CprE 3810: Computer Organization and Assembly-Level Programming

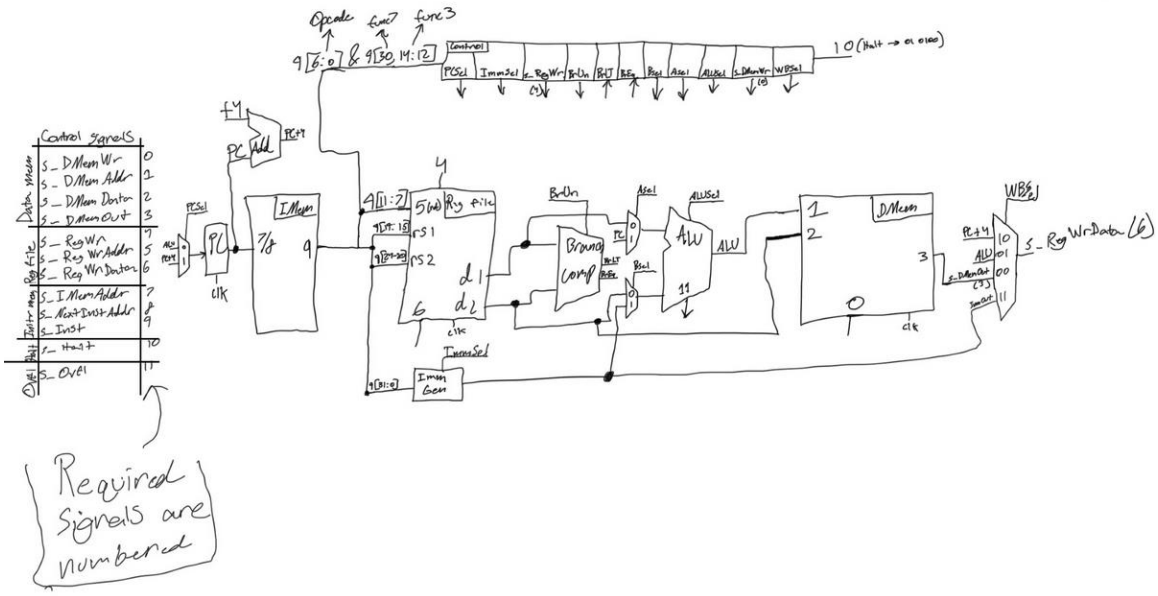
Project Part 1 Report

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.

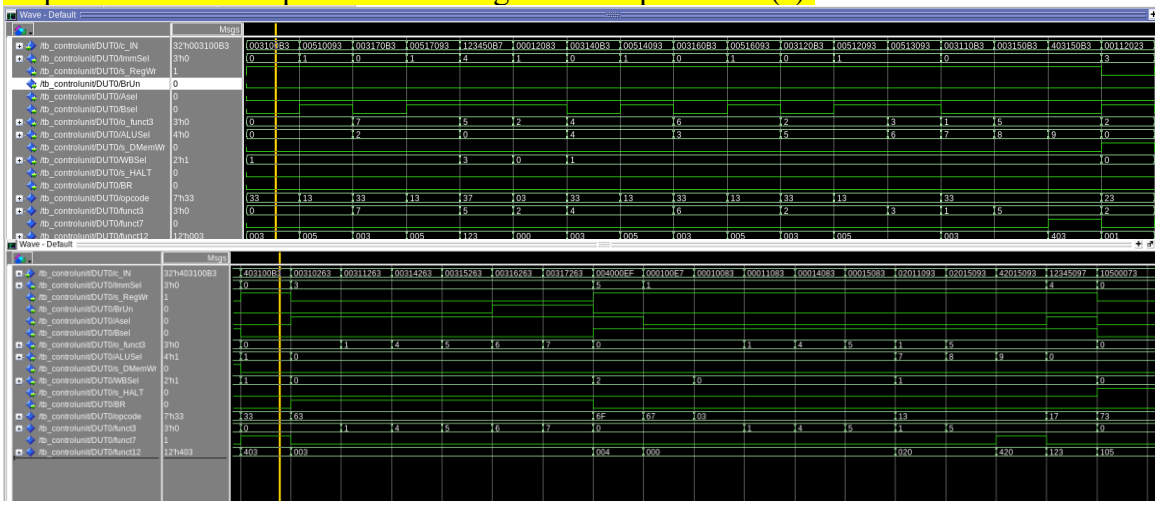


Note: The BrLt and BrEq are internal signals of the branch comp unit. They are not used as inputs or outputs but are there to help know hats going on inside the branch comp unit.

[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.

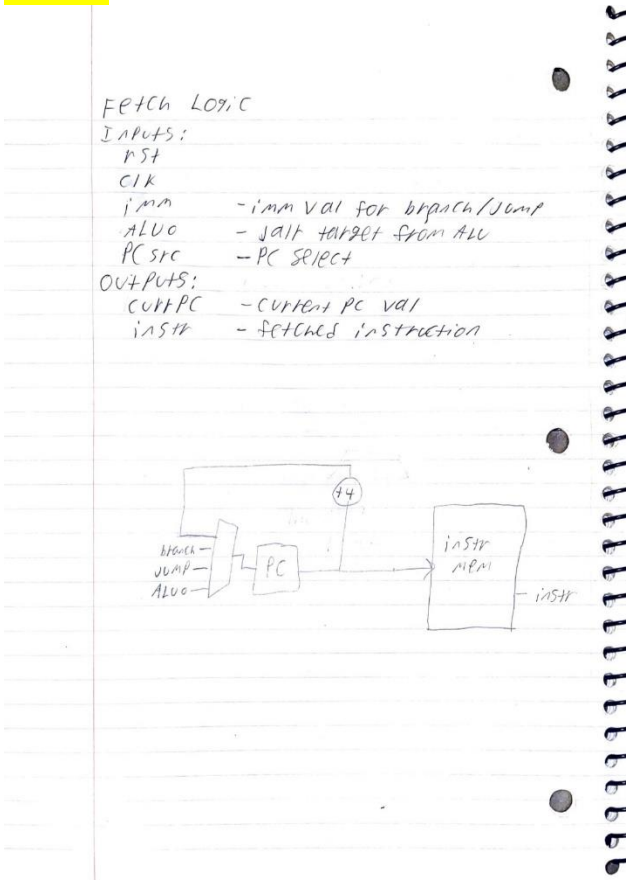
	A	B	C	D	E	F	G	H	I	J	K
1	Instr	Opcode	ImmSel	s_RegWr	BrUn	Asel	Bsel	s_DMemV	WBSel	s_HALT	BR
2	add	110011	0	1	0	0	0	0	1	0	0
3	addi	10011	1	1	0	0	1	0	1	0	0
4	and	110011	0	1	0	0	0	0	1	0	0
5	andi	10011	1	1	0	0	1	0	1	0	0
6	lui	110111	100	1	0	0	1	0	11	0	0
7	lw	11	1	1	0	0	1	0	0	0	0
8	xor	110011	0	1	0	0	0	0	1	0	0
9	xori	10011	1	1	0	0	1	0	1	0	0
10	or	110011	0	1	0	0	0	0	1	0	0
11	ori	10011	1	1	0	0	1	0	1	0	0
12	slt	110011	0	1	0	0	0	0	1	0	0
13	slti	10011	1	1	0	0	1	0	1	0	0
14	sltiu	10011	1	1	0	0	1	0	1	0	0
15	sll	110011	0	1	0	0	0	0	1	0	0
16	srl	110011	0	1	0	0	0	0	1	0	0
17	sra	110011	0	1	0	0	0	0	1	0	0
18	sw	100011	11	0	0	0	1	1	0	0	0
19	sub	110011	0	1	0	0	0	0	1	0	0
20	beq	1100011	11	0	0	1	0	0	0	0	1
21	bne	1100011	11	0	0	1	0	0	0	0	1
22	blt	1100011	11	0	0	1	0	0	0	0	1
23	bge	1100011	11	0	0	1	0	0	0	0	1
24	bltu	1100011	11	0	1	1	0	0	0	0	1
25	bgeu	1100011	11	0	1	1	0	0	0	0	1
26	jal	1101111	101	1	0	1	1	0	10	0	0
27	jalr	1100111	1	1	0	0	1	0	10	0	0
28	lb	11	1	1	0	0	1	0	0	0	0
29	lh	11	1	1	0	0	1	0	0	0	0
30	lbu	11	1	1	0	0	1	0	0	0	0
31	lhu	11	1	1	0	0	1	0	0	0	0
32	slli	10011	1	1	0	0	1	0	1	0	0
33	srli	10011	1	1	0	0	1	0	1	0	0
34	srai	10011	1	1	0	0	1	0	1	0	0
35	auipc	10111	100	1	0	1	1	0	1	0	0
36	wfi	1110011	0	0	0	0	0	0	0	1	0

[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).



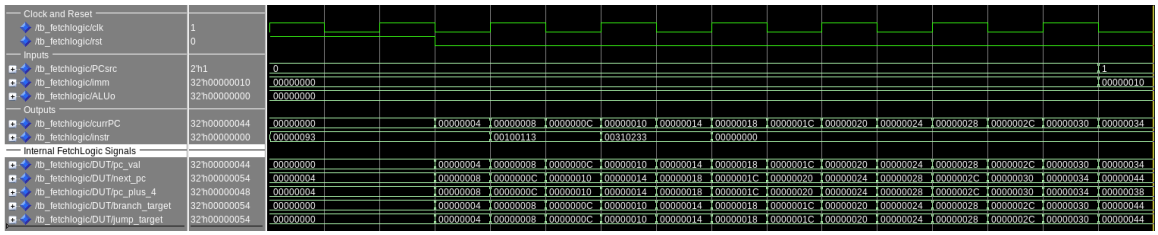
[Part 3.2. (a)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.
For normal instructions, the fetch logic just increments pc +4 to get the 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?

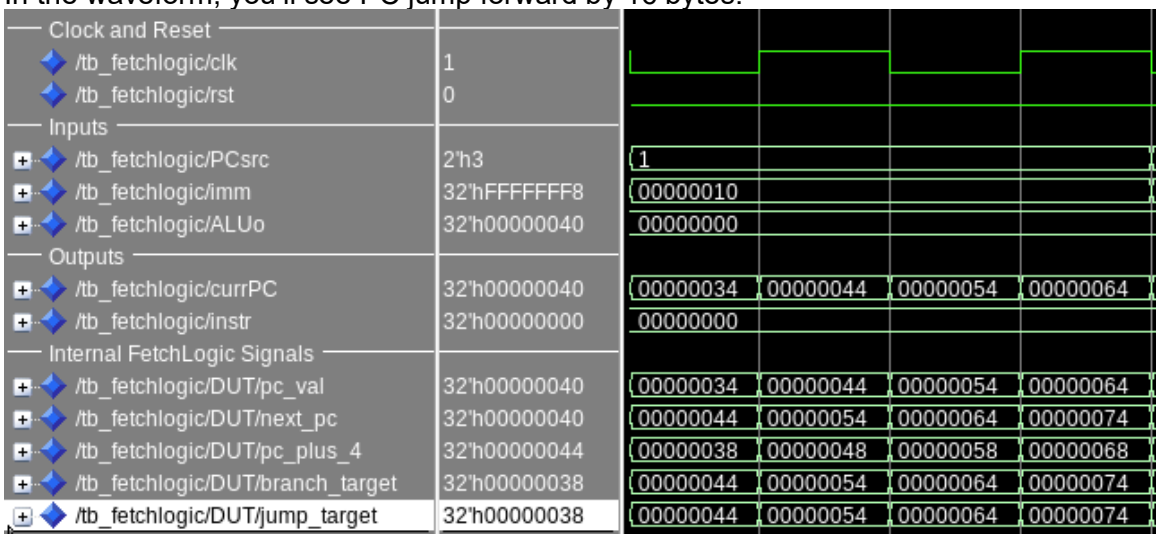


[Part 3.2.(c)] Implement your new instruction fetch logic using VHDL. Use QuestaSim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the QuestaSim waveforms in your writeup.

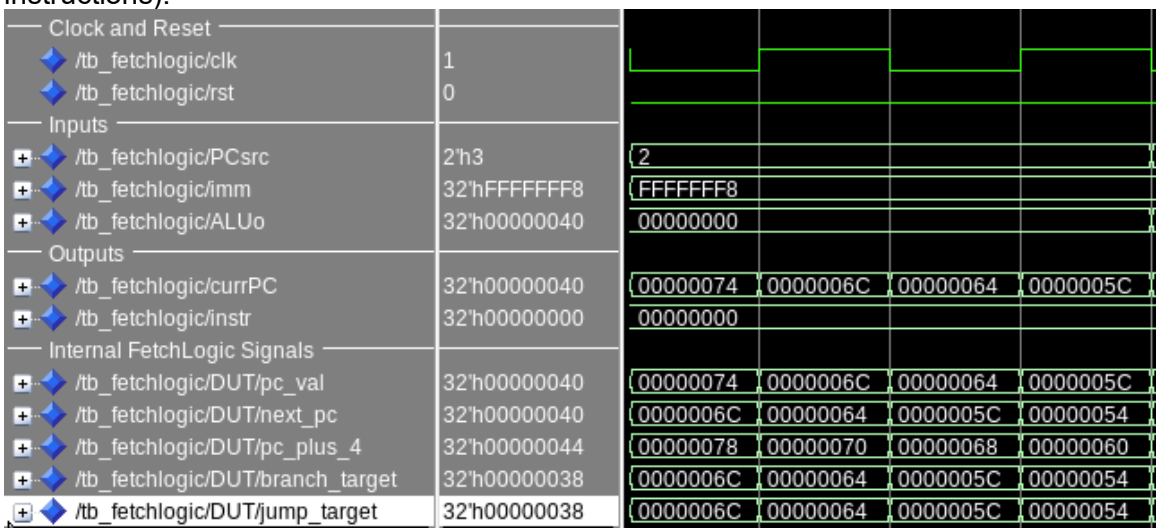
Test 1
When PCsrc = "00", the PC increments normally by 4 each clock cycle (PC_next = PC + 4), representing sequential instruction fetch.



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.

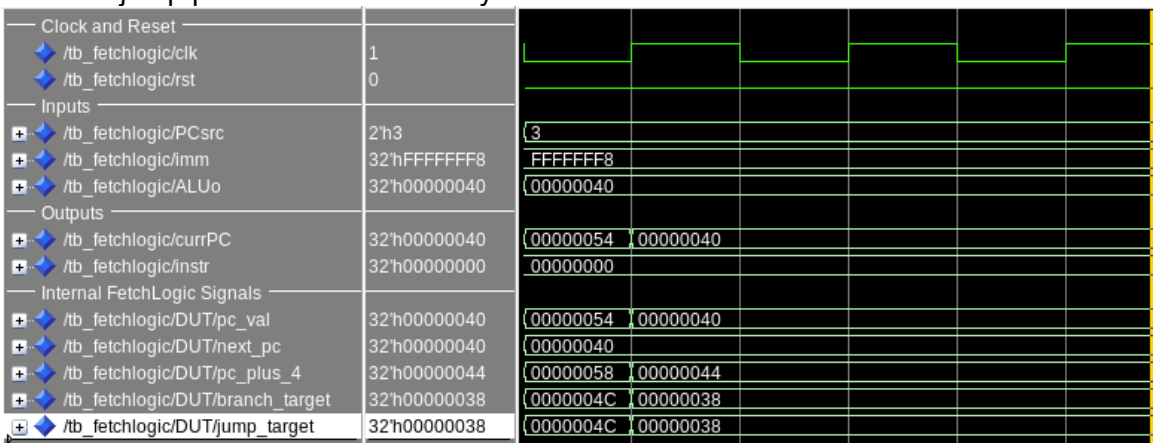


Test 3
or PCsrc = "10", the PC performs a jump using the immediate offset (PC + imm).
Since imm is negative, the waveform shows the PC jumping backward by 8 bytes (two instructions).



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.

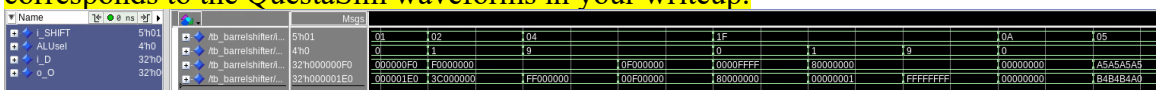
[Part 3.3.1.(b)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

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.

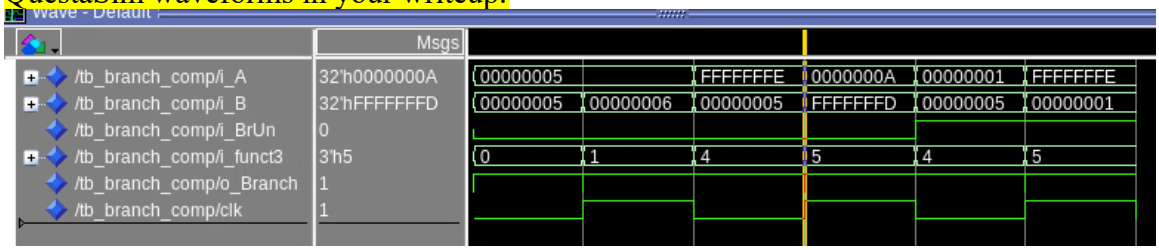


Test 1: Logical Left Shift. Test 2: Logical Right Shift. Test 3: Arithmetic right shift sign=1. Test 4: Arithmetic right shift sign=0. Test 5: Logical Left Shift edge case. Test 6: Logical right shift edge case. Test 7: Arithmetic right shiftedge case. Test 8: Zero input. Test 9: Pattern Test.

[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.

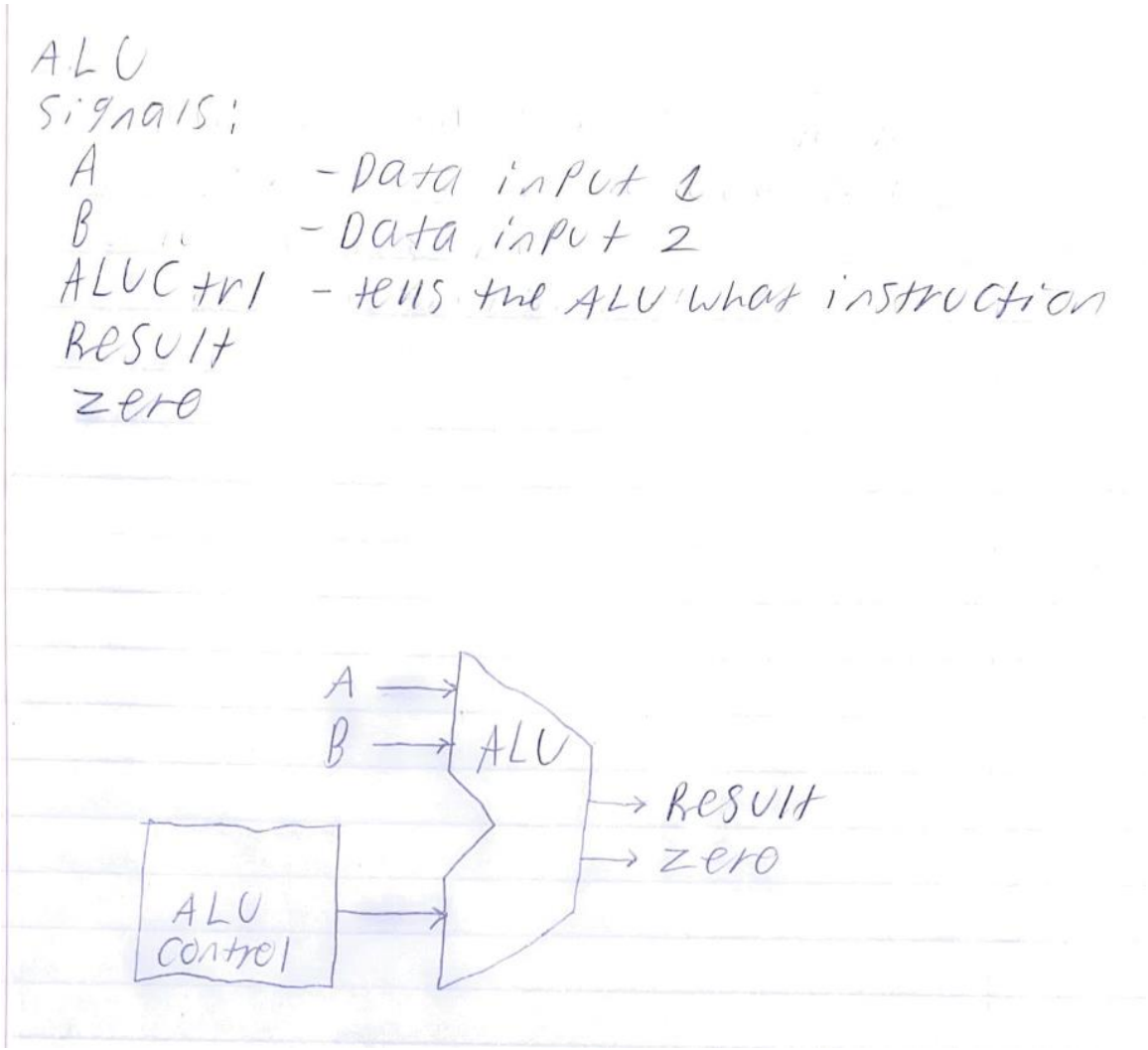
Our design approach was to try to compact everything into blocks to make it easier to read and to make it more organized. One way to help was to separate the branch logic from the ALU into its own block.

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



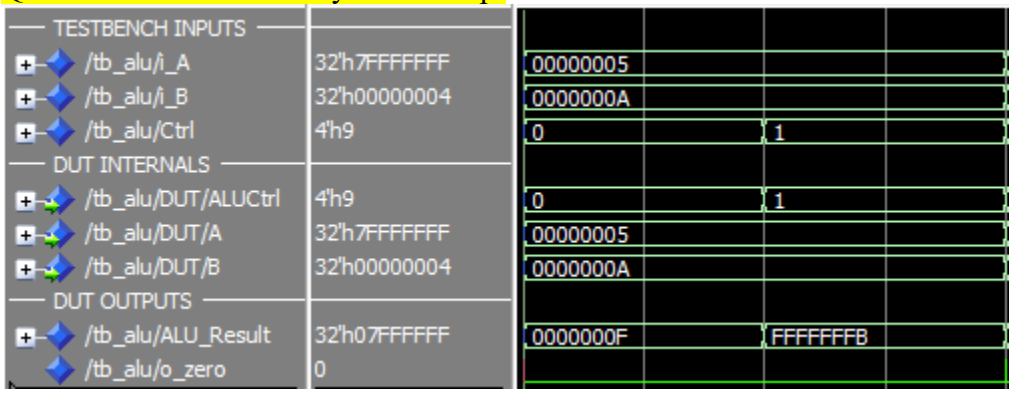
The branch comp will output a signal of 1 signaling it is a branch being executed and the BrUn will indicate if its signed or unsigned.

[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.

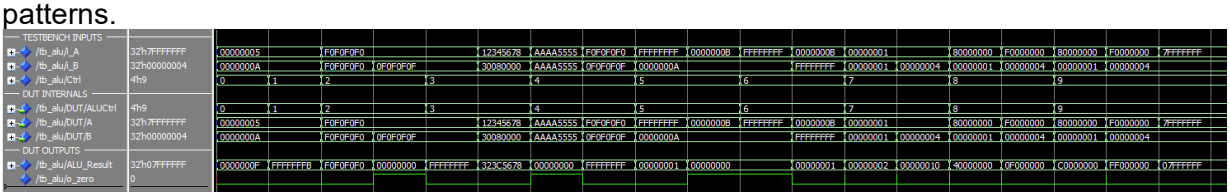


When the Ctrl signal is set to "0000", the ALU is expected to perform an addition operation. In this test case:
i_A = x"00000005"

i_B = x"0000000A"
The expected output of this addition operation is i_A + i_B = 5 + 10 = 15, which corresponds to x"0000000F" in hexadecimal.

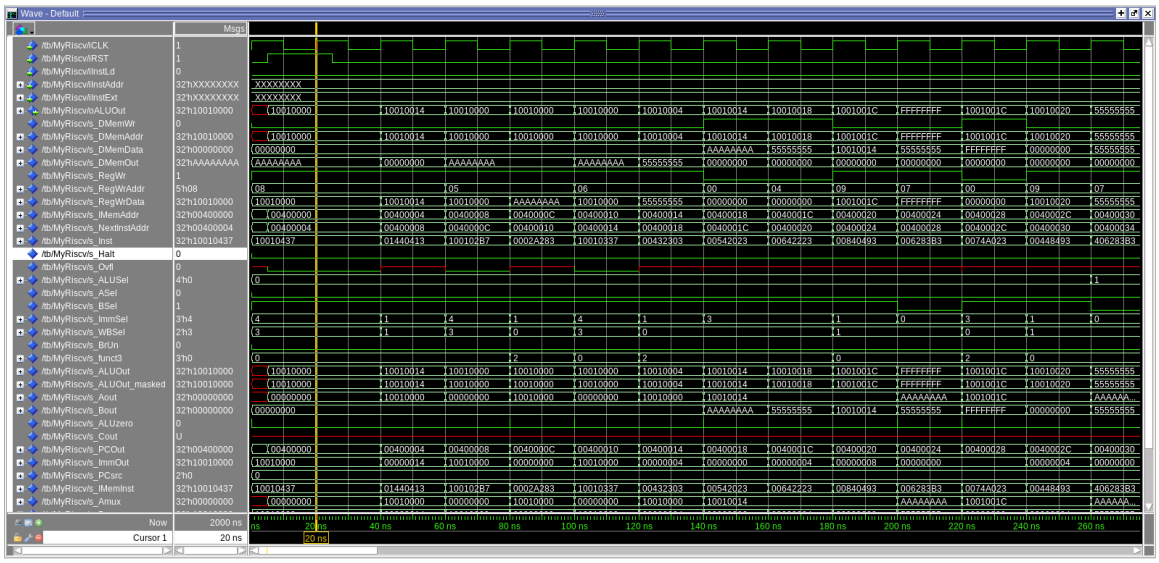
When Ctrl is set to "0001", the ALU performs a subtraction operation. In this case:
i_A = x"00000005"
i_B = x"0000000A"
The expected result of this subtraction is i_A - i_B = 5 - 10 = -5, which is represented as x"FFFFFFFB" in two's complement.

[Part 3.3.8] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.
This test plan is comprehensive because it covers a wide range of essential ALU operations, including arithmetic (addition and subtraction), bitwise logic (AND, OR, XOR), comparison (signed and unsigned less-than), and shift operations (SLL, SRL, SRA). Each test checks the ALU's ability to correctly handle both positive and negative numbers, ensuring proper functionality across different scenarios, such as two's complement representation for subtraction and sign extension for shifts. By verifying both simple and complex cases, like handling large values, sign preservation, and bitwise manipulation, the plan ensures that the ALU performs all necessary operations correctly, including handling edge cases like negative results, shifts, and different input patterns.



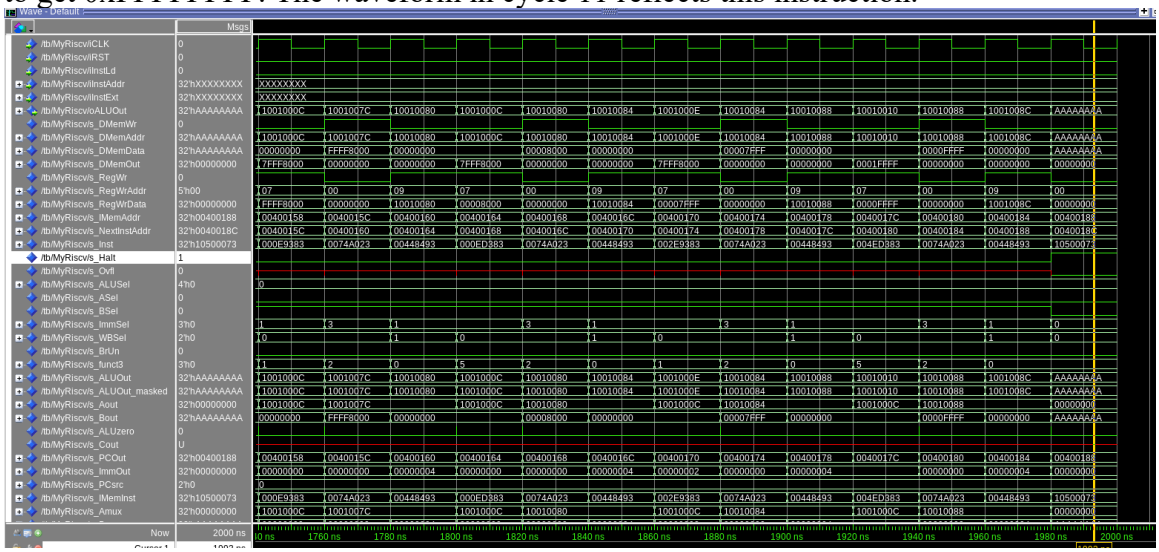
[Part 4] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[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.



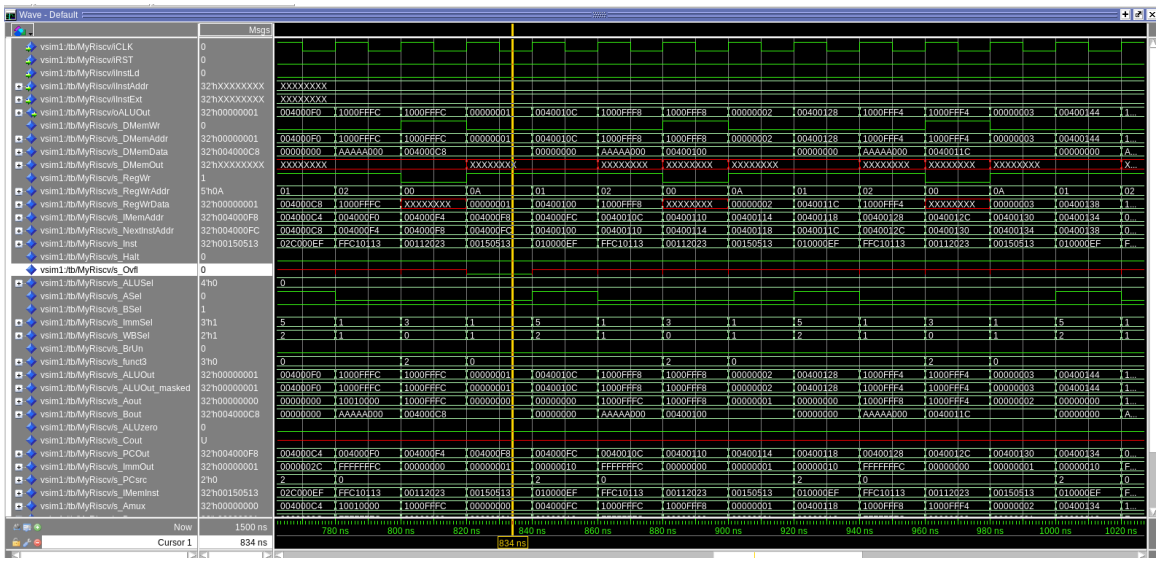
Proj1 base test screenshot^^initialization instructions

This screenshot contains the initial instructions and initializations for the base test. We know this has correctness due to the waveforms matching the RISCV test codes. For example: in cycle 11 we are adding t0 and t1 → 0xAAAAAAAA + 0x55555555 together to get 0xFFFFFFFF. The waveform in cycle 11 reflects this instruction.



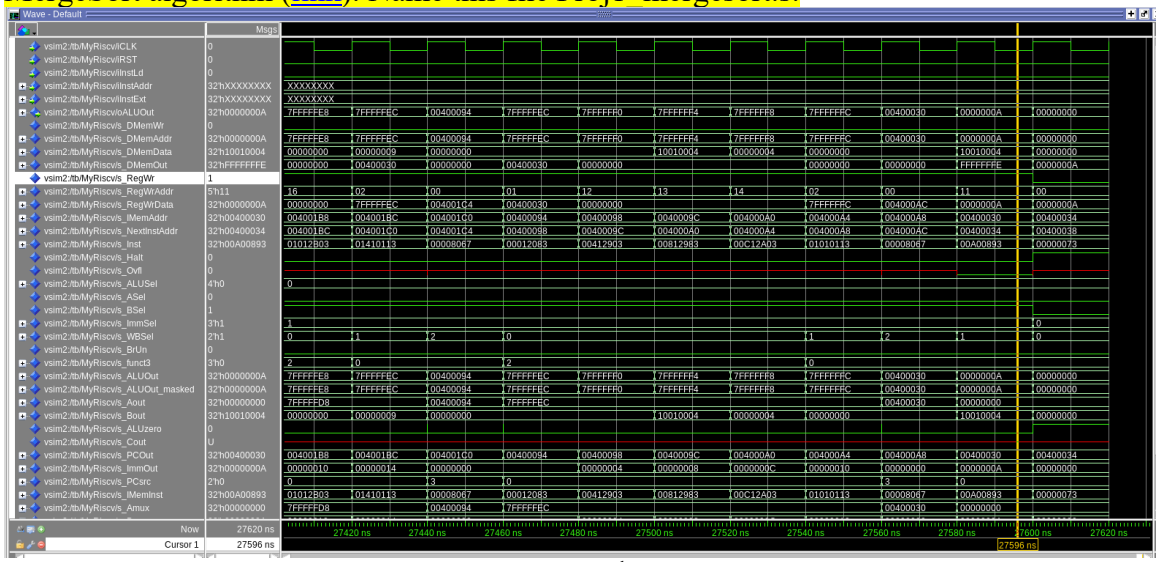
At the end of the test we trigger the s_Halt signal which recognizes the 1110011 Halt opcode from the RISCV ISA.

[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.



This screenshot contains the funcA #Call depth 1 in our test program. The cycle the line identifier is on corresponds with the addi a0, a0, 1 instruction in the call depth. This reflects that our processor is able to reach this point in the RISCV program and execute instructions such as jal and branching.

[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.



One way we can know it executed correctly is that the 13th instruction in IMEM should be fetched and called as the last instruction before the halt call. The waveform has 00A00893 as the instruction which mirrors the 13th instruction in IMEM and performs the li a7, 10 before the halt call.

[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?

Our critical path is generally the default critical path in most single cycle processors. The critical path is PC → Imem → ALU → DMEM → WBMux → RegFile(WriteBack). Since it is dictated by the lw and the lw has to take this path there isnt much to change.

The largest latency is the DMEM so that would be the best place to improve the frequency.