CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: Ayr Neto
Shobhit Singh

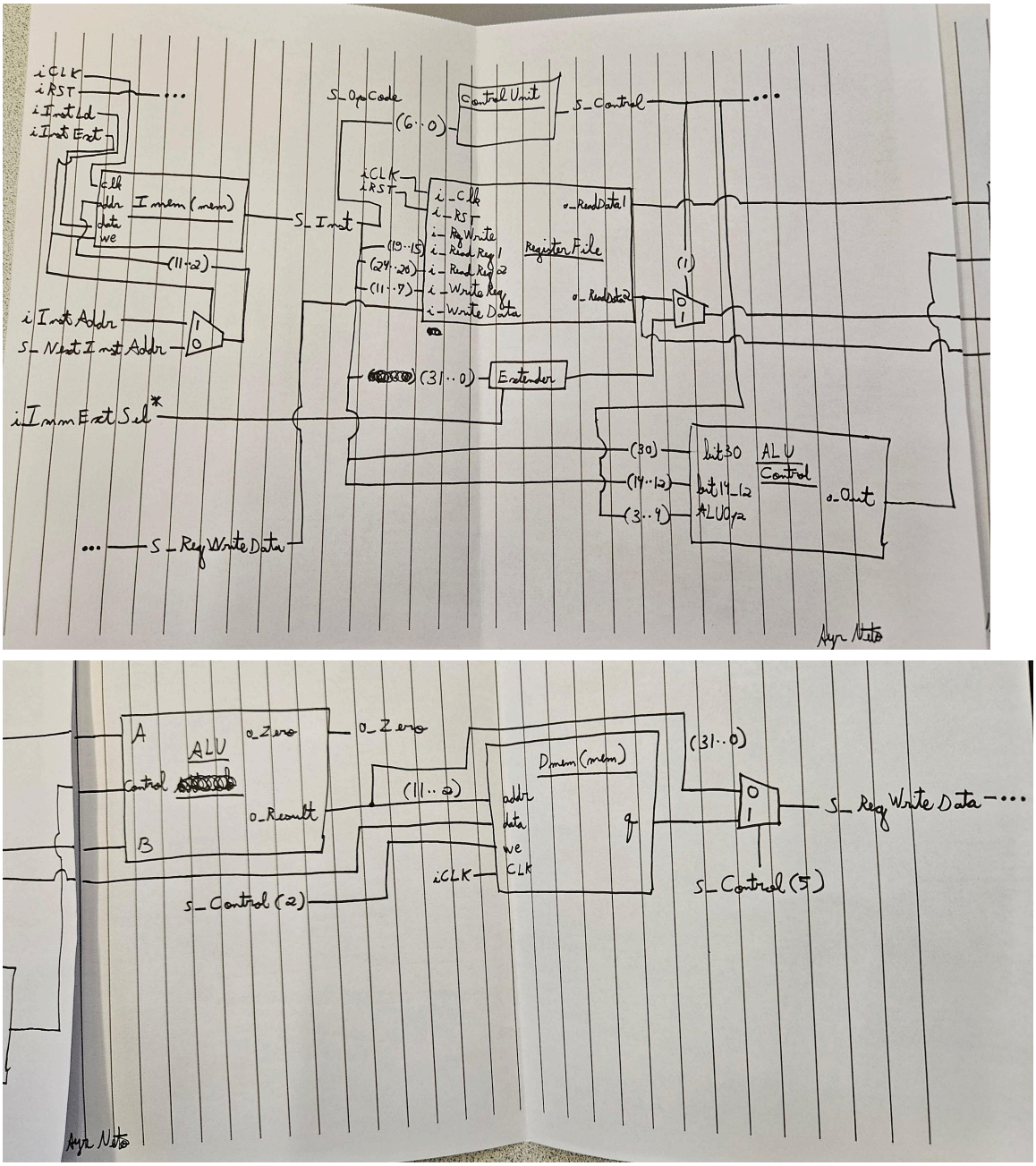
Project Teams Group #: 7

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.

Version 1: Not sure what iInstLd and iInstExt do exactly, there's no direct explanation. Without the Control Unit, I will need to directly input the instruction opcode (6 through 0) into the ALU, MUX's, etc.

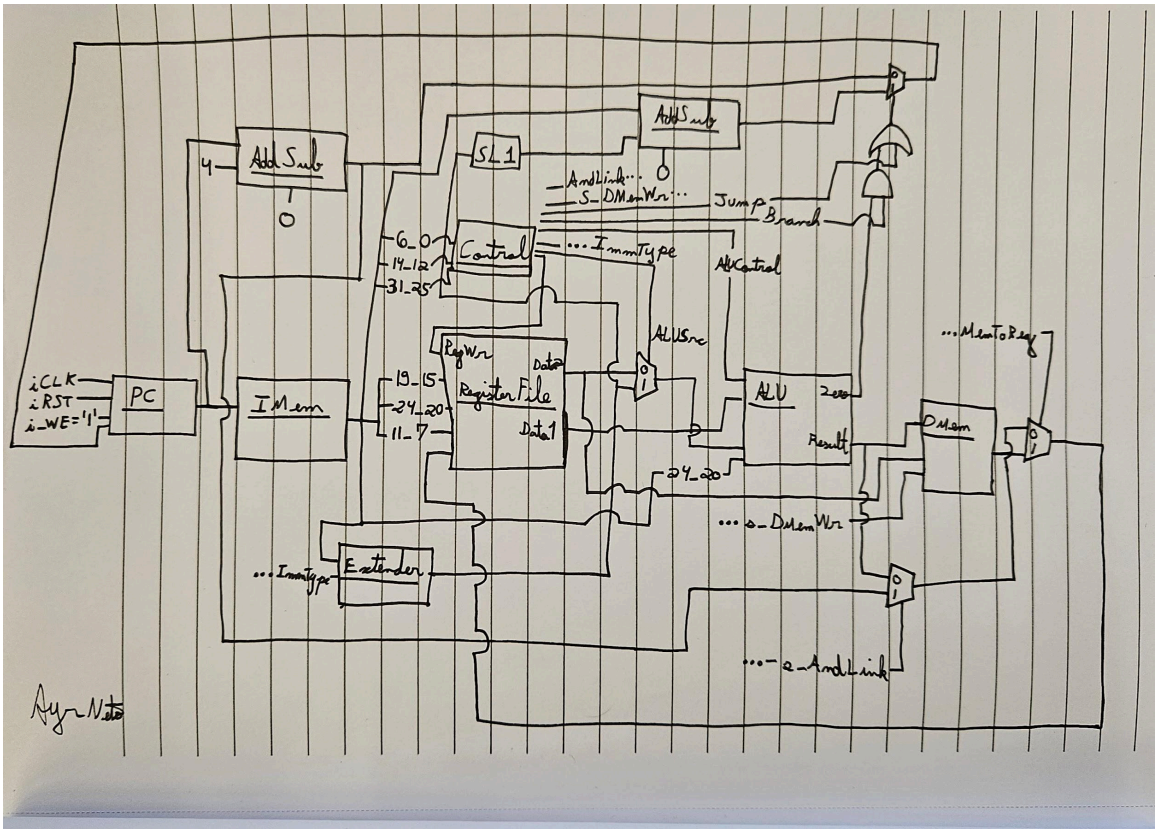
Version 2: Nevermind, I will add the Control Unit on the schematic. And should (surely) be easy to implement.
Currently not sure how to know when to sign vs zero extend (maybe I only need to sign extend). For now, using a Select for it.
Will also need to make a new ALU. The one with only 1-bit selector will not work (ALU Control outputs 4 bits…)



Version 3: Will change the Control Unit. Instead of 1 output, have multiple outputs (just to make reading easier), and add new output signals, such as Jump, AndLink, ImmType, and ALUControl (got rid of ALUOp signal and ALU Control component and will have the Control Unit deal with it).

The Control Unit will output a 3bit signal to be sent to the Extender to select which bits to extend in the instruction.

Version 4:
We will have the instruction bits 24 through 20 mapped to the ALU's 'shamt' input.

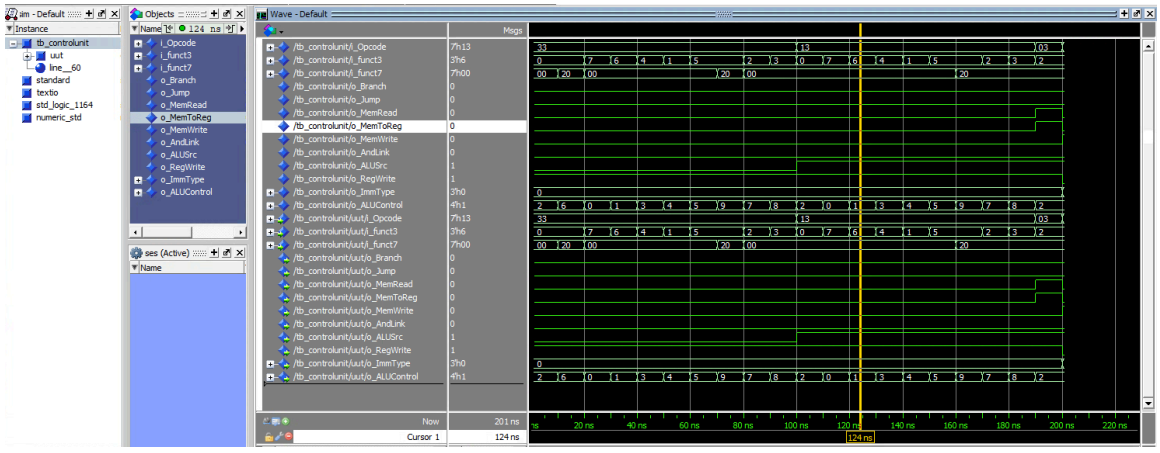


[Part 3.1.a.] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your 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.

Instruction	Opcode (Binary)	Funct3 (Binary)	Funct7 / Imm (Binary)	ALUSrc	ALUControl	ImmType	ResultSrc	MemWrite	RegWrite
addi	"0010011"	"000"	imm[31:20]	1 *(select extender output as B)*	2 *(ADD)*	"000" *(I-type sign-extend)*	0 *(take ALU result)*	0	1
andi	"0010011"	"111"	imm[31:20]	1	0 *(AND)*	"000"	0	0	1
ori	"0010011"	"110"	imm[31:20]	1	1 *(OR)*	"000"	0	0	1
xori	"0010011"	"100"	imm[31:20]	1	3 *(XOR)*	"000"	0	0	1
slli	"0010011"	"001"	imm[31:25]="0000000"	1	4 *(SLL)*	"000" *(treat shamt as I-type)*	0	0	1
srli	"0010011"	"101"	imm[31:25]="0000000"	1	5 *(SRL)*	"000"	0	0	1
srai	"0010011"	"101"	imm[31:25]="0100000"	1	6 *(SRA)*	"000"	0	0	1
slti	"0010011"	"010"	imm[31:20]	1	7 *(SLT signed)*	"000"	0	0	1
sltiu	"0010011"	"011"	imm[31:20]	1	8 *(SLT unsigned)*	"000"	0	0	1

[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).

ControlUnit.vhd implements a purely combinational decoder: each opcode/funct3/funct7 triple selects a constant bundle for ALUSrc, ALUControl, ImmType, MemToReg, MemRead, MemWrite, RegWrite, Branch, Jump, and AndLink. The unit test in tb_ControlUnit.vhd iterates through every R-, I-, load/store, branch, jump, and U-type pattern from the spreadsheet and asserts equality against the expected control vector; the QuestaSim trace shows all outputs toggling deterministically in zero time.

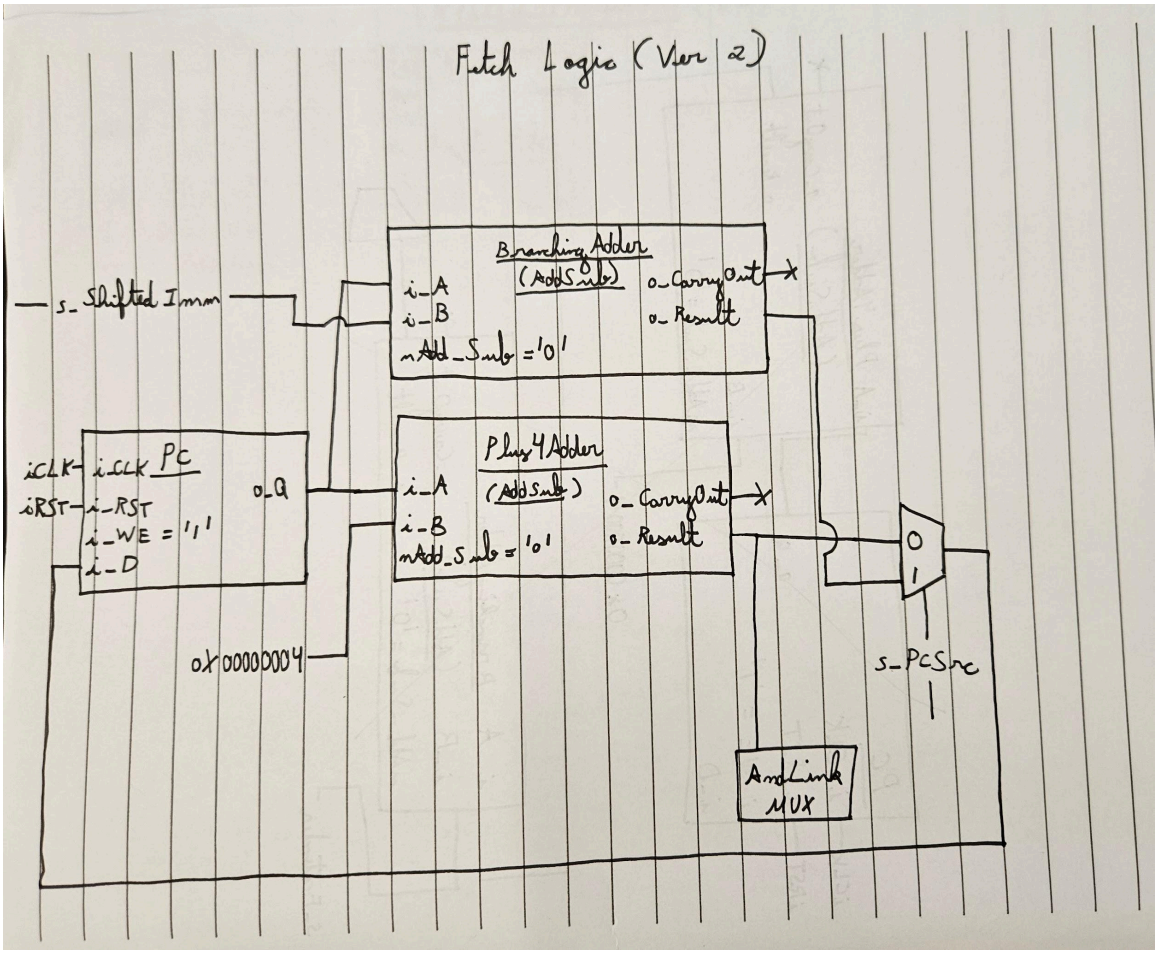


[Part 3.2. (a)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.
For the fetch logic, we need a “Plus4Adder” that adds 4 to the output of the PC, used for “normal” instructions. We also need another adder to add the PC output and the instruction (after extended), used for branching and jumping instructions.

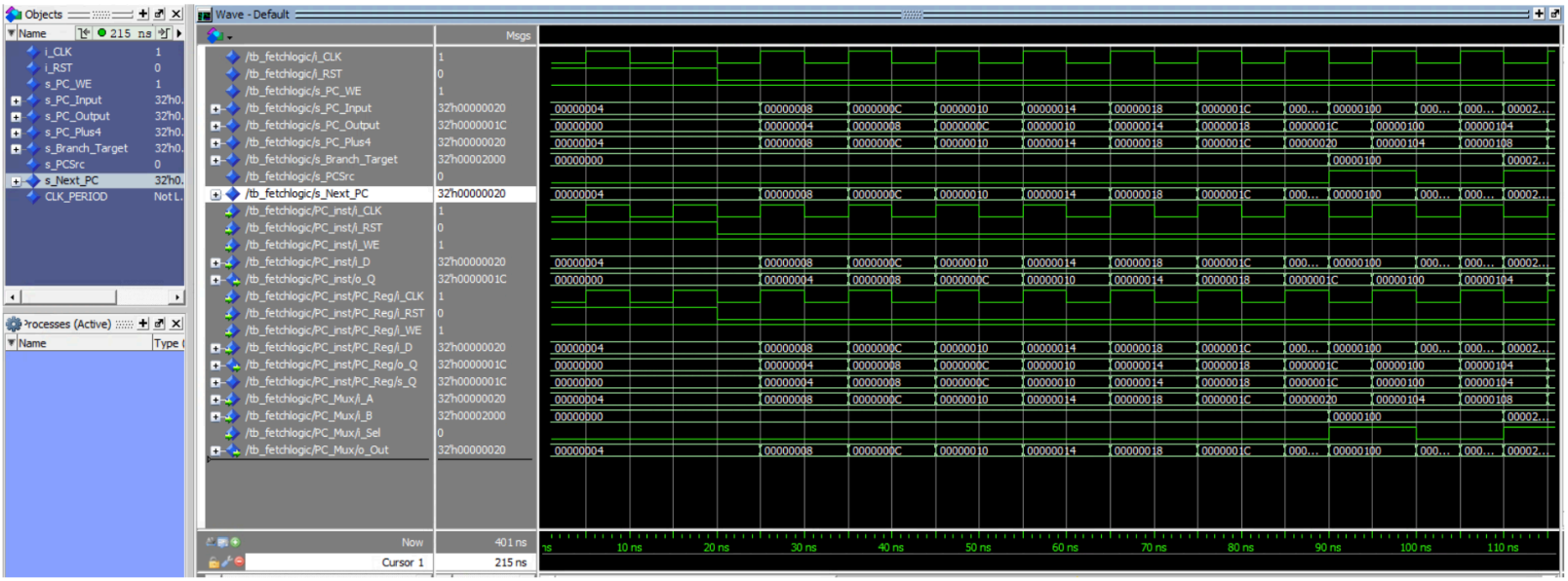
Each adder goes to a MUX that selects the PC’s next instruction address with Control Unit’s Branch signal and the ALU’s Zero signal. If both signals are 1, the MUX selects the second adder’s output.

However, we will need to figure out a way to implement ‘jalr’ because that will require Register Read Data + Immediate offset to be sent to the PC as address.
Possible solution: add another MUX, that takes the “Plus4Adder”’s output and the ALU’s result output and select using a signal called “AndLink” to link back the value.

[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.



Fetch logic lives in RISCV_Processor.vhd (PCSrc_MUX, s_JumpTarget, s_LinkData). tb_FetchLogic.vhd drives reset, straight-line increments, branch taken/not taken, jump, jalr, and write-enable stall; waveforms show PC stepping by 4, redirecting to the programmed target, holding when WE=0, and resetting to 0 on i_RST.

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

Logical shifts replaces the left bits with 0's after shifting. Arithmetic shifts the left bits with the sign value (the original number's left-most bit) after shifting.

There's no need for 'sla' because 'sll' would do the same thing: fill the right bits with '0's after shifting.

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

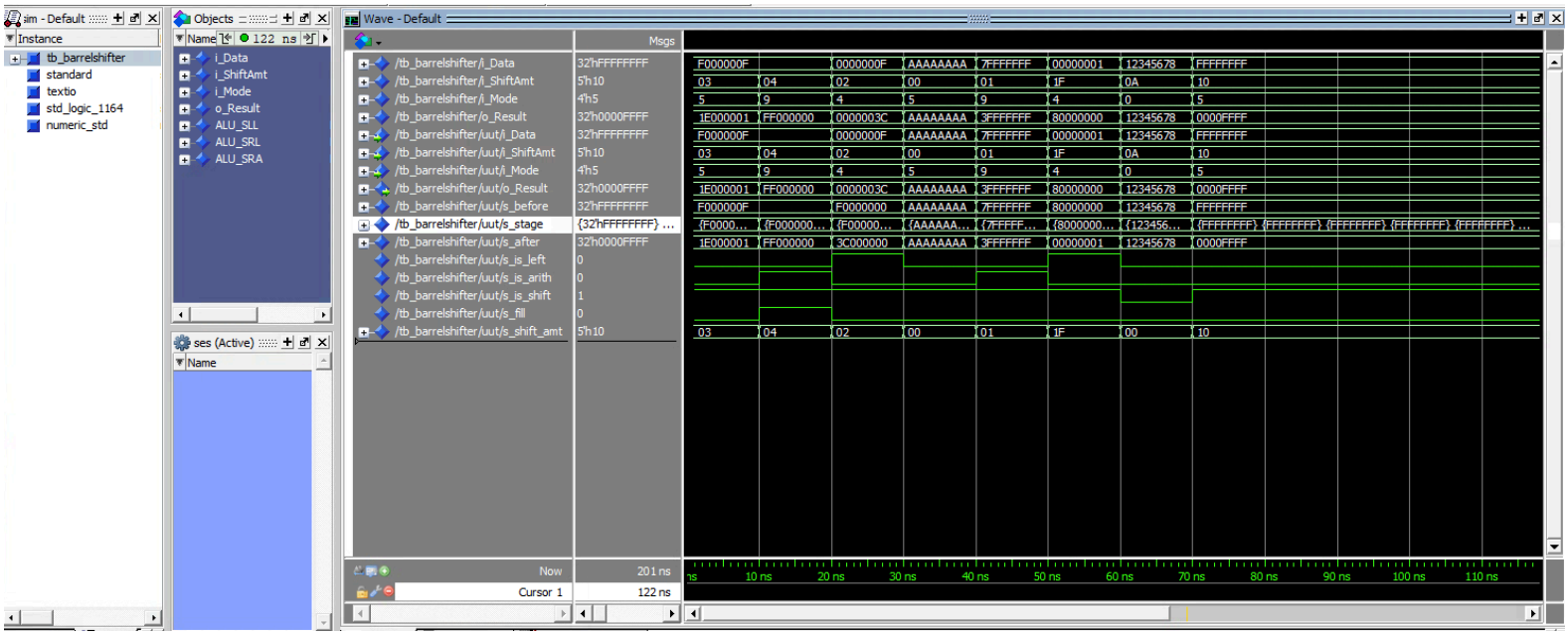
BarrelShifter.vhd decodes i_Mode: logical shifts zero-fill; arithmetic right shifts set s_fill to the sign bit. Cascaded 2:1 mux stages shift by 1, 2, 4, 8, and 16 positions under control of i_ShiftAmt.

[Part 3.3.1.(c)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

The same right-shift network handles left shifts by reversing the operand bits on entry and reversing the output (reverse_bits function) when ALU_SLL is selected.

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

In tb_BarrelShifter, the waveform shows s_Result zero-filling for SRL, sign-extending for SRA, and matching the reversed-right-shift for SLL. Annotate each time slot with the expected 32-bit literal.



[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.

We will have the instruction bits 24 through 20 mapped to the ALU's 'shamt' input.
For the Zero output, in the ALU if the final result is equal to all 0's, Zero will equal '1', otherwise it's '0'. It can be done with gates, but behavioral is easier.
I also modified a little bit of the AddSub component to allow the SLT instruction: if the nAdd_Sub signal is "11", it signals a SLT instruction, and the left-most bit is used to decide if the number is either negative or not.

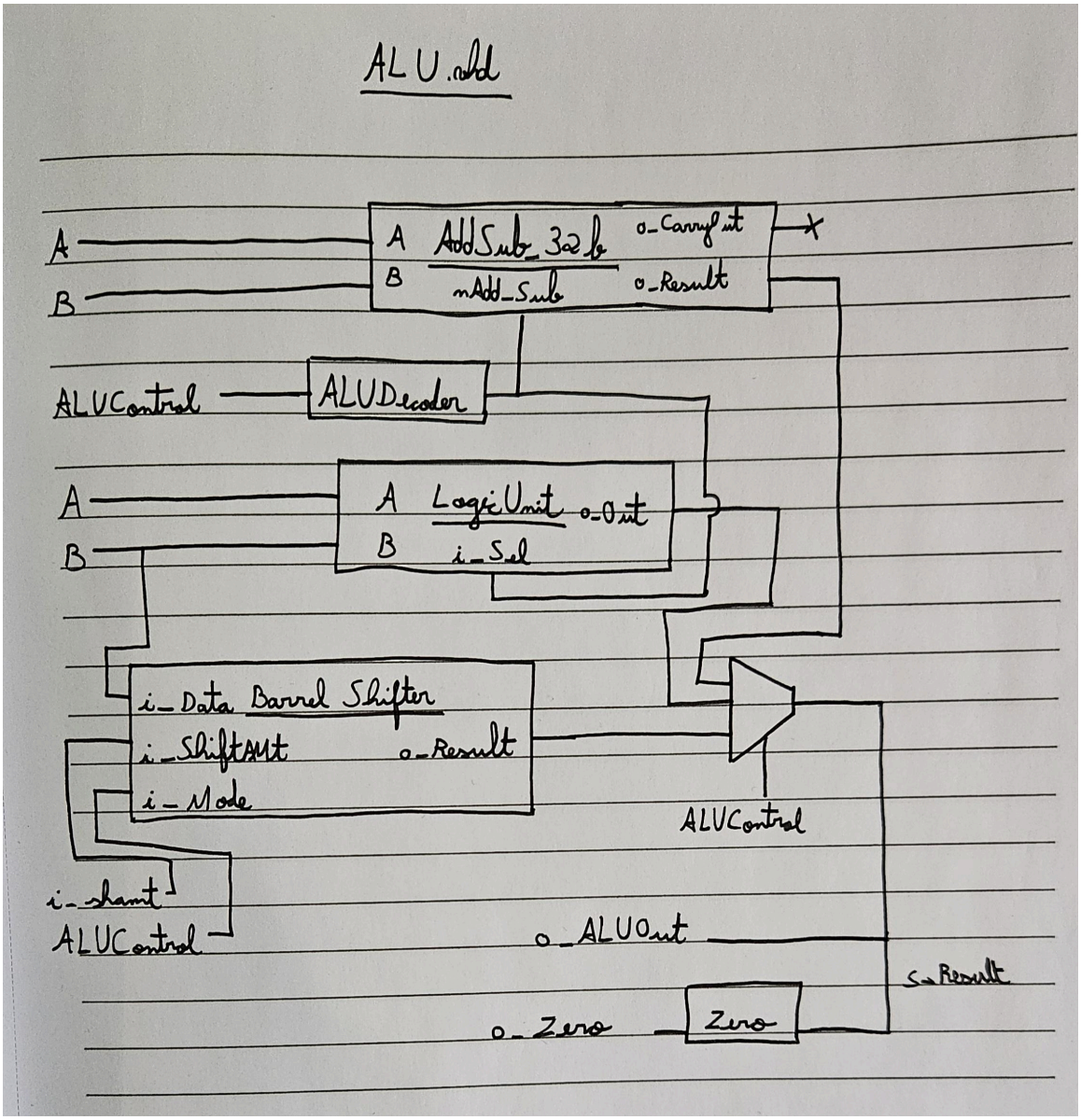
[Part 3.3.2.(b)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.
tb_ALU.vhd sequences adds, subs, logical ops, shifts, and compares; the captured traces confirm o_ALUOut and derived flags match expectations for positive/negative operands and edge cases (e.g., SRA of a negative number, SLTU with wrap-around).

[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?

For the Zero output, in the ALU if the final result is equal to all 0's, Zero will equal '1', otherwise it's '0'.

For SLT, we assign all bits to '0' and the right-most bit to the sign bit of the original result.

For SLTU, we assign all bits to '0' and the right-most bit assigned to the inverse of the CarryOut output.



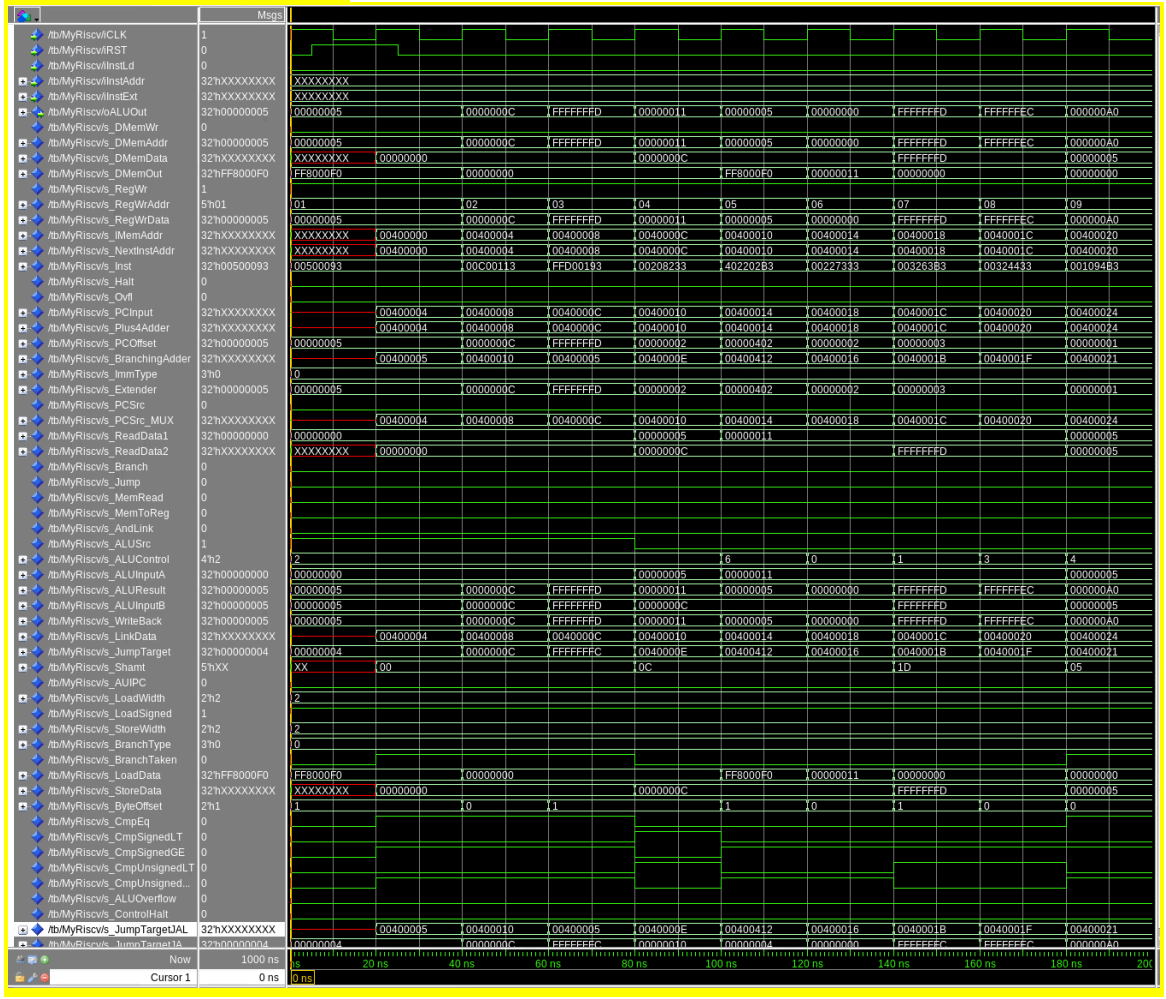
[Part 3.3.5] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

tb_ALU.vhd sequences adds, subs, logical ops, shifts, and compares; the captured traces confirm o_ALUOut and derived flags match expectations for positive/negative operands and edge cases (e.g., SRA of a negative number, SLTU with wrap-around).

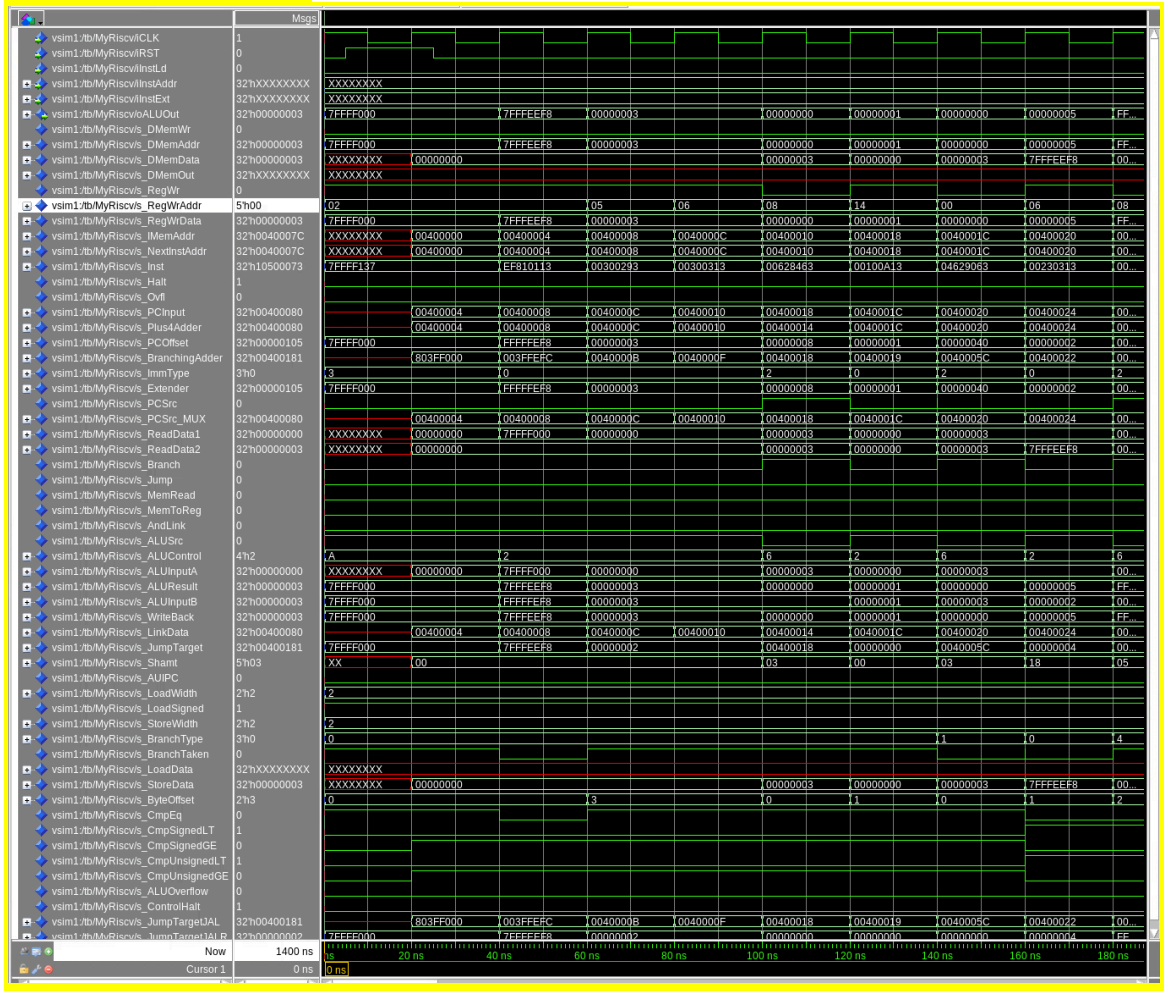
[Part 3.3.8] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.
Unit benches (tb_ControlUnit, tb_BarrelShifter, tb_ALU, tb_Extender, tb_FetchLogic) cover module-level corner cases; the three architectural programs (Proj1_base_test.s, Proj1_cf_test.s, Proj1_mergesort.s) exercise full-system arithmetic, control flow (depth ≥ 5), loads/stores, and recursive memory usage. Toolflow runs confirm the processor's register/memory traces match RARS, providing coverage across the required ISA subset.

[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.

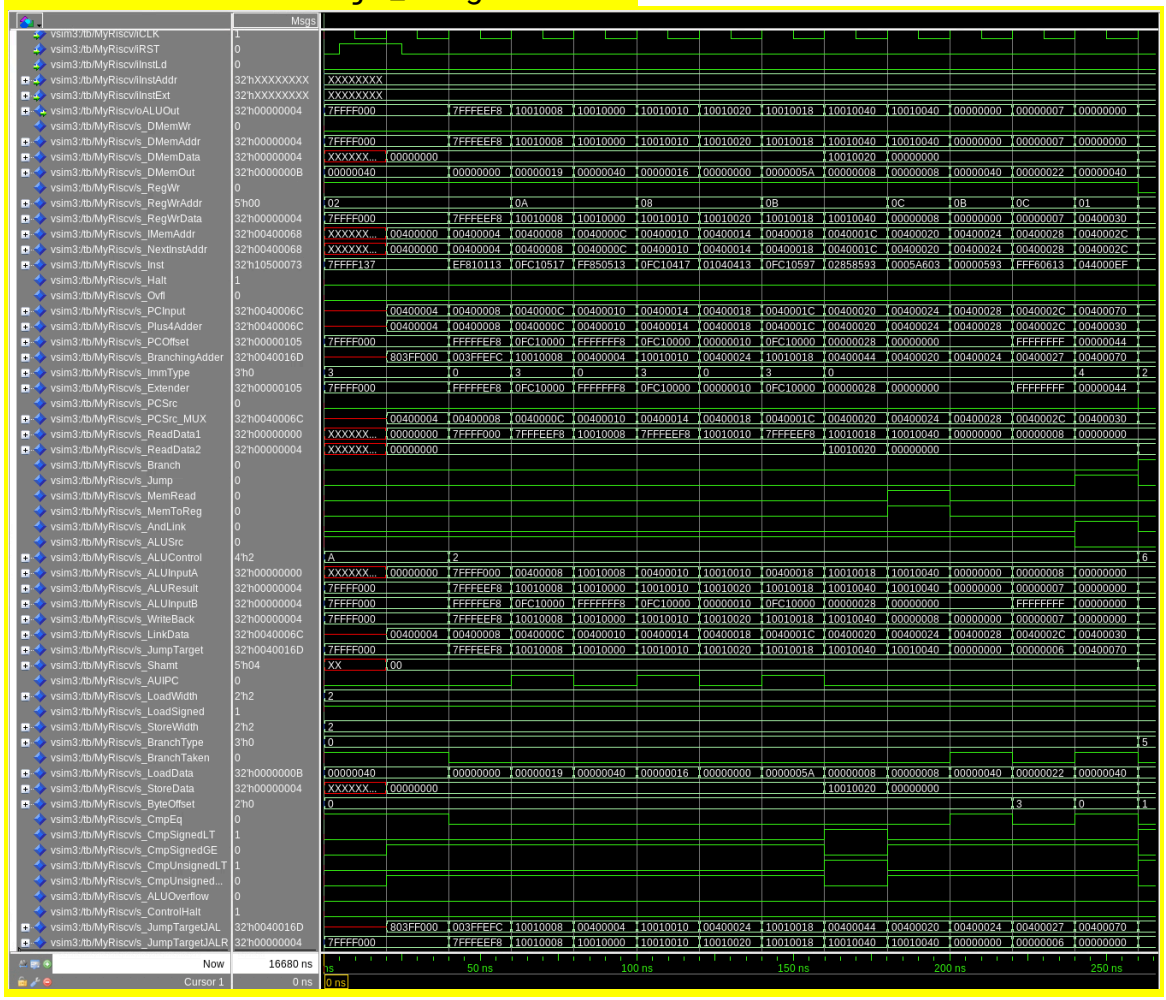
[Part 4.a] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 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?

Maximum frequency: 17.75 MHz (critical-path arrival 59.693 ns versus a 20 ns requirement, slack -36.325 ns)

Critical path: PC bit 8 (ProgramCounter) → instruction memory address decoding (IMem) → register file read chain → ALU barrel shifter (ALU_inst|Shifter_Unit) → ALU result

multiplexing (ALU_inst|Mux24) → data memory
(DMem|ram~5346x, ultimately mem:DMem|ram~173).

Frequency improvement:

1. reduce ALU shifter depth
2. add a register between the ALU result and data memory
   write data to break the combinational chain