CprE 3810: Computer Organization and Assembly-Level Programming

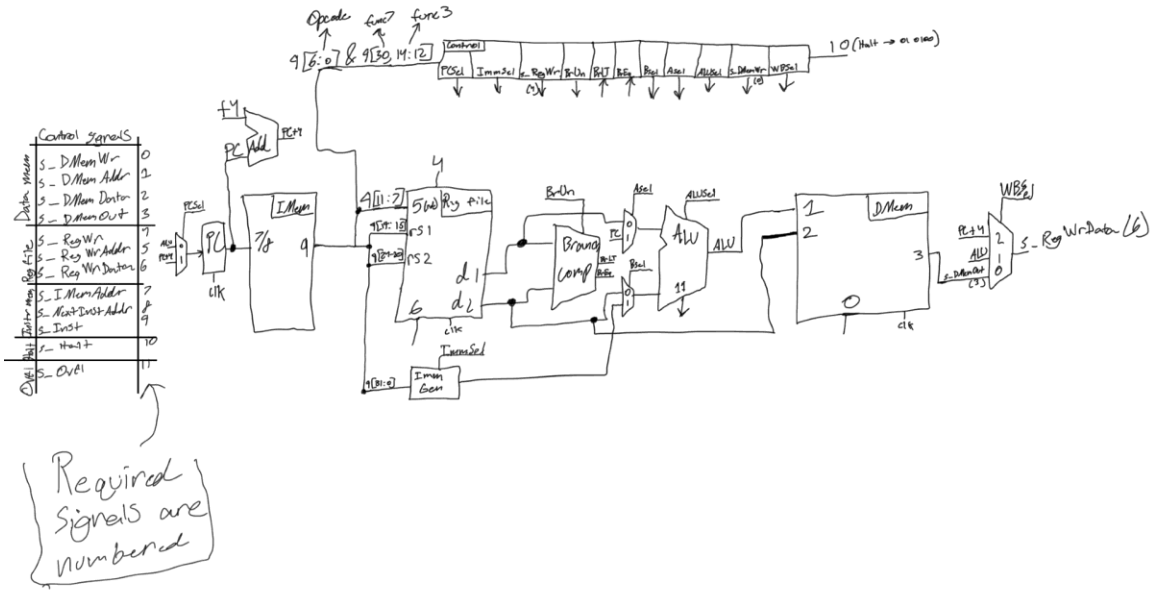
Project Part 1 Report

Team Members: Luke Olsen, Matthew Estes

Project Teams Group #: A_02

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

[Part 2 (d)] Include your final RISC-V processor schematic in your lab report.

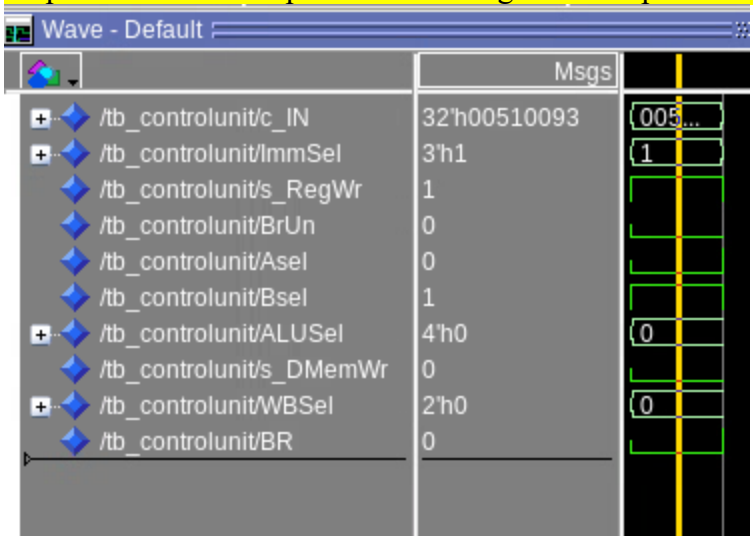


Note: The BrLt and BrEq signals noted on the Branch Comp entity are sub control signals that help dictate branching. We added them into the control block on the top for easier reading and to make it known they are a type of control signal.

[Part 3.1.a.] 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.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Inst	Opcode[6:0]	FUNCT3	FUNCT7/IMM	HEX Value	ImmSel	s_RegWr	BrUn	Bsel	Asel	ALUSel	s_DMemWr	WBSel
2	add	0110011	000	0	0x00000033	000	1	0	0	0	0000	0	00
3	addi	0010011	000	0000000	0x00000013	001	1	0	1	0	0000	0	00
4	and	0110011	111	0000000	0x00007033	000	1	0	0	0	0010	0	00
5	lui	0110111	N/A	0 -> [31:12]	0x00000037	111	1	0	1	1	0000	0	10
6	lw	0000011	010	000000000000	0x00002003	010	1	0	1	0	0000	0	01
7	xor	0110011	100	0	0x00004033	000	1	0	0	0	0100	0	00
8	xori	0010011	100	000000000000	0x00004013	001	1	0	1	0	0100	0	00
9	or	0110011	110	0	0x00006033	000	1	0	0	0	0011	0	00
10	ori	0010011	110	000000000000	0x00006013	001	1	0	1	0	0011	0	00
11	slt	0110011	010	0	0x00002033	000	1	0	0	0	0101	0	00
12	slti	0010011	010	000000000000	0x00002013	001	1	0	1	0	0101	0	00
13	sltiu	0010011	011	000000000000	0x00003013	001	1	0	1	0	0110	0	00
14	sll	0110011	001	0	0x00001033	000	1	0	0	0	0111	0	00
15	srl	0110011	101	0	0x00005033	000	1	0	0	0	1000	0	00
16	sra	0110011	101	1	0x40005033	000	1	0	0	0	1001	0	00
17	sw	0100011	010	000000000000	0x00002023	011	0	0	1	0	0000	1	00
18	sub	0110011	000	1	0x40000033	000	1	0	0	0	0001	0	00
19	beq	1100011	000	000000000000	0x00000063	100	0	0	0	1	0000	0	00
20	bne	1100011	001	000000000000	0x00001063	100	0	0	0	1	0000	0	00
21	blt	1100011	100	000000000000	0x00004063	100	0	0	0	1	0000	0	00
22	bge	1100011	101	000000000000	0x00005063	100	0	0	0	1	0000	0	00
23	bltu	1100011	110	000000000000	0x00006063	100	0	1	0	1	0000	0	00
24	bgeu	1100011	111	000000000000	0x00007063	100	0	1	0	1	0000	0	00
25	jal	1101111	N/A	0 -> [20:1]	0x0000006F	101	1	0	1	1	0000	0	10
26	jalr	1100111	000	000000000000	0x00000067	110	1	0	1	1	0000	0	10
27	lb	0000011	000	000000000000	0x00000003	010	1	0	1	0	0000	0	01
28	lh	0000011	001	000000000000	0x00001003	010	1	0	1	0	0000	0	01
29	lbu	0000011	100	000000000000	0x00004003	010	1	0	1	0	0000	0	01
30	lhu	0000011	101	000000000000	0x00005003	010	1	0	1	0	0000	0	01
31	slli	0010011	001	000000000000	0x00001013	001	1	0	1	0	0111	0	00
32	srli	0010011	101	000000000000	0x00005013	001	1	0	1	0	1000	0	00
33	srai	0010011	101	010000000000	0x40005013	001	1	0	1	0	1001	0	00
34	auipc	0010111	N/A	0 -> [31:12]	0x00000017	111	1	0	1	1	0000	0	10
35	wfi(HALT)	0010100	N/A	0 -> [31:0]	0x0000000B	000	0	0	0	0	0000	0	00
36													

[Part 3.1.(b)] 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).

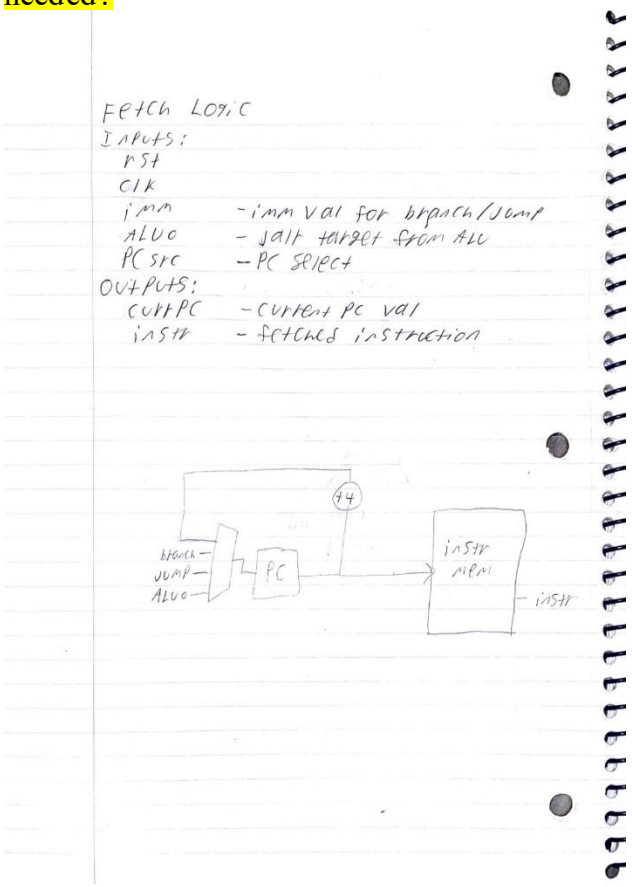


This is for addi. Will add more screenshots once addi works on entire processor.

[Part 3.2. (a)] 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.

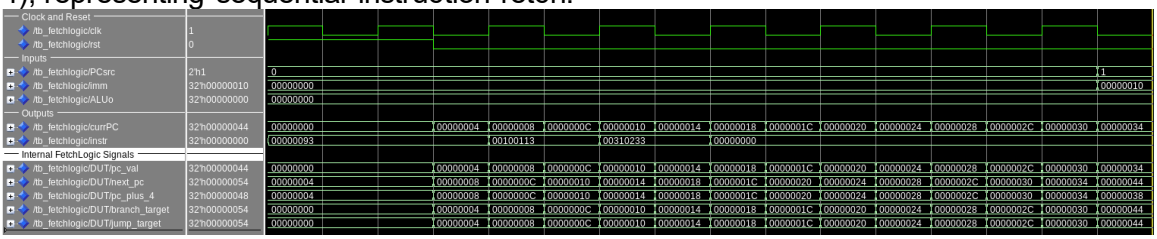
For normal instructions, the fetch logic just increments pc +4 to get the next instruction. For branch instructions, the pc must be added with the immediate value. For jump instructions like jal or jalr the pc must be updated with a target address that comes from either the immediate value or a register. For jump and link the fetch logic also needs to store the return address into a register for safe keeping.

[Part 3.2. (b)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

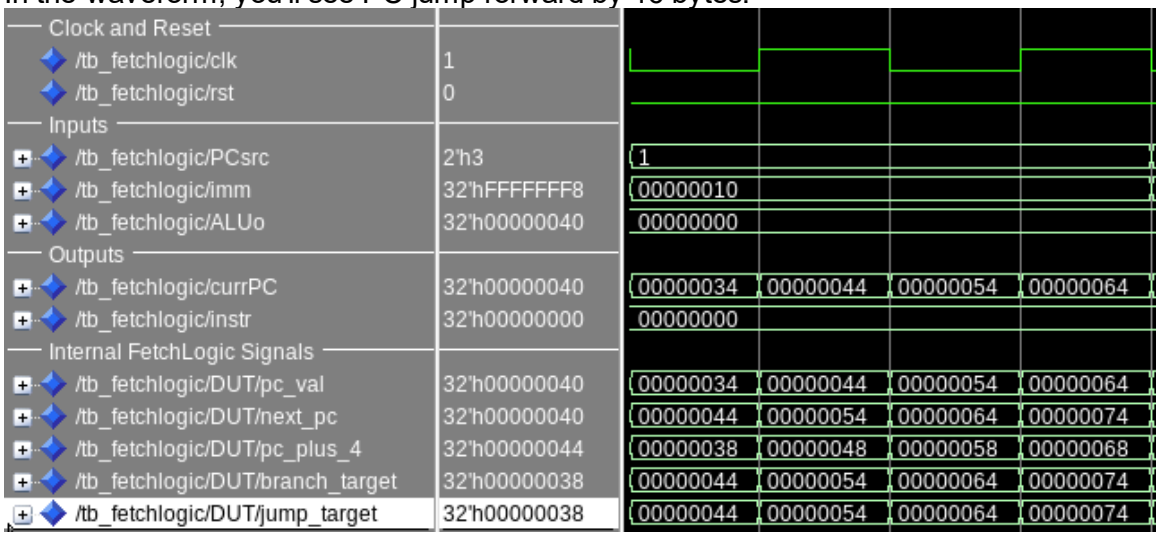


[Part 3.2.(c)] Implement your new instruction fetch logic using VHDL. Use QuestaSim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the QuestaSim waveforms in your writeup.

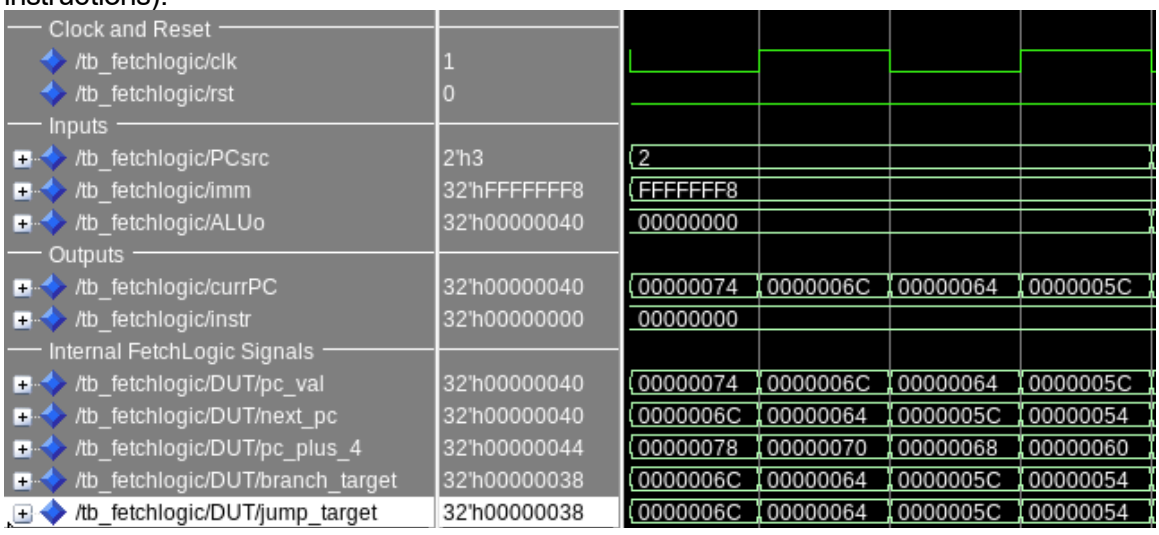
Test 1
When PCsrc = "00", the PC increments normally by 4 each clock cycle (PC_next = PC + 4), representing sequential instruction fetch.



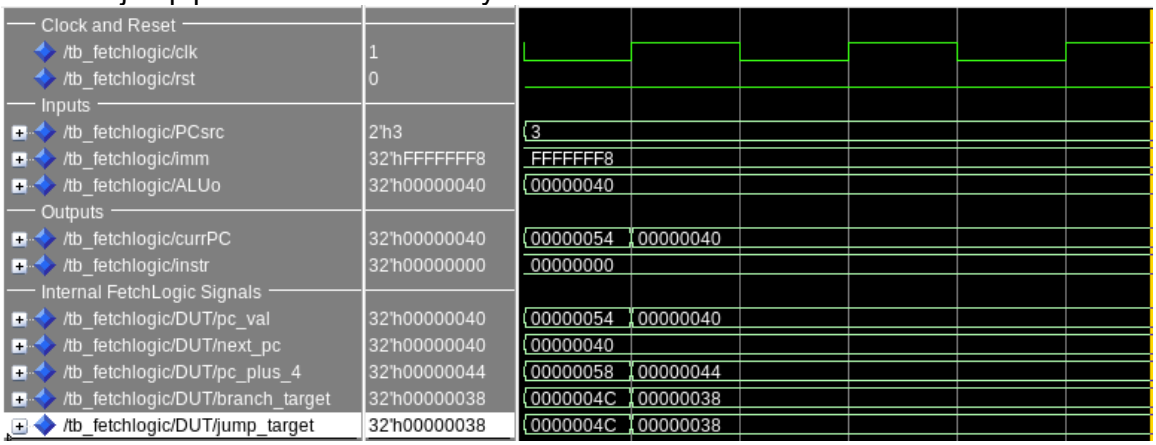
Test 2
When PCsrc = "01", the ALU selects the branch target (PC + imm).
In the waveform, you'll see PC jump forward by 16 bytes.



Test 3
or PCsrc = "10", the PC performs a jump using the immediate offset (PC + imm).
Since imm is negative, the waveform shows the PC jumping backward by 8 bytes (two instructions).



Test 4
When PCsrc = "11", the PC loads directly from the ALU output (PC_next = ALUo), which corresponds to JALR.
In the waveform, the PC jumps immediately to address 0x00000040, confirming that the absolute jump path functions correctly.



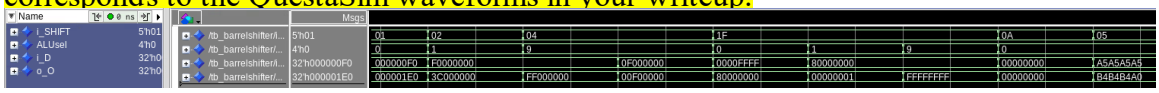
[Part 3.3.1.(a)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does RISC-V not have a `sla` instruction?

- SRL: Logical shifts are better for unsigned values. Logical shifts shift the entire value for any value. This is good for multiplying and dividing values.
- SRA: Better for signed values. Arithmetic shifts keep the signed value of the number, a type of extension.

[Part 3.3.1.(b)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.
The structural 32 bit right shifter is made using a generate statement to make 32 muxes. We then wire them to the corresponding neighbors based on the corresponding shift creating the cascading affect. The arithmetic and logical shifts are controlled by the i_ARI control signal. For logical zeros are shifted into the MSBs. For arithmetic shifts we shift the sign bit as the “fill bit”.

[Part 3.3.1.(c)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.
Adding left shift support we use i_DIR to control the direction. For left shifts the bits are flipped or “reversed”. This way we don’t need to implement an entire second left shifter to perform the same thing.

[Part 3.3.1.(d)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.



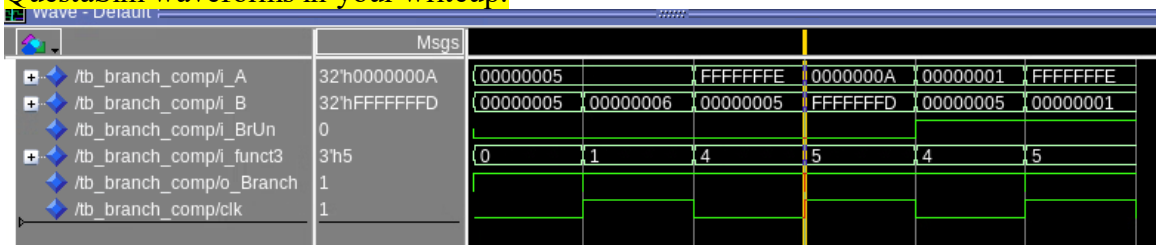
Test 1: Logical Left Shift. Test 2: Logical Right Shift. Test 3: Arithmetic right shift sign=1. Test 4: Arithmetic right shift sign=0. Test 5: Logical Left Shift edge case. Test 6:

Logical right shift edge case. Test 7: Arithmetic right shiftedge case. Test 8: Zero input. Test 9: Pattern Test.

[Part 3.3.2.(a)] 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.

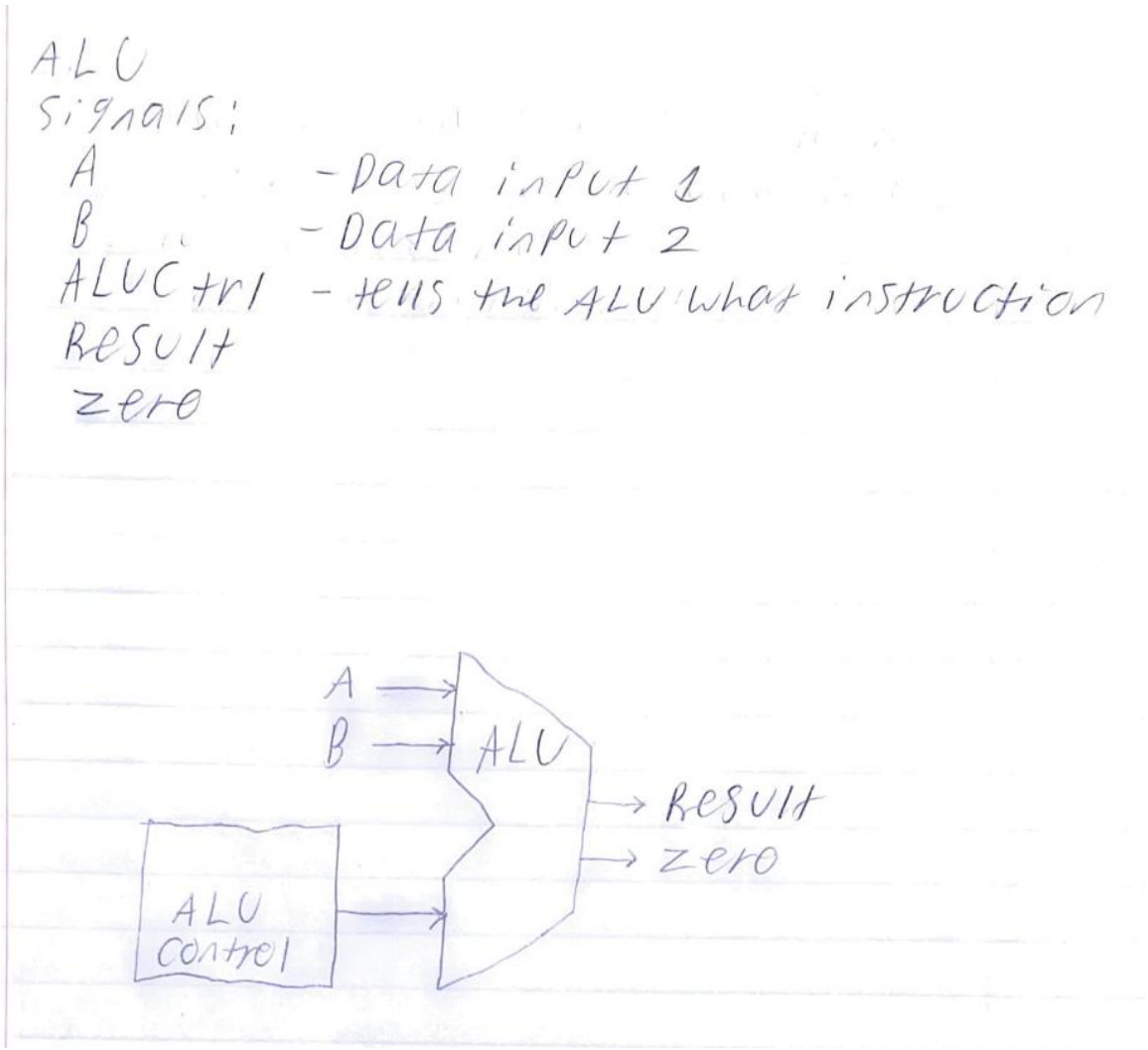
Our design approach was to try to compact everything into blocks to make it easier to read and to make it more organized. One way to help was to separate the branch logic from the ALU into its own block.

[Part 3.3.2.(b)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.



The branch comp will output a signal of 1 signaling it is a branch being executed and the BrUn will indicate if its signed or unsigned.

[Part 3.3.3] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: How is Zero calculated? How is `slt` implemented?



Zero is calculated by looking at the result value and seeing if it is all 0's.
slt compares two signed integers and sets the result to 1 if A < B, otherwise 0.

[Part 3.3.5] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

[Part 3.3.8] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

[Part 4] 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.

[Part 4.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 4.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 4.c] Create and test an application that sorts an array with N elements using the MergeSort algorithm (link). Name this file Proj1_mergesort.s.

[Part 5] 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?