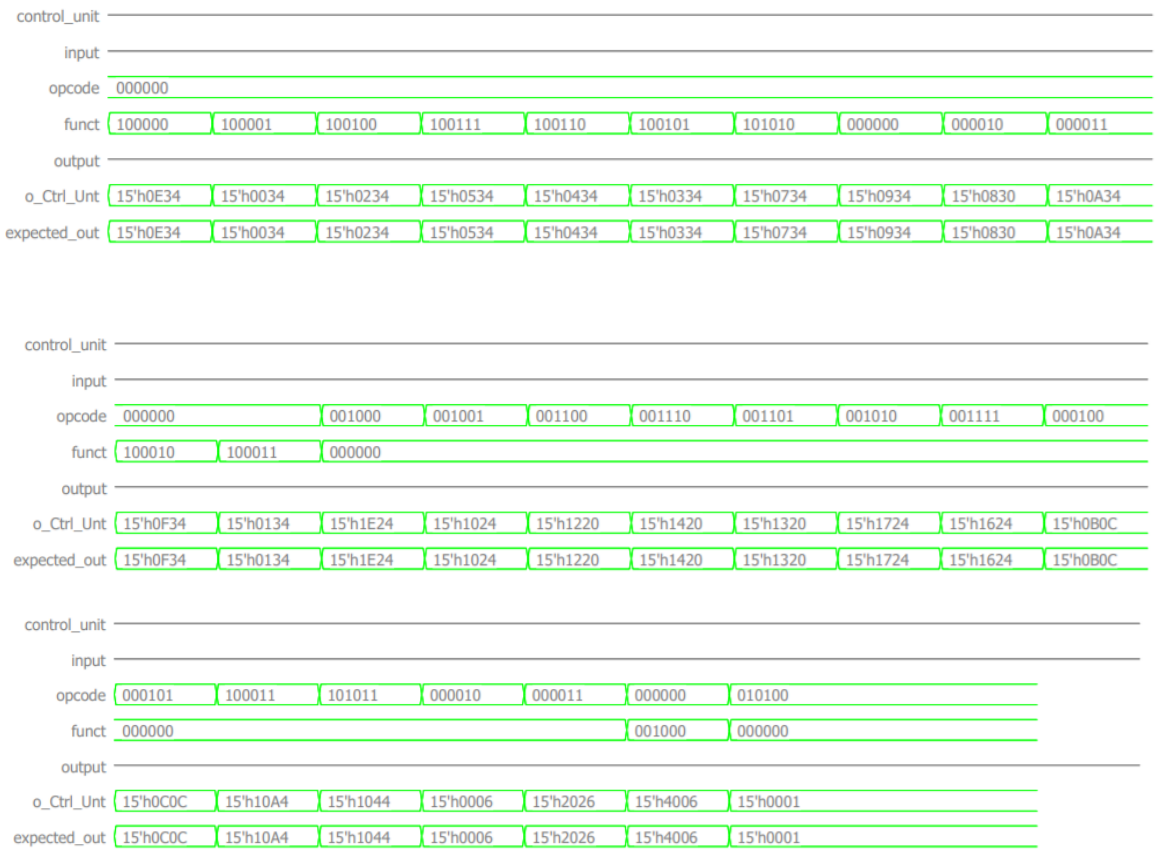
	Instruction	Opcode (Binary)	Funct (Binary)	Control Signals									
				jr [jr mux]	jal [jal mux]	MemtoReg	MemWrite (we_mem)	RegWrite (we_reg)	RegDst	Branch (PCSrc)	SignExt [determines sign extension on 16 to 32 bit extender]	j [jump]	
	I-TYPE												
	addi	"001000"	"------"	0	0	0 [addi does NOT read from memory]	0 [addi does NOT write to memory]	1 [addi writes back to a register]	0 [addi uses rt as destination register rather than rd]	0 [use PC+4 no branch]	1 [addi sign extended]	0 [no jump]	
	addiu	"001001"	"------"	0	0	0 [addiu does NOT read from memory]	0 [addiu does NOT write to memory]	1 [addiu writes back to a register]	0 [addiu uses rt as destination register rather than rd]	0 [use PC+4 no branch]	1 [addi sign extended]	0 [no jump]	
	andi	"001100"	"------"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use PC+4 no branch]	0 [zero extended]	0 [no jump]	
	xori	"001110"	"------"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use PC+4 no branch]	0 [zero extended]	0 [no jump]	
	ori	"001101"	"------"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use PC+4 no branch]	0 [zero extended]	0 [no jump]	
	slti	"001010"	"------"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use PC+4 no branch]	1 [sign extended]	0 [no jump]	
	lui	"001111"	"------"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]	
	beq	"000100"	"------"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to a register]	0 [does not matter]	1 [use branch if needed]	1 [does not matter but choose one because more sign extensions]	0 [no jump]	
	bne	"000101"	"------"	0	0	0 [does NOT read from memory]	0 [does NOT write to memory]	0 [does NOT write to a register]	0 [does not matter]	1 [use branch if needed]	1 [does not matter but choose one because more sign extensions]	0 [no jump]	
	lw	"100011"	"------"	0	0	1 [reads from memory]	0 [does NOT write to memory]	1 [writes back to a register]	0 [uses rt as destination register rather than rd]	0 [use PC+4 no branch]	1 [sign extended]	0 [no jump]	
	sw	"101011"	"------"	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]	
	R-TYPE												
	add	"000000"	"100000"	0	0	0 [add does NOT read from memory]	0 [add does NOT write to memory]	1 [add writes back to a register]	1 [add uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]	
	addu	"000000"	"100001"	0	0	0 [addu does NOT read from memory]	0 [addu does NOT write to memory]	1 [addu writes back to a register]	1 [addu uses rd as destination]	0 [use PC+4 no branch]	1 [does not matter but choose one because more sign extensions]	0 [no jump]	
	and	"000000"	"100100"	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 extensions]	0 [no jump]	
	nor	"000000"	"100111"	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 extensions]	0 [no jump]	
	xor	"000000"	"100110"	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 extensions]	0 [no jump]	
	or	"000000"	"100101"	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 extensions]	0 [no jump]	
	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 extensions]	0 [no jump]	
	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 extensions]	0 [no jump]	
	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]	0 [don't think it matters? zero extended]	0 [no jump]	
	sra	"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 extensions]	0 [no jump]	
	sub	"000000"	"100010"	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 extensions]	0 [no jump]	
	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 extensions]	0 [no jump]	
	jr	"000000"	"001000"	1	0	0 [does not matter]	0 [does NOT write to memory]	0 [does not matter]	0 [does not matter]	0 [does not matter]	1 [does not matter but choose one because more sign extensions]	1 [jump]	
	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]	1 [does not matter but choose one because more sign extensions]	1 [jump]	
	jal	"000011"	"------"	1	1	0 [does NOT read from memory]	0 [does NOT write to memory]	1	0 [does not matter]	0 [does not matter]	1 [does not matter but choose one because more sign extensions]	1 [jump]	

[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).*

Below is the format for each instruction of the control unit, the coding mainly consisted of with select statements (shown below). The tb output waveform compares the control_unit.vhd outputs with the expected outputs derived from our spreadsheet from part 2a.

```
30  ------------------------- FORMAT of o_Ctrl_Unt ----------------------------
31  -- " 14     13      12       11-8        7        6      5      4       3       2    1   0
32  -- " 0      0       0        0000        0        0      1      1       0       1    0"  0
33  -- " jr     jal     ALUSrc   ALUControl  MemtoReg we_mem we_reg RegDst  PCSrc   SignExt j" Halt
34  --                                                                     Branch
35  ------------------------------------------------------------------------------
```

```
37  with i_funct select s_RTYPE <=
38      "0001110001101 00"  when "100000", -- add
39      "0000000001101 00"  when "100001", -- addu
40      "0000010001101 00"  when "100100", -- and
41      "0000101001101 00"  when "100111", -- nor
42      "0000100001101 00"  when "100110", -- xor
43      "0000011001101 00"  when "100101", -- or
44      "0000111001101 00"  when "101010", -- slt
45      "0000111001101 00"  when "101011", -- sltu
46      "0001001001101 00"  when "000000", -- sll
47      "0001000001100 00"  when "000010", -- srl
48      "0001010001101 00"  when "000011", -- sra
49      "0001111001101 00"  when "100010", -- sub
50      "0000001001101 00"  when "100011", -- subu
51      "1000000000001 10"  when "001000", -- jr
52      "0000000000000 00"  when others;
53
54  with i_opcode select o_Ctrl_Unt <=
55      s_RTYPE         when "000000", -- RTYPE
56      "0011110001001 00"  when "001000", -- addi
57
58      "0000000000000 01"  when "010100", -- halt
59
60      "0010000001001 00"  when "001001", -- addiu
```



[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.*
There are 27 unique instructions: addi, addiu, andi, xori, ori, slti, lui, beq, bne, lw, sw, add, addu, and, nor xor, or, slt, sll, srl, sra, sub, subu, jr, j, jal & halt.

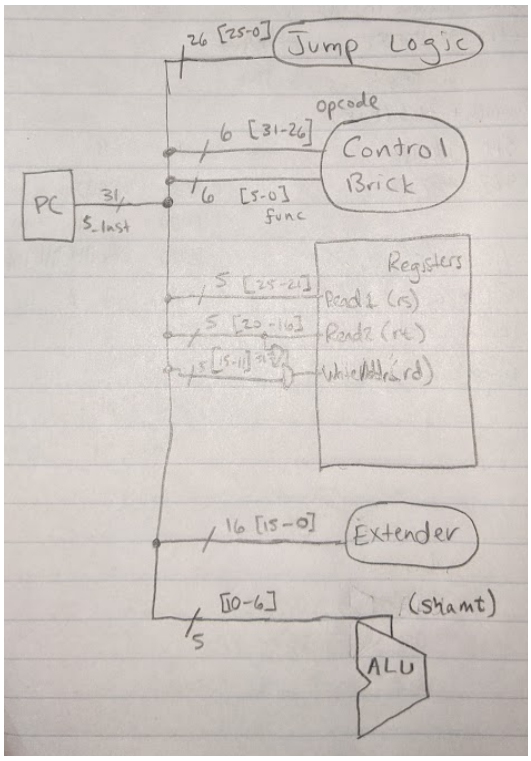
All of these instructions need to have separate control states. Many of the I-type instructions are exactly the same as J-type counterparts except that ALUSrc is set to receive an immediate value instead of one from the register.

There are two control signals that control the addressing: j and Branch. Branch requires a specific ALU output, while j (short for jump) is unconditional. The jump register can receive an address from a register or from a 26 bit immediate. (This mux is controlled by the jr signal). Perhaps misleadingly, the jal control signal writes the current address (+4) to register 31, and the signal itself is not at all related to addressing. (The jal instruction enables the j signal).

The remaining signals include two Write Enables (MemWrite and RegWrite) and two pathing signals (RegDst and MemtoReg), which, respectively, determine if rs or rd should decide the final register write destination and whether the memory or the ALU should write to the register.

Last but not least, we have SignExtend that is the only bit control signal that does not correspond with a Mux. It is fed into the Extended component to determine if the immediate should be sign-extended or zero-extended.

[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?*

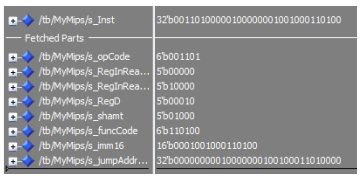


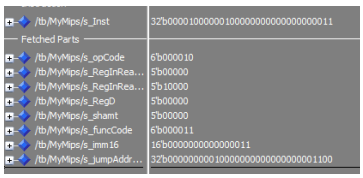
Instruction Fetch (left) requires a 32 bit signal from the PC. As seen in the picture, the processor splits the instruction into several more manageable pieces, which it then handles individually. The instructions are always connected to their respective components, and the output of the component is ignored when necessary.

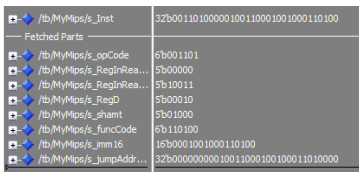
For example, the jump address is calculated every instruction and then ignored via control signals.

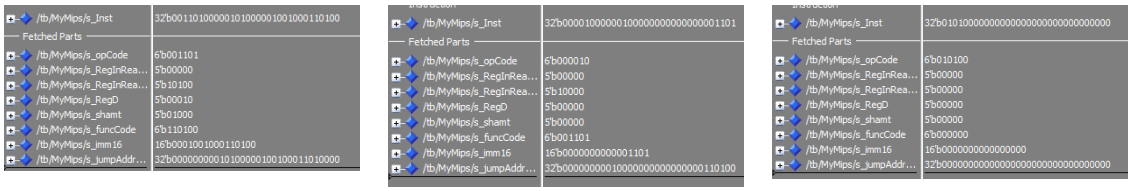
Speaking of Control Signals, we have 15 bits worth. Four of the bits correspond to ALU Opcode, which is determined from the Opcode or Function Code. We have jr, jal, ALUSrc, MemtoReg, MemWrite, RegWrite, RegDst, Branch, SignExt, j, & Halt.

[Part 2 (b.iii)] *Implement your new instruction fetch logic using VHDL. Use Modelsim 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 Modelsim waveforms in your writeup.*









In addition to the instructions tested above, every instruction was tested individually to ensure that the fetch worked as expected. These cover every control flow possibility, because literally every instruction is accounted for.

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

Srl and sra are both shift right operations, which move bits from left to right. Srl shifts logical, meaning no matter what bit 31 holds, the added bits will be zeros. Sra is arithmetic, meaning if bit 31 is a one, then ones will be used to shift in, otherwise zeros will. Arithmetic allows the shifting of negative numbers by keeping the number negative but still shifting bits. Sla is not needed since bit 0 does not affect the sign of the number and thus shifting in ones would hold no distinct advantage.

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

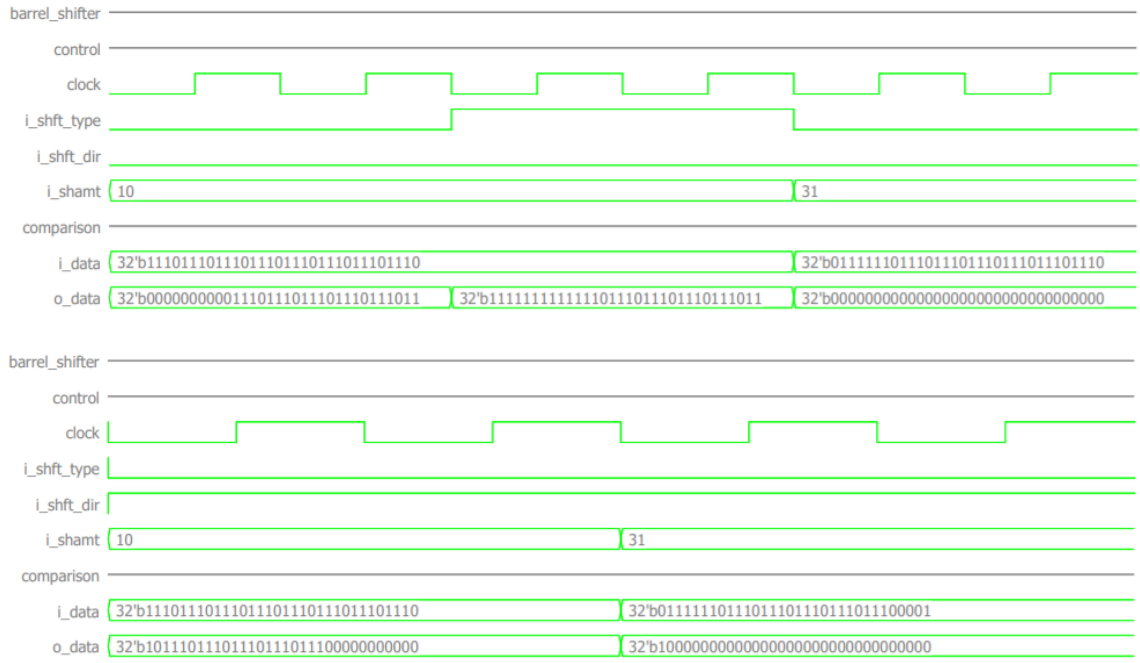
Our VHDL code implements arithmetic and logical operations but having a control bit to a 2 to 1 mux that selects whether the added bits to the input should be 0 or input(31). This allows for shifting in strictly zeros or shifting between 0's and 1's based on whether the number is positive or negative.

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

In order to add left shifting operations the only thing that needs to be added is two simple 2 to 1 muxs. The first mux selects between the normal input and the left shift input where bit 31 becomes bit 0, bit 0 becomes 31, bit 30 becomes bit 1, etc until the input is completely flipped around. Then at the output, another mux is used to either keep the normal right shift output, or flip the left shift output back around to be left shifted. Lastly, the control bit for logical versus arithmetic needs to be always set to zero for srl.

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

Below shows the output from the barrel shifter test bench. I_shift_type dictates logical(0) or arithmetic(1). I_shift_dir dictates shift right(0) versus shift left(1). Finally, i_shamt controls shift amount. As can be seen in the waveforms when srl is done no 1's are shifted in, but when sra is used for the same number 1's are. Shifting left is also shown to work once i_shift_dir changes to one.

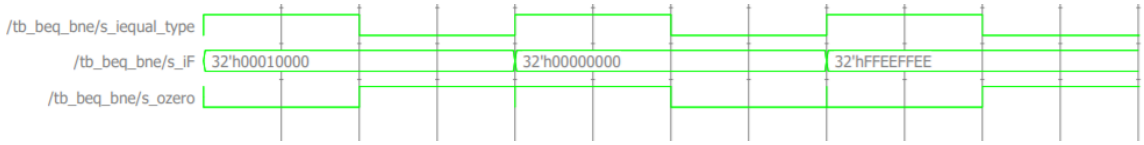


[Part 2 (c.ii.1)] *In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.*

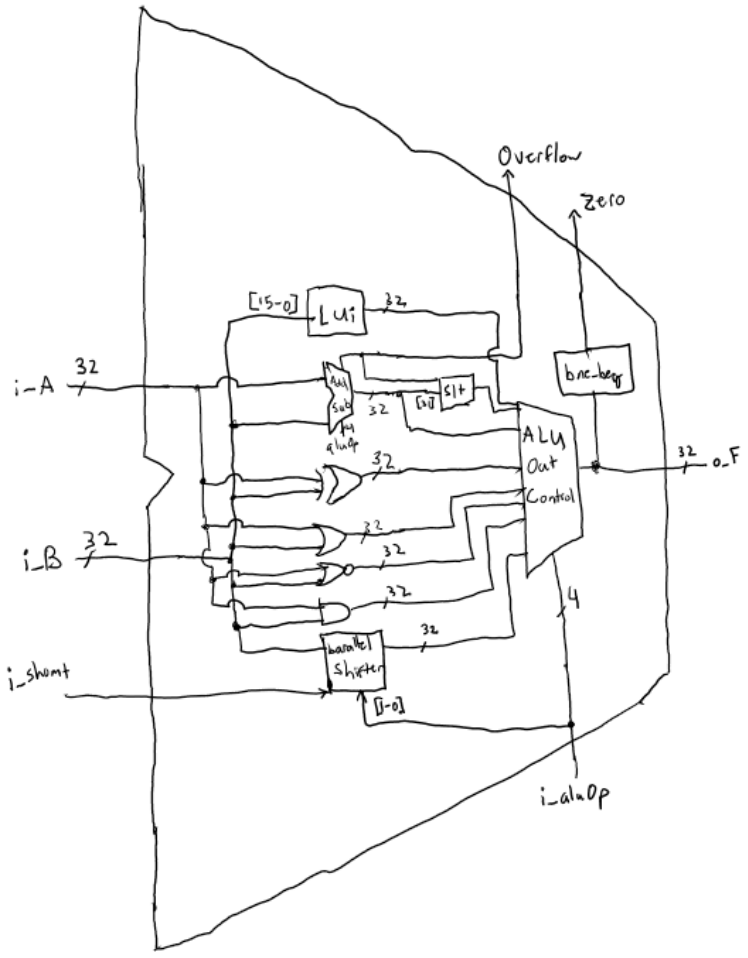
The design approach of the ALU mainly involved breaking down the functionality into smaller parts based on each opcode needed and then combining those parts into one big file. The first hurdle was deciding on whether to implement all functionality as separate blocks or directly implement them inside the ALU. After discussing with one another it was decided that we would have a mix. The xor, or, and, nor, and lui were all implemented inside the ALU since their implementation was quick and easy using modelsims logic operations within processes and setting portions of inputs to outputs (for lui). The harder to implement units like the adder_subtractor, barrel_shifter, and beq_bne were implemented separately since they were more complicated and in order for ease of readily inside the ALU they were created and tested separately.

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

The only other components that were tested separately were the adder_subtractor unit and the beq_bne unit. The adder_subtractor unit was extensively tested in previous labs and only an overflow needed to be added to it so the unit was not tested again. The beq_bne unit was tested by taking a single 32 bit input (assuming that it was already xored by the alu) and output a 1 based on the value of is_equal_type where (iF == 0) when 1 and (iF != 0) when 0 as can be seen below.



[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?*

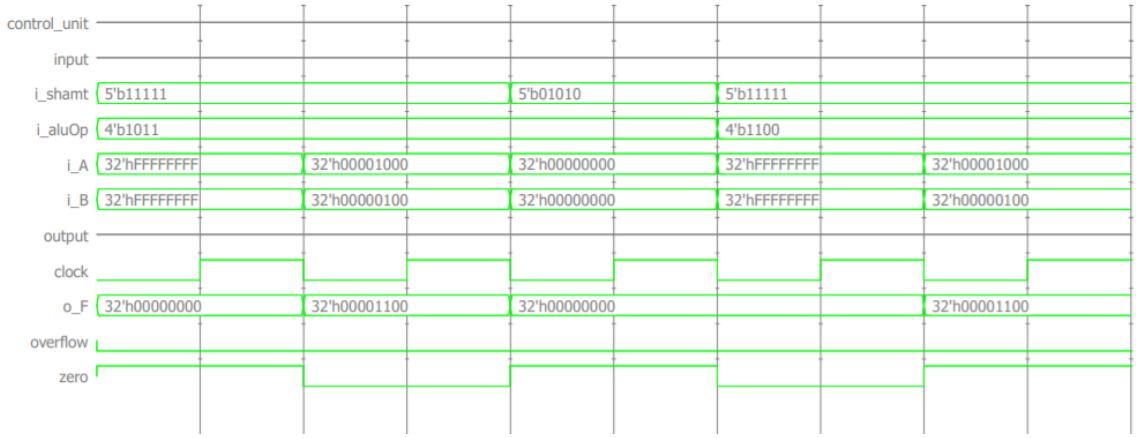


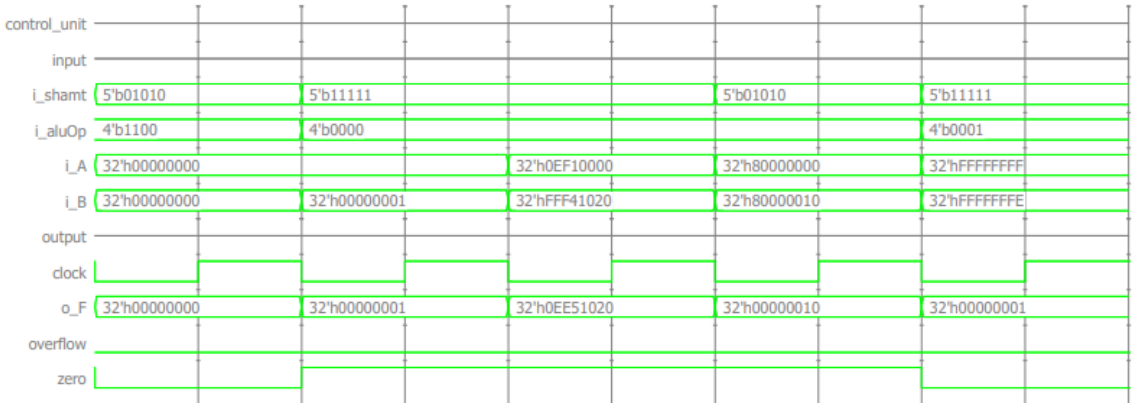
Slt was calculated using the subtraction of i_B from i_A and then taking the signed bit of the output. In order to account for overflow bit 31 of the add_sub unit output was anded with the overflow bit as shown. Zero was calculated by taking the xor at the output of the alu_out_control and then using an or tree to see if it was equal to zero. Finally, the overflow was calculated inside the add_sub unit by xoring Cout(31) with Cout(30).

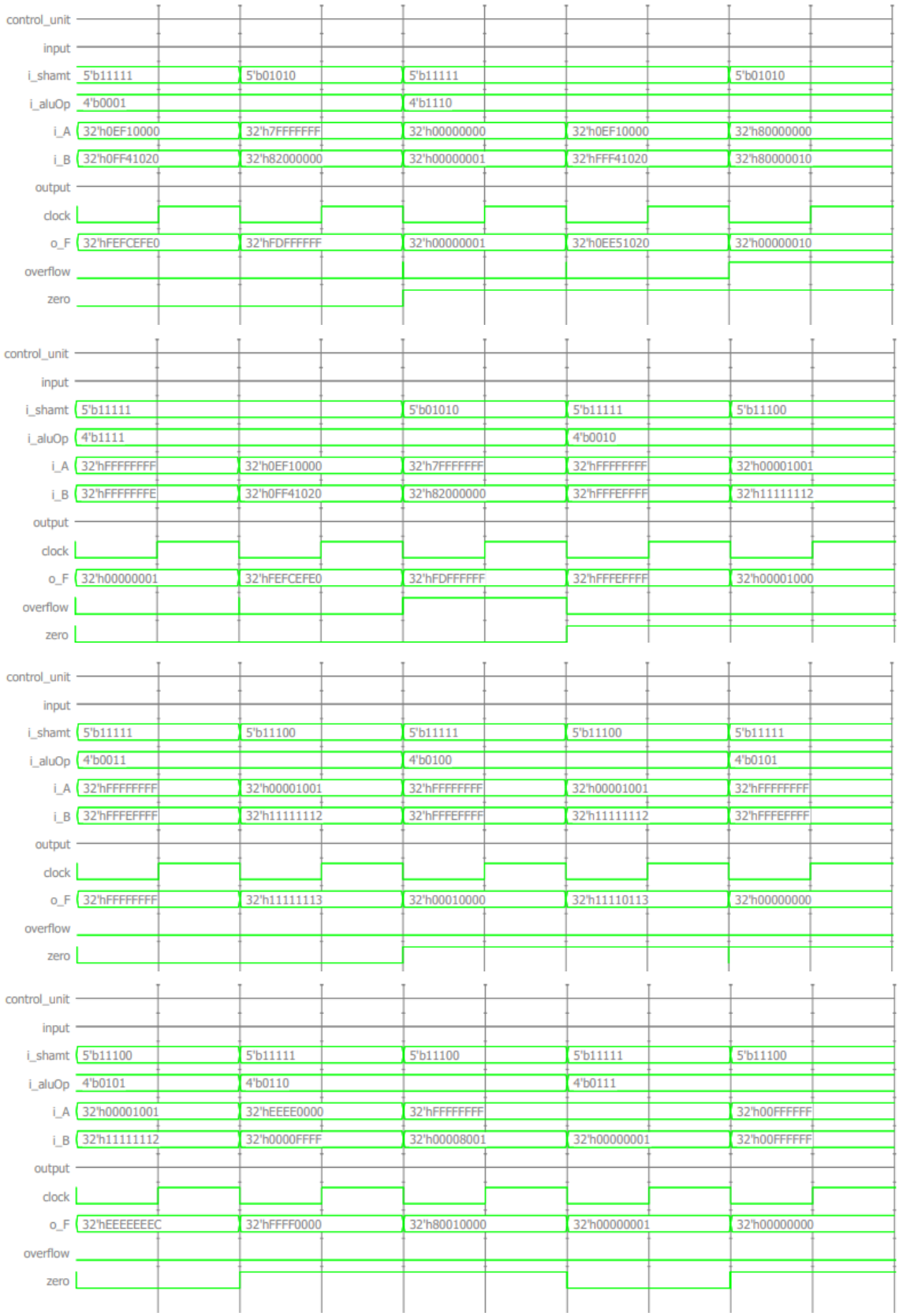
[Part 2 (c.v)] *Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.*

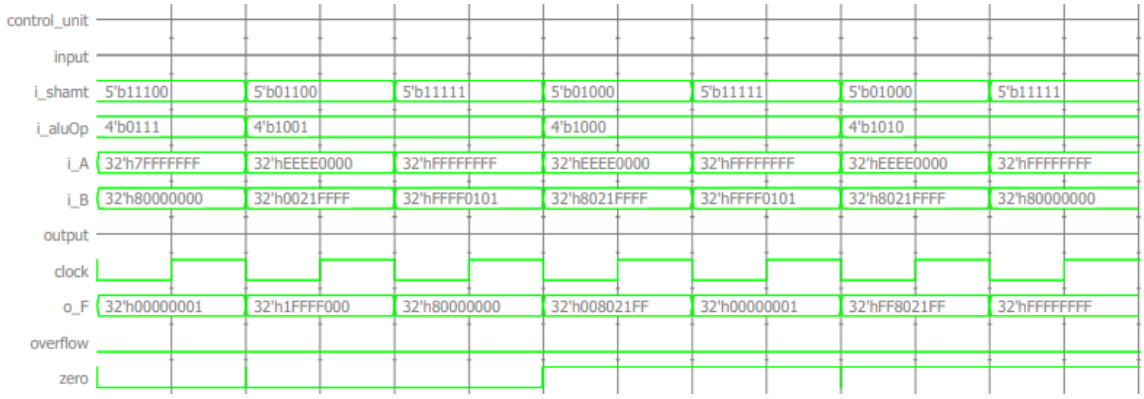
Each opcode was tested with a variety of cases including common and edge. Below shows what each i_aluOp corresponds to in terms of alu function and from there the expected output can be calculated and compared to the designated output.

addu	0000	sll	1001
subu	0001	srl	1000
and	0010	sra	1010
or	0011	beq	1011
xor	0100	bne	1100
nor	0101		1101
lui	0110	add	1110
slt	0111	sub	1111






control_unit
input
i_shamt
5'b11111
5'b01010
5'b11111
5'b01010
i_aluOp
4'b0001
4'b1110
i_A
32'h0EF10000
32'h7FFFFFFF
32'h00000000
32'h0EF10000
32'h80000000
i_B
32'h0FF41020
32'h82000000
32'h00000001
32'hFFF41020
32'h80000010
output
clock
o_F
32'hFEFCEFE0
32'hFDFFFFFF
32'h00000001
32'h0EE51020
32'h00000010
overflow
zero
control_unit
input
i_shamt
5'b11111
5'b01010
5'b11111
5'b11100
i_aluOp
4'b1111
4'b0010
i_A
32'hFFFFFFFF
32'h0EF10000
32'h7FFFFFFF
32'hFFFFFFFF
32'h00001001
i_B
32'hFFFFFFFE
32'h0FF41020
32'h82000000
32'hFFFEFFFF
32'h11111112
output
clock
o_F
32'h00000001
32'hFEFCEFE0
32'hFDFFFFFF
32'hFFFEFFFF
32'h00001000
overflow
zero
control_unit
input
i_shamt
5'b11111
5'b11100
5'b11111
5'b11100
5'b11111
i_aluOp
4'b0011
4'b0100
4'b0101
i_A
32'hFFFFFFFF
32'h00001001
32'hFFFFFFFF
32'h00001001
32'hFFFFFFFF
i_B
32'hFFFEFFFF
32'h11111112
32'hFFFEFFFF
32'h11111112
32'hFFFEFFFF
output
clock
o_F
32'hFFFFFFFF
32'h11111113
32'h00010000
32'h11110113
32'h00000000
overflow
zero
control_unit
input
i_shamt
5'b11100
5'b11111
5'b11100
5'b11111
5'b11100
i_aluOp
4'b0101
4'b0110
4'b0111
i_A
32'h00001001
32'hEEEE0000
32'hFFFFFFFF
32'h00FFFFFF
i_B
32'h11111112
32'h0000FFFF
32'h00008001
32'h00000001
32'h00FFFFFF
output
clock
o_F
32'hEEEEEEEC
32'hFFFF0000
32'h80010000
32'h00000001
32'h00000000
overflow
zero



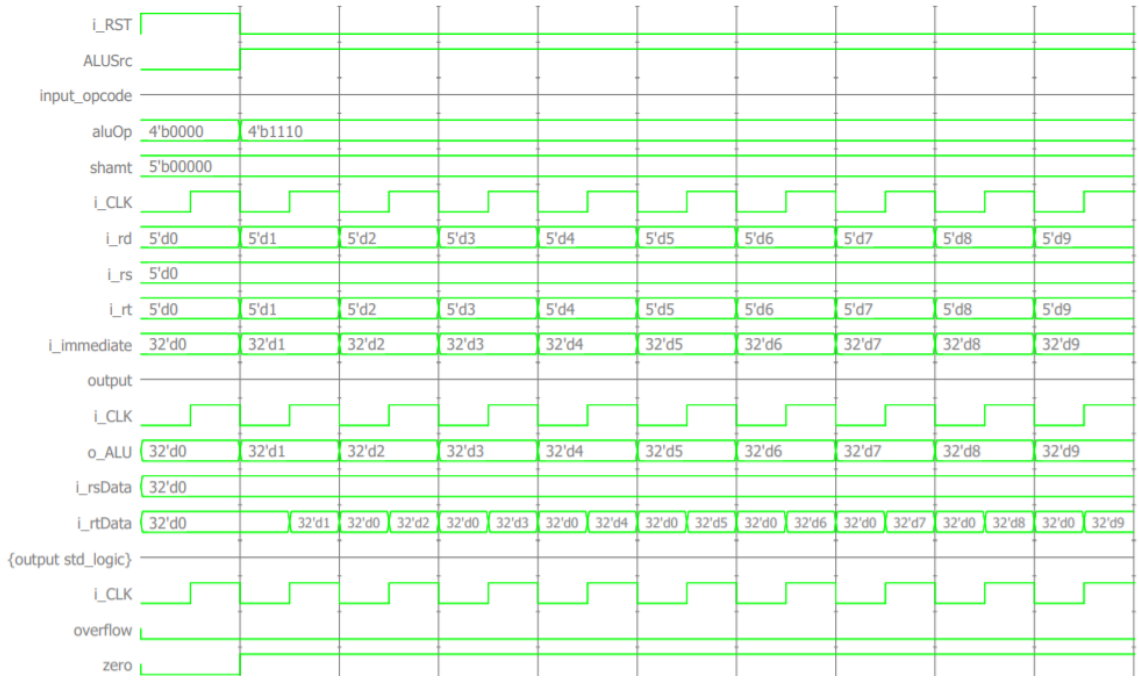
[Part 2 (c.viii)] *justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.*

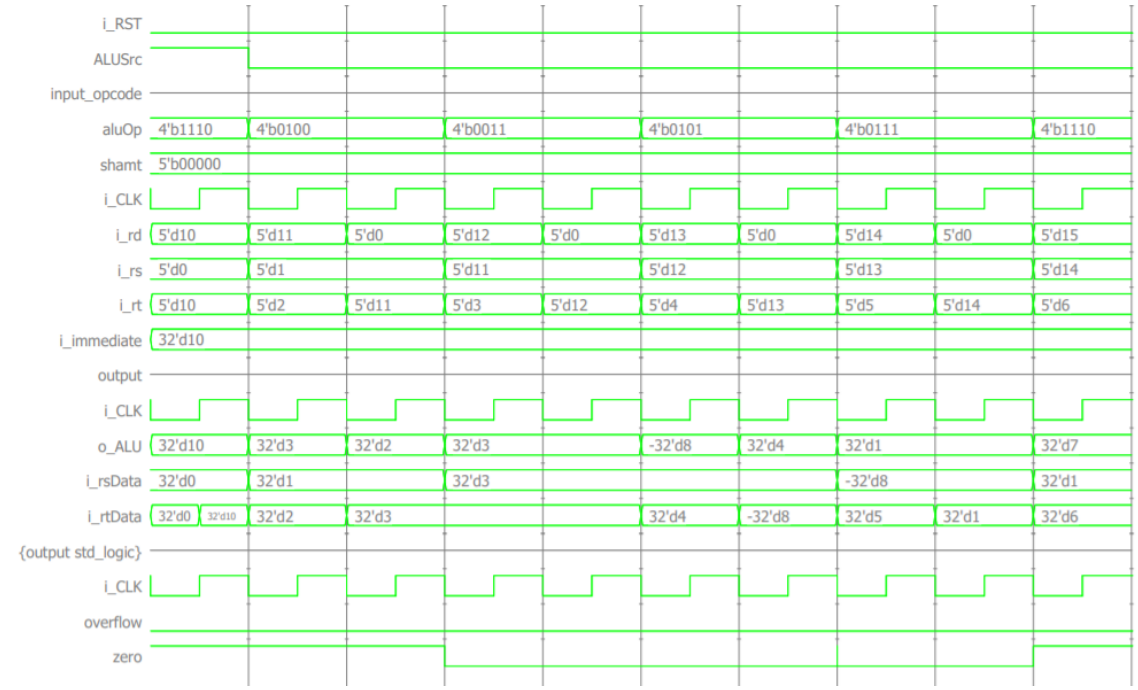
The test bench run below using the datapath from lab2 is comprehensive and tests all aspects of the alu. The expected output is 16383 which matches up with the value that irtData holds on the last clock cycle. A buffer step that lasted a full clock cycle in between each MIPS instruction was added in which i_rtData was set to the output of the register that was just updated for easy error checking. The test plan is comprehensive because it tests a variety of ordinary cases for all functions as well as possible edge cases, including overflow, shifting right logical when the 31 bit is a 1, and shifting right arithmetic when the 31 bit is a 1.

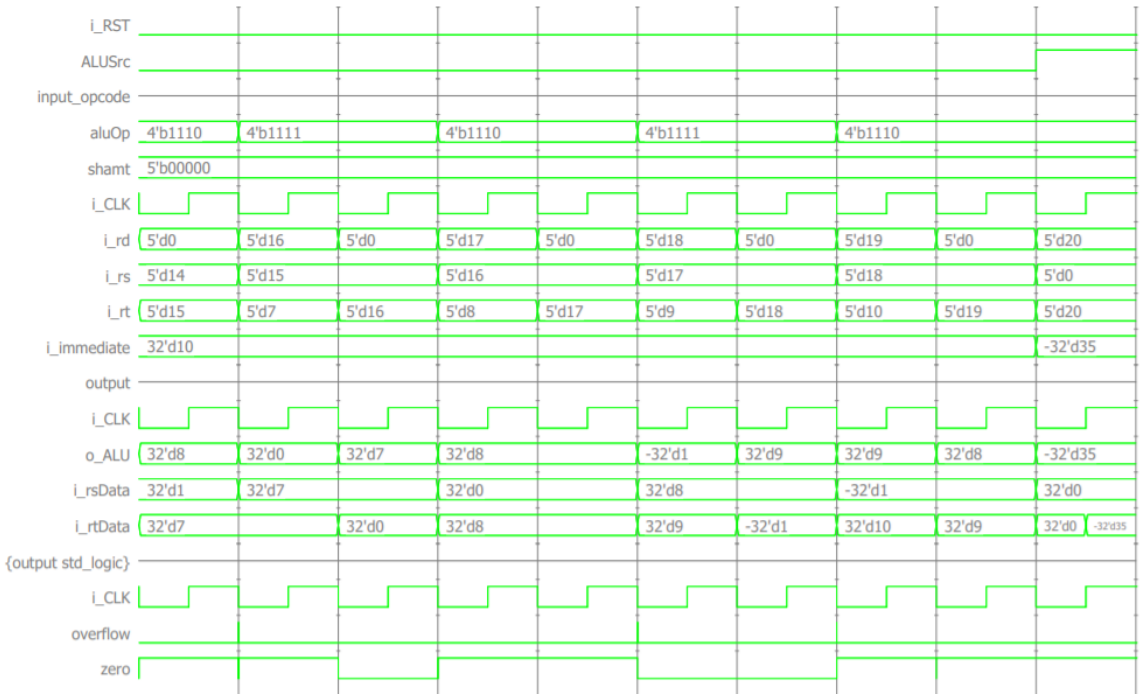
The following MIPS code represents all the functions run through the tb.

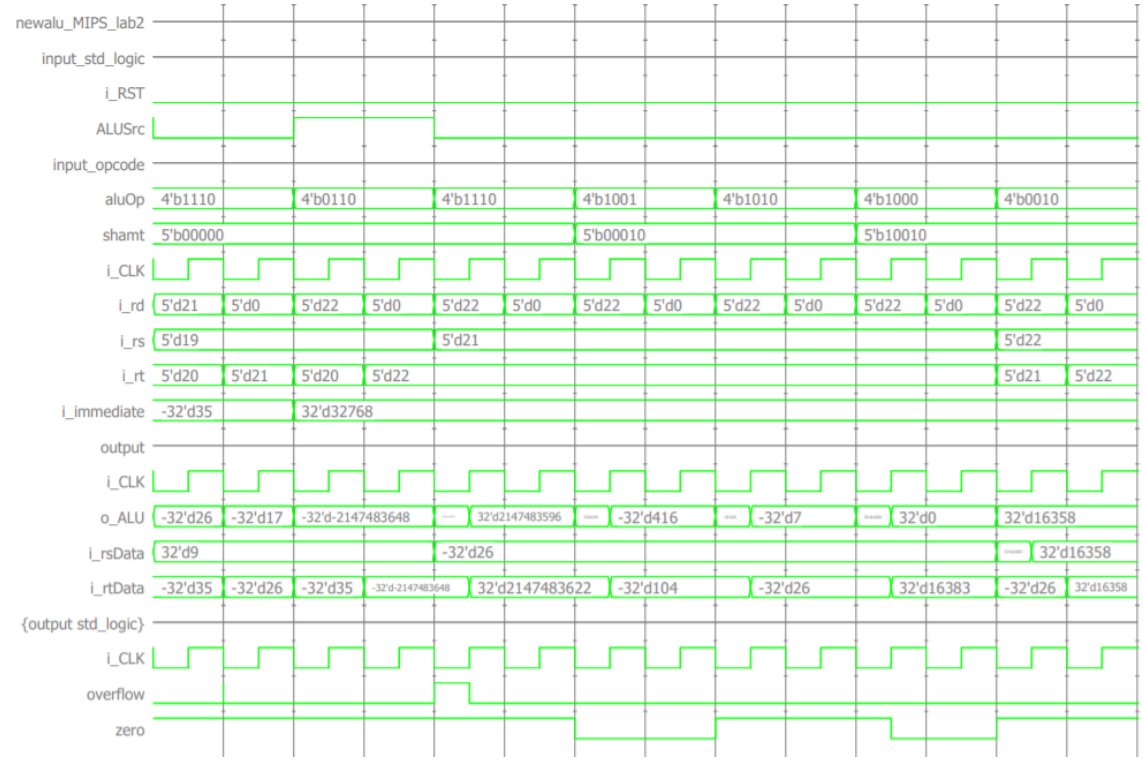
addi $1, $0, 1
addi $2, $0, 2
addi $3, $0, 3
addi $4, $0, 4
addi $6, $0, 6
addi $7, $0, 7
addi $8, $0, 8
addi $9, $0, 9
addi $10, $0, 10
xor $11, $1, $2
or $12, $11, $3
nor $13, $12, $4
slt $14, $13, $5
add $15, $14, $6
sub $16, $15, $7
add $17, $16, $8
sub $18, $17, $9
add $19, $18, $10
addi $20, $0, -35

add $21, $19, $20
lui $22, 0x8000
add $22, $21, $22
sll $22, 2
sra $22, 2
srl $22, 18
and $22, $22, $21

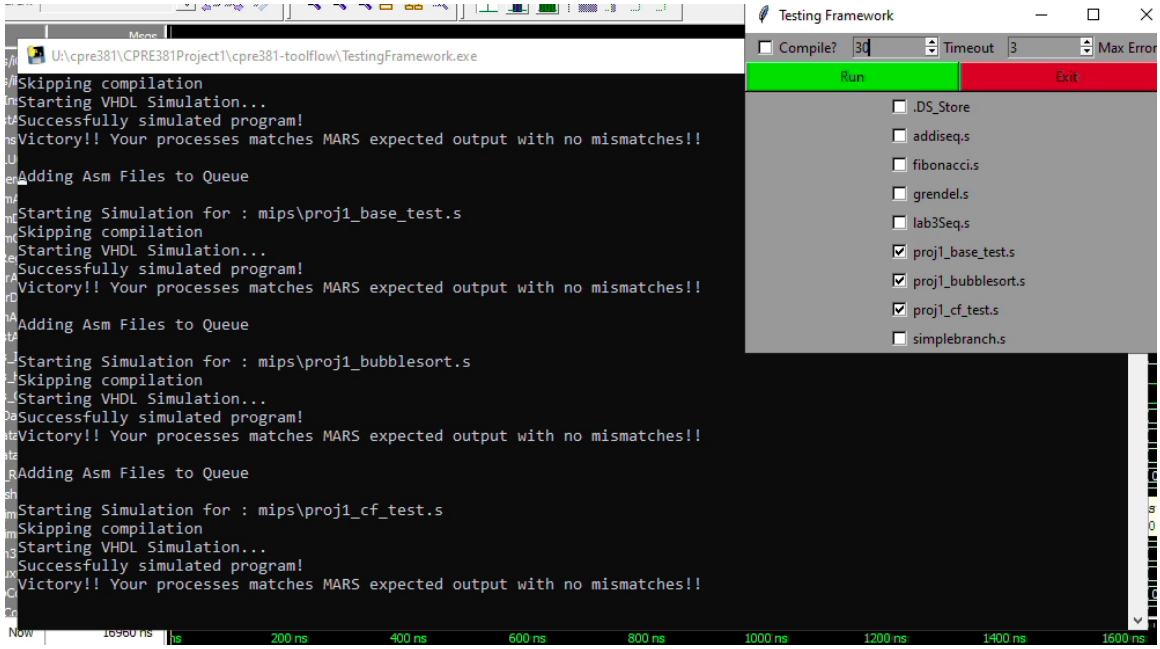





i_RST
ALUSrc
input_opcode
aluOp 4'b1110 4'b1111 4'b1110 4'b1111 4'b1110
shamt 5'b00000
i_CLK
i_rd 5'd0 5'd16 5'd0 5'd17 5'd0 5'd18 5'd0 5'd19 5'd0 5'd20
i_rs 5'd14 5'd15 5'd16 5'd17 5'd18 5'd0
i_rt 5'd15 5'd7 5'd16 5'd8 5'd17 5'd9 5'd18 5'd10 5'd19 5'd20
i_immediate 32'd10 -32'd35
output
i_CLK
o_ALU 32'd8 32'd0 32'd7 32'd8 -32'd1 32'd9 32'd9 32'd8 -32'd35
i_rsData 32'd1 32'd7 32'd0 32'd8 -32'd1 32'd0
i_rtData 32'd7 32'd0 32'd8 32'd9 -32'd1 32'd10 32'd9 32'd0
{output std_logic}
i_CLK
overflow
zero

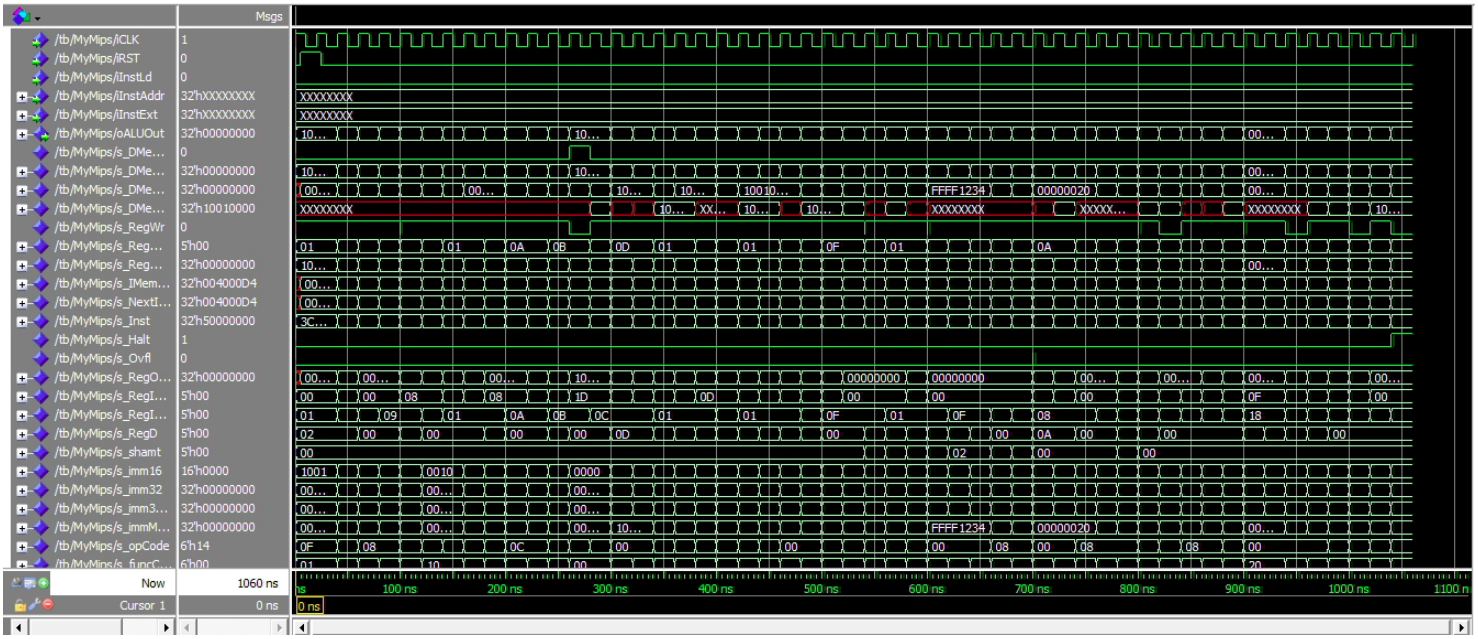

newalu_MIPS_lab2
input_std_logic
i_RST
ALUSrc
input_opcode
aluOp 4'b1110 4'b0110 4'b1110 4'b1001 4'b1010 4'b1000 4'b0010
shamt 5'b00000 5'b00010 5'b10010
i_CLK
i_rd 5'd21 5'd0 5'd22 5'd0 5'd22 5'd0 5'd22 5'd0 5'd22 5'd0 5'd22 5'd0 5'd22 5'd0
i_rs 5'd19 5'd21 5'd22
i_rt 5'd20 5'd21 5'd20 5'd22 5'd21 5'd22
i_immediate -32'd35 32'd32768
output
i_CLK
o_ALU -32'd26 -32'd17 -32'd-2147483648 32'd2147483596 -32'd416 -32'd7 32'd0 32'd16358
i_rsData 32'd9 -32'd26 32'd16358
i_rtData -32'd35 -32'd26 -32'd35 32'd2147483622 -32'd104 -32'd26 32'd16383 -32'd26
{output std_logic}
i_CLK
overflow
zero

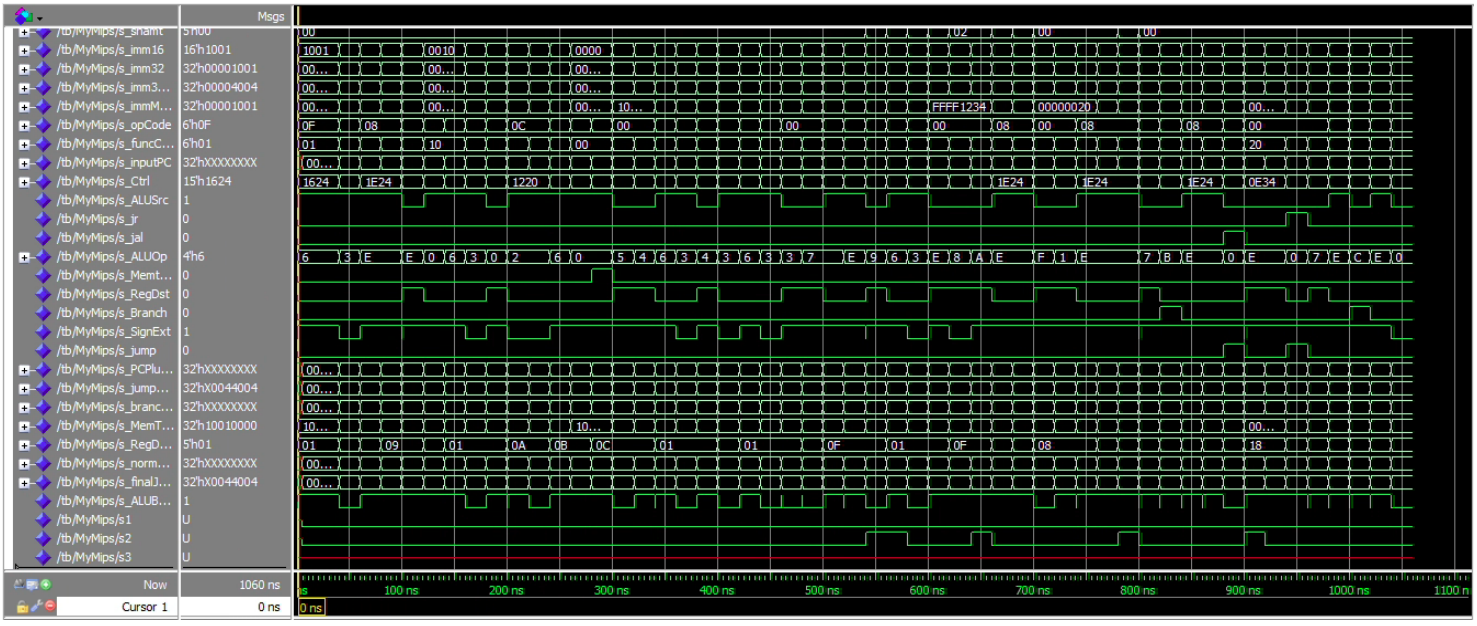
[Part 3] *In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.*



[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.*

This test is a basic tour in showing off MIPS assembly instructions, it tests all of the required logic and arithmetic instructions implemented in our processor.



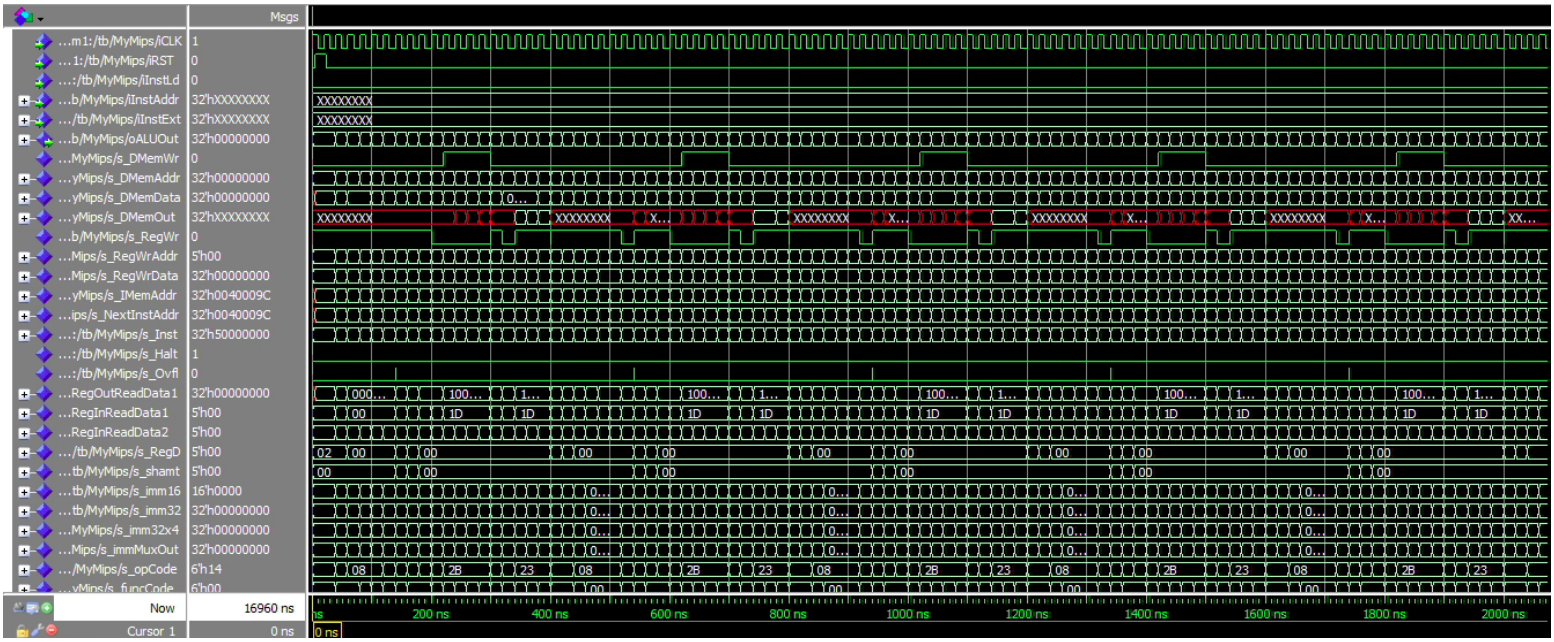


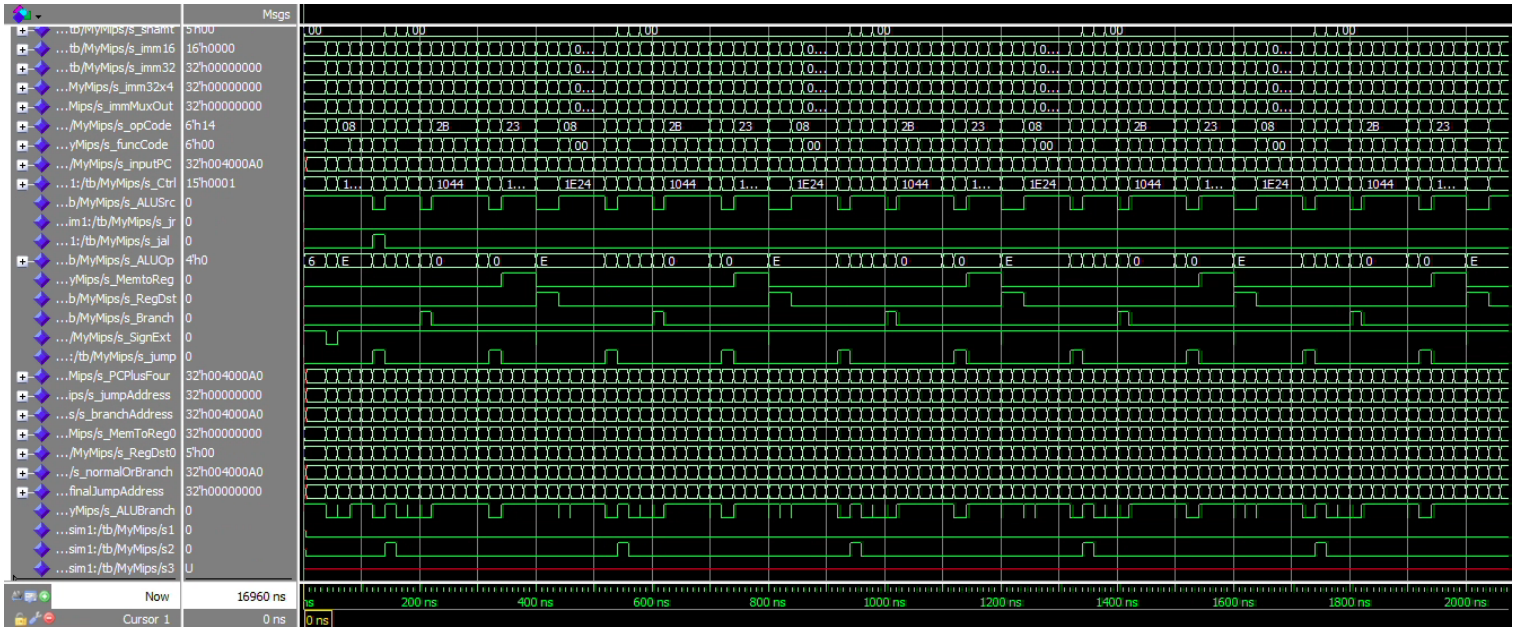
The base tests ran without issue. The testing framework threw no errors and examination of the waveform logs shows everything where it needs to be. The correct instructions run, it interprets pseudoinstructions correctly, it reads the correct registers and the ALU comes up with the correct results, which are stored in the correct registers.

Every relevant register and signal were as expected.

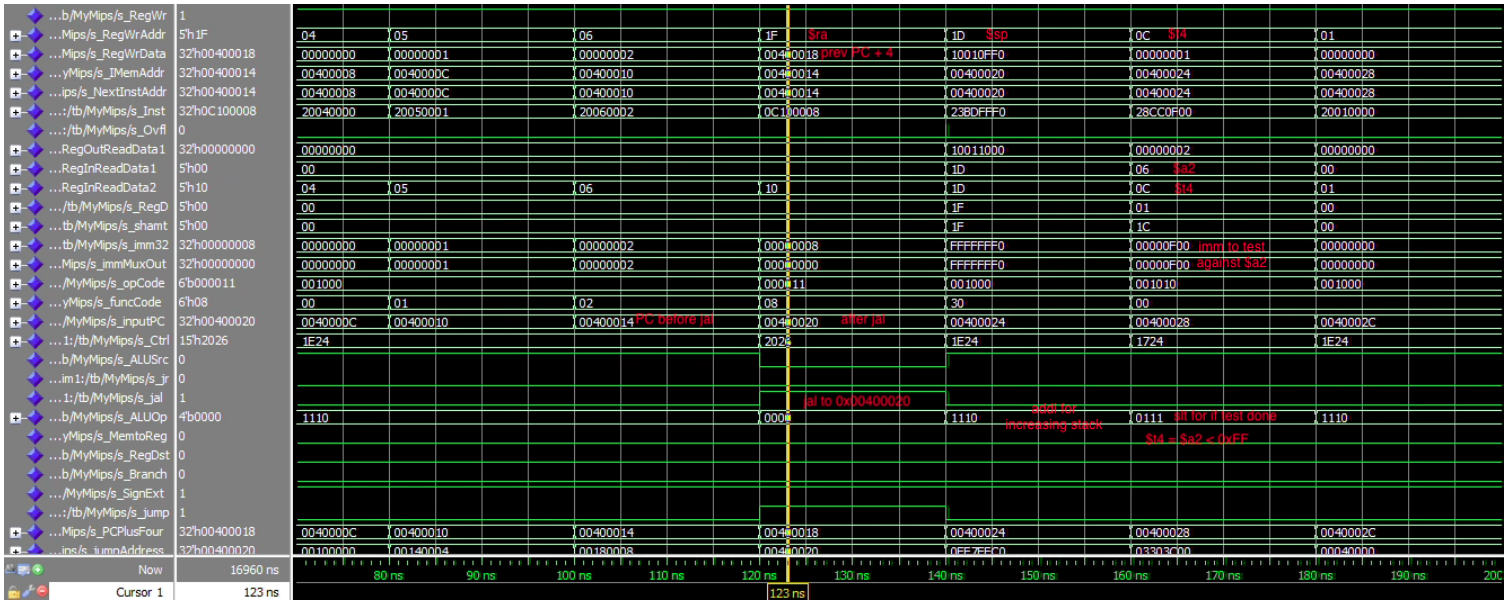
[Part 3 (b)] *Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.*

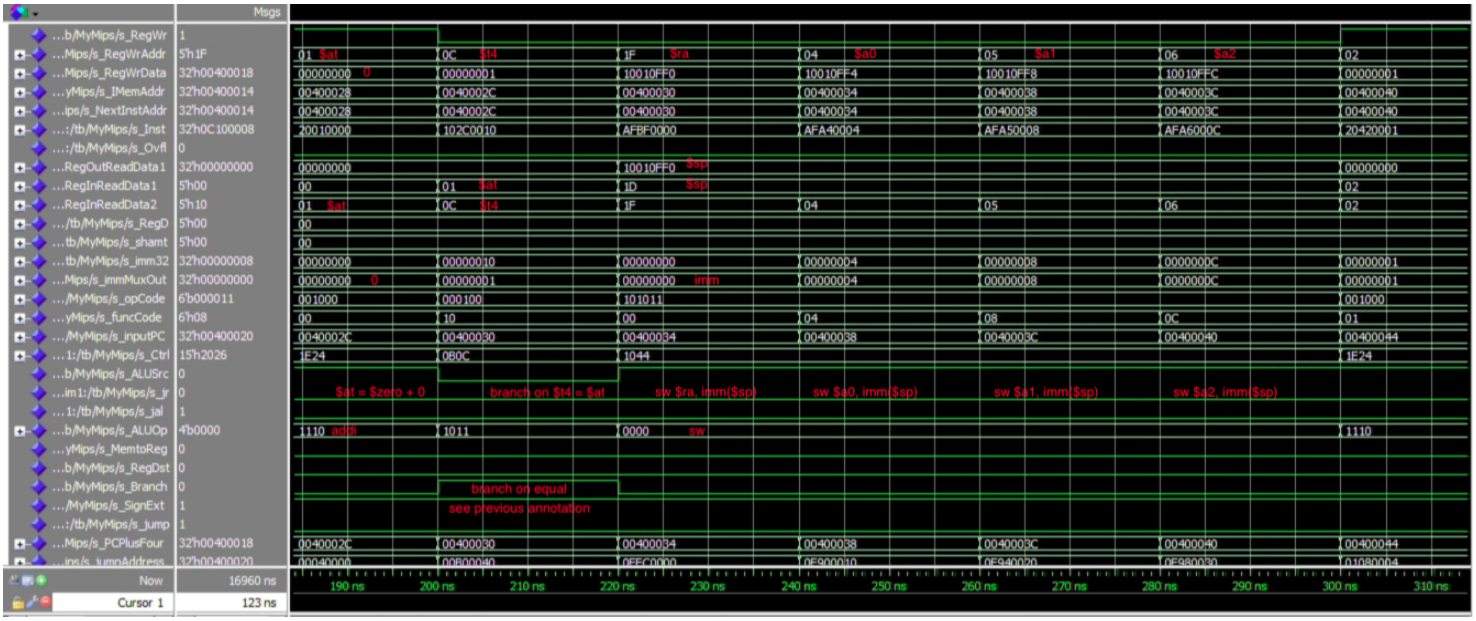
This test uses nested functions with individual activation records. The first activation record is inputted with $a0-2. The stack is pushed back 16 bits, the return address is located at 0($sp), followed by the three values. The function adds up the sum of each value and its subordinates (that is, $a2 = $a2 + $a1 + $a0; $a1 = $a1 + $a0; $a0 = $a0 + 1), which is then stored in memory at their respective original addresses, the $a register values are updated to their new sums and the function calls itself again, pushing the stack forward and repeating the process until $a2 is greater than 0xf00. At this point the function begins to shrink the stack and add each $a0-2 into $v1, when $a0 is 1 the function returns to the original $ra from the first jal and finishes the test.





Once again the test ran without issue.





Every relevant register and signal were as expected.

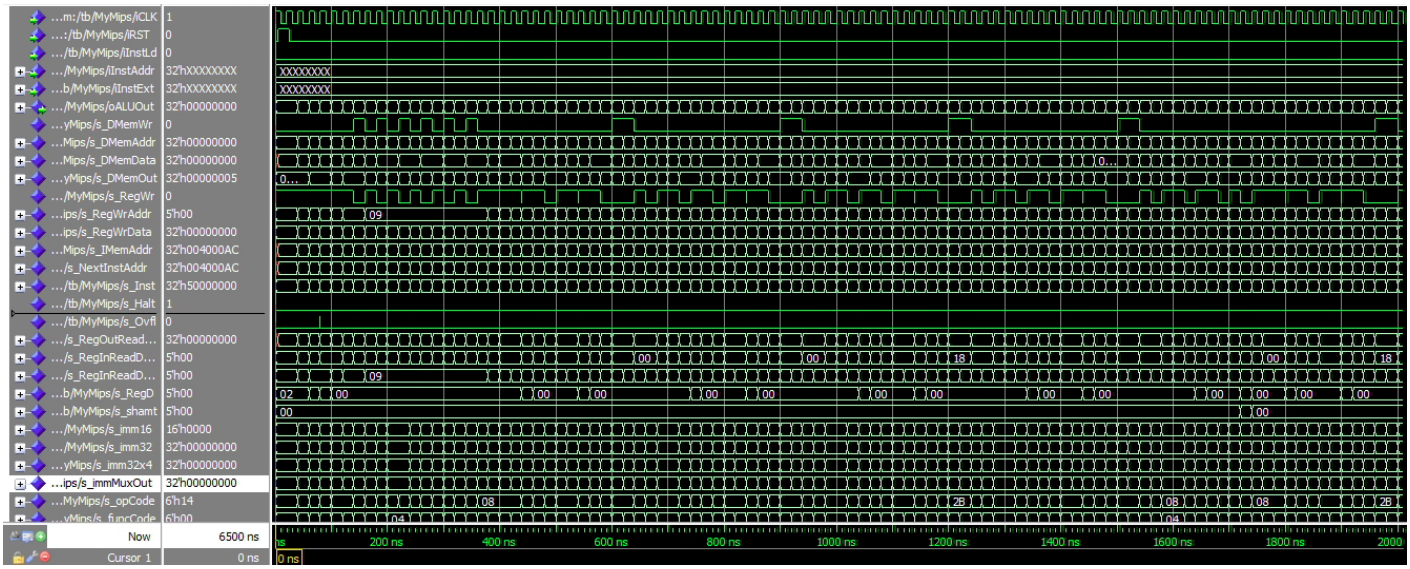
When jal is run, the jal signal goes high, ALUSrc goes low, the PC is updated to the specified address loaded form the immediate mux, and the value of PC before the update is written to $ra + 4. No issues here.

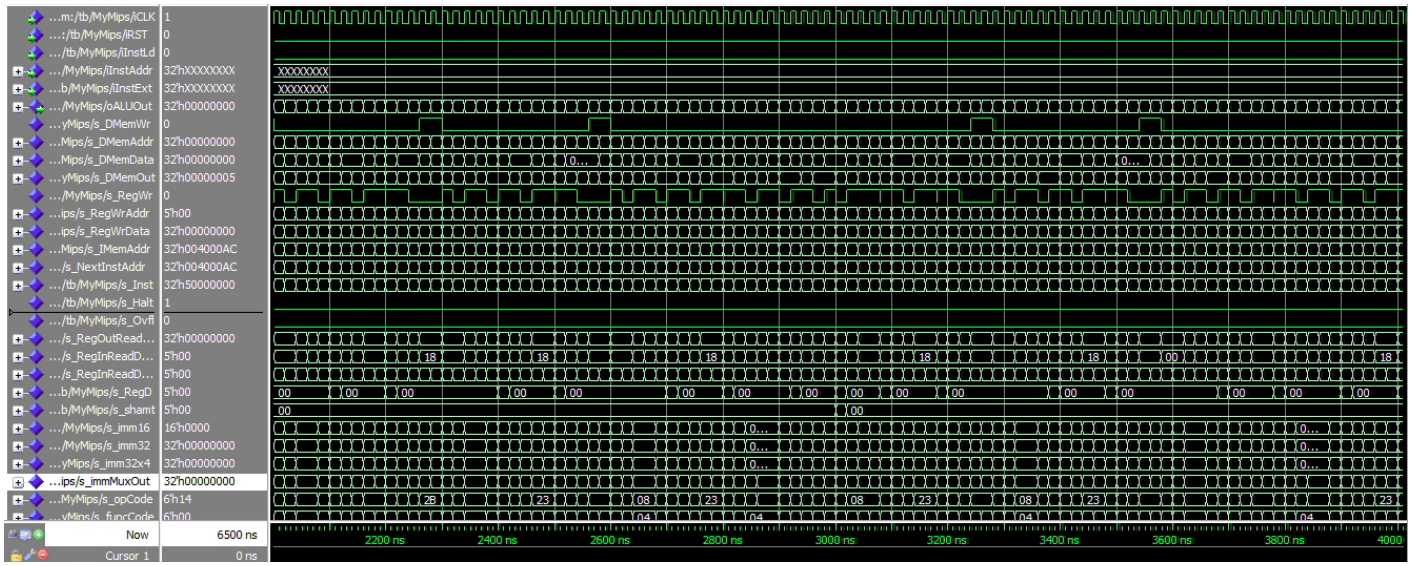
When beq is run, the processor puts the desired imm value into the $at register before running beq and reading the values of rd and $at to determine equality. The branch signal goes high.

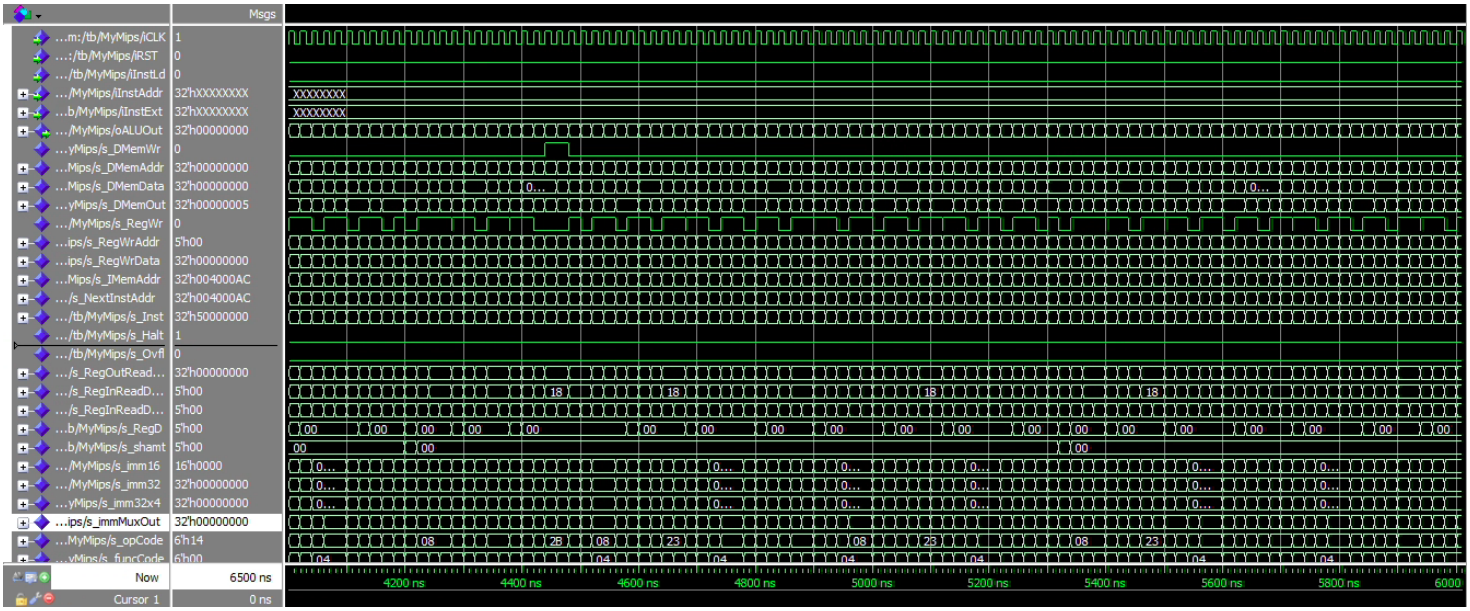
Following that are 4 instances of the sw instruction being run, once again no issues. These tests show that our processor follows its requirements.

[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.*

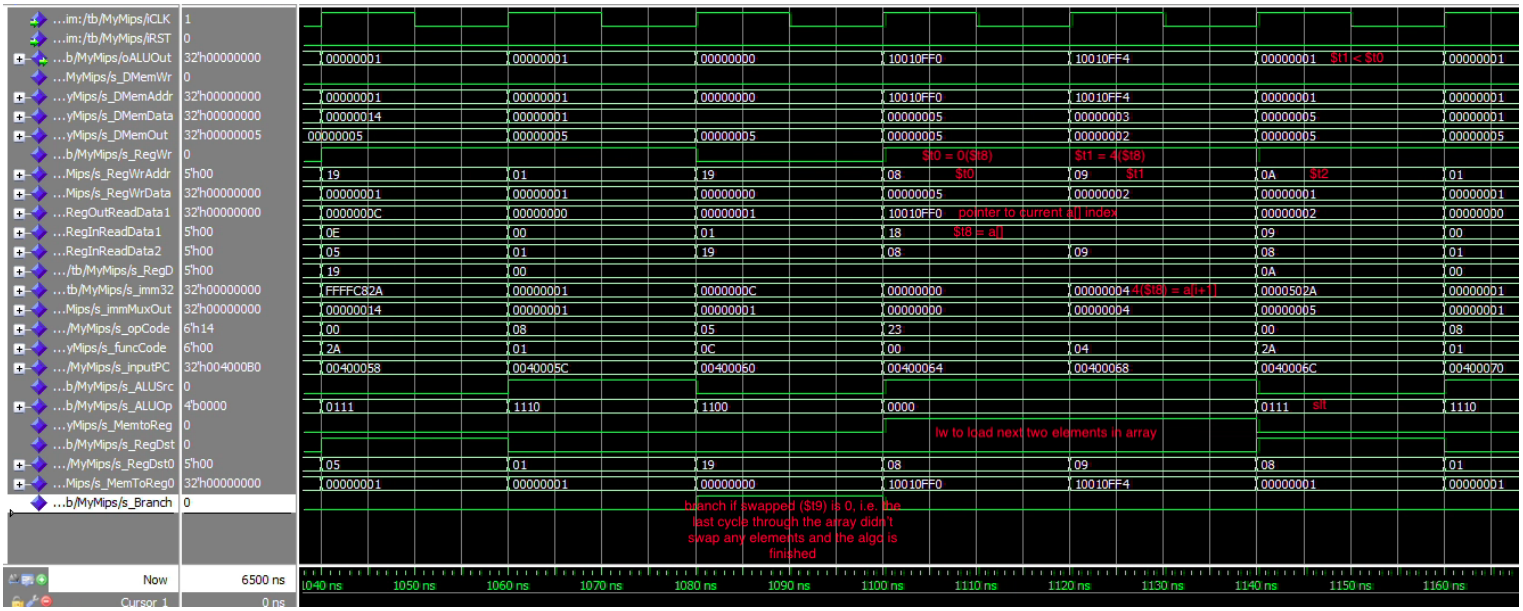
This assembly function uses the simple and inefficient bubblesort sorting algorithm. It makes room on the stack to store the array, then iterates over 0 to N-1 elements. At each element, it checks to see if N + 1 is less than N, if so it switches the values, otherwise it skips to the beginning of the loop again. Since elements may be more than 1 position away from their ideal position, the bubblesort function must iterate over the array until it goes through the entire array without switching any elements.

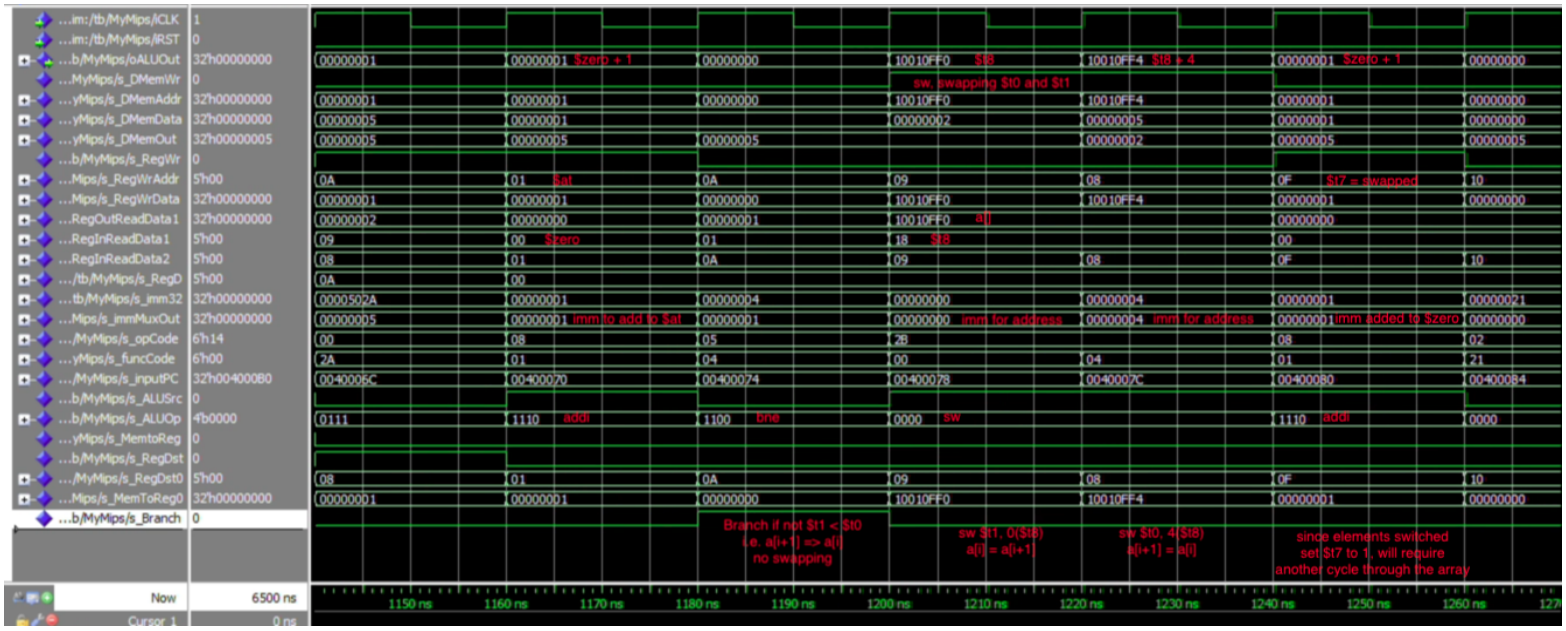



...m:/tb/MyMips/iCLK
...:/tb/MyMips/iRST
.../tb/MyMips/iInstLd
.../MyMips/iInstAddr
...b/MyMips/iInstExt
.../MyMips/oALUOut
...yMips/s_DMemWr
...Mips/s_DMemAddr
...Mips/s_DMemData
...yMips/s_DMemOut
.../MyMips/s_RegWr
...ips/s_RegWrAddr
...ips/s_RegWrData
...Mips/s_IMemAddr
.../s_NextInstAddr
.../tb/MyMips/s_Inst
.../tb/MyMips/s_Halt
.../tb/MyMips/s_Ovfl
.../s_RegOutRead...
.../s_RegInReadD...
.../s_RegInReadD...
...b/MyMips/s_RegD
...b/MyMips/s_shamt
.../MyMips/s_imm16
.../MyMips/s_imm32
...yMips/s_imm32x4
...ips/s_immMuxOut
...MyMips/s_opCode
...yMips/s_funcCode
1
0
0
32'hXXXXXXXX
32'hXXXXXXXX
32'h00000000
0
32'h00000000
32'h00000000
32'h00000005
0
5'h00
32'h00000000
32'h004000AC
32'h004000AC
32'h50000000
1
0
32'h00000000
5'h00
5'h00
5'h00
5'h00
16'h0000
32'h00000000
32'h00000000
32'h00000000
6'h14
6'h00
Now
6500 ns
Cursor 1
0 ns
2200 ns
2400 ns
2600 ns
2800 ns
3000 ns
3200 ns
3400 ns
3600 ns
3800 ns
4000



No issues running the bubble sort test on our CPU in the testing framework. Examining the waveform logs shows correct behavior.

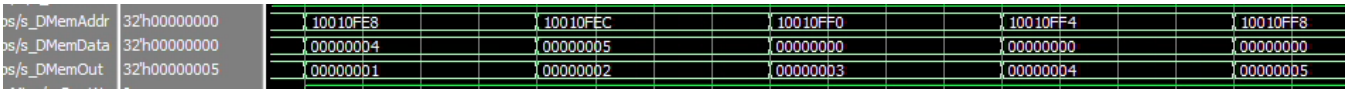




Every relevant register and signal were as expected.

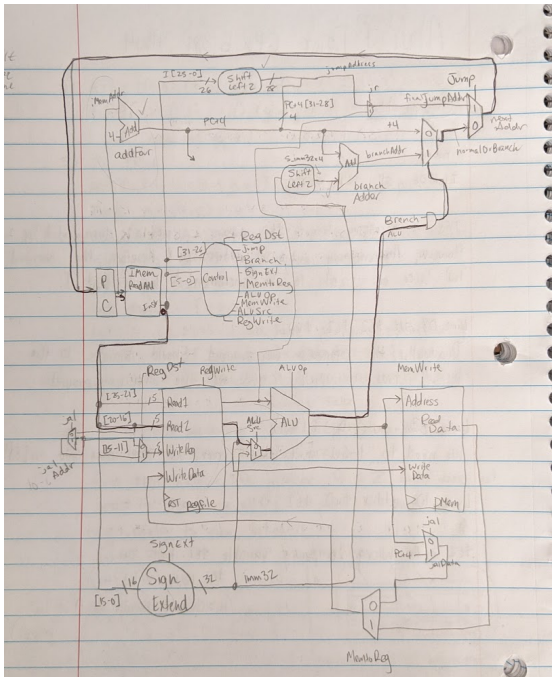
In these specific instructions, the processor begins by checking the value of $t7 which is abstracted as a boolean swapped? value. If it is 0, it means that the processor did not switch any elements during the prior traverse of the array. This means the array as sorted, it would jump to the end of the function if this was the case. In this specific section, swapped is 1 and the process will continue. The processor uses $t8 as the moving pointer to the current element, a[i]. It loads a[i] and a[i+1] into $t0 and $t1 respectively. It uses slt to determine if a[i+1] is less than a[i]. If it is not the case, it jumps past the swapping logic since the elements are sorted already. If it is the case, the processor stores $t1 in 0($t8) and $t0 in 4($t8). It then sets the value of $t7, the swapped value, to 1, as this array will require at least another traverse in order to sort it with this algorithm.

Further analysis shows expected behavior across all of the relevant elements of the processor. At the end, it successfully sorts the array {5, 4, 3, 2, 1} into {1, 2, 3, 4, 5} requiring 5 traverses.



All in all, our processor successfully passed testing.

[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 schematics. What components would you focus on to improve the frequency?*



The critical path of the processor is highlighted in pen, and numerically comes out to 42.834 nanoseconds.

Interestingly enough, the critical path is not reading from the data memory (which is what I had assumed), but actually the branching instructions. After reviewing our code, I found why: when we branch, we are essentially running it through the ALU a second time by comparing the ALUresult to zero.

In order to improve efficiency, we would definitely focus on improving the branch component inside the ALU, as it would be fairly easy to make it run much faster.