CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: _____Drew Swanson_____

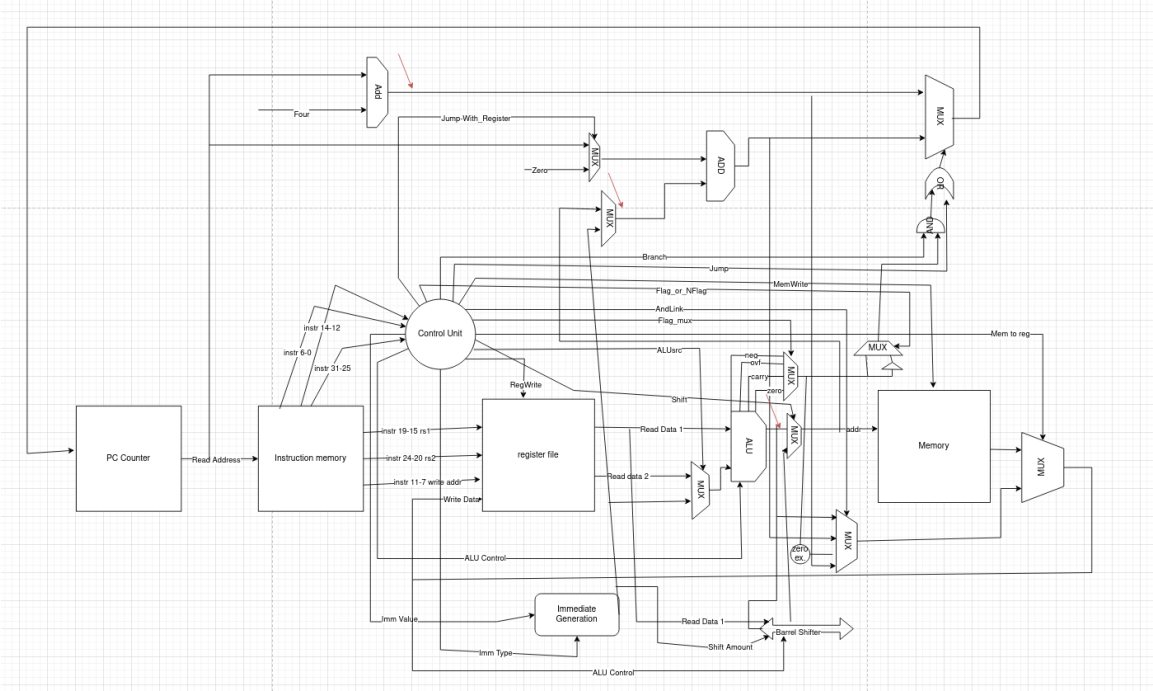
_____Anthon Worsham_____

_____

Project Teams Group #:_____B3_____

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

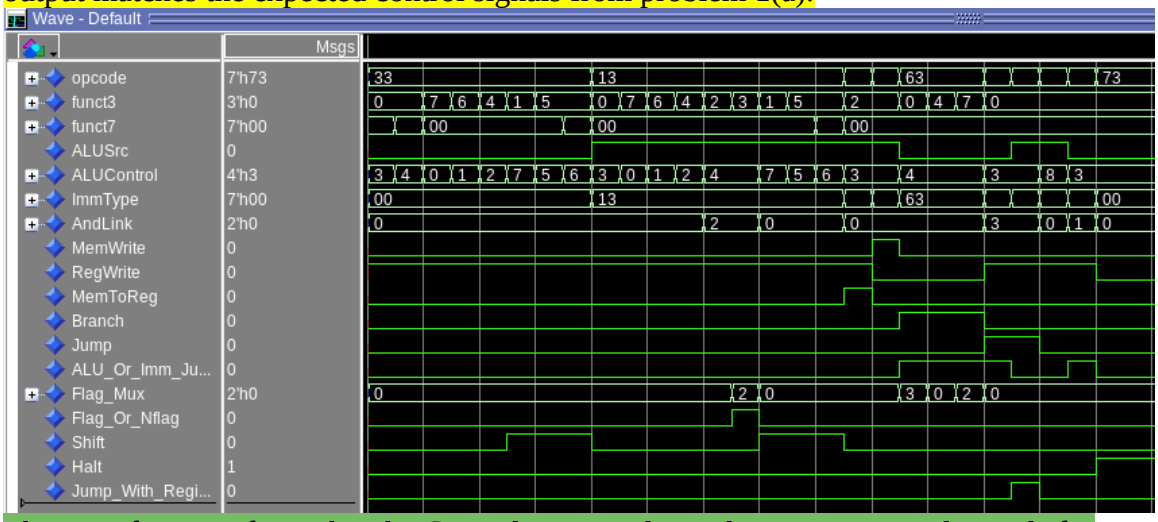
[Part 2 (d)] Include your final RISC-V processor schematic in your lab report.



[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.

See submissions folder

[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).

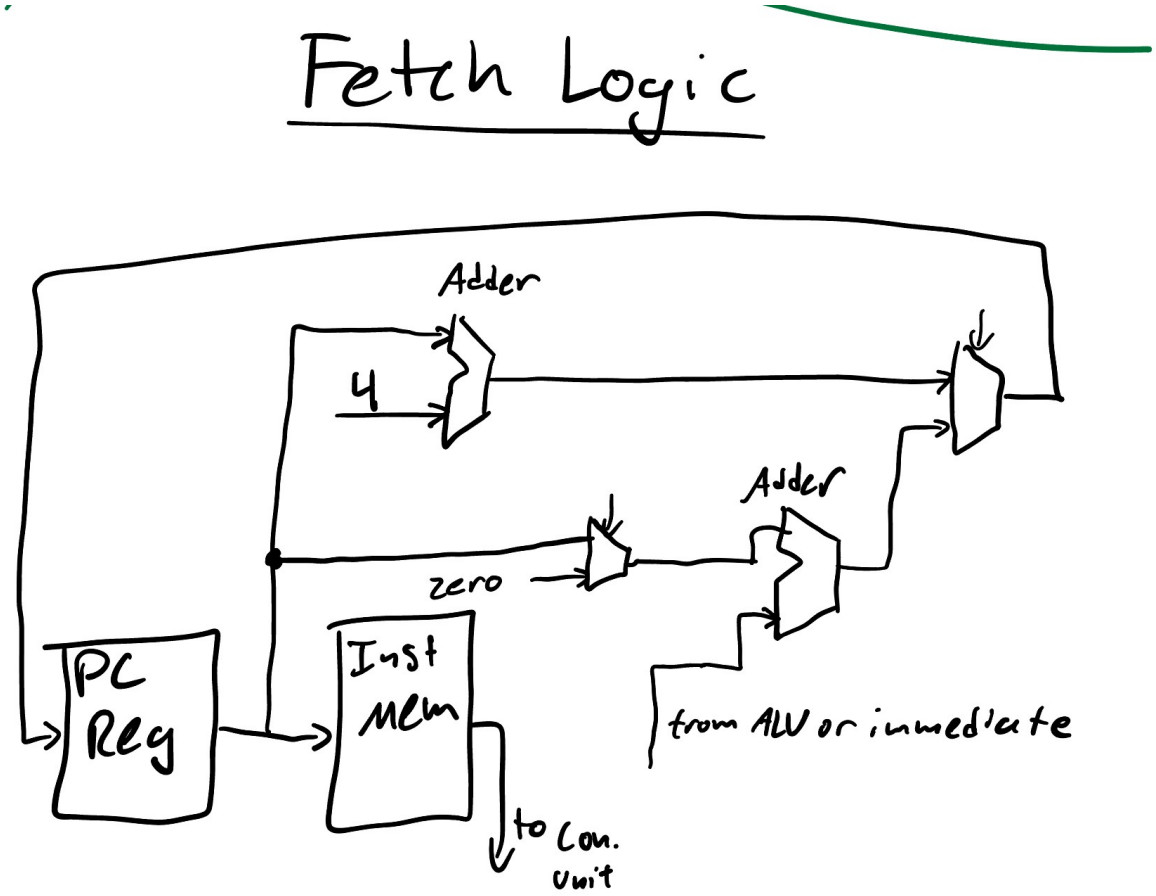


The waveform confirms that the Control Unit produces the correct control signals for each instruction type from Problem 1(a). R-type operations (opcode = 0x33) show RegWrite = 1, ALUSrc = 0, and the proper ALUControl codes for ADD, SUB, logic, and shift instructions. I-type instructions (0x13) correctly set ALUSrc = 1 for immediate inputs, with Shift = 1 only for shift-immediate operations. Load (0x03) and store (0x23) instructions use ALUSrc = 1 for address generation, distinguishing MemToReg = 1 for loads and MemWrite = 1 for stores. Branch instructions (0x63) assert Branch = 1, ALUControl = CMP, and vary Flag_Mux based on funct3. Jump instructions (0x6F and 0x67) set Jump = 1, RegWrite = 1, and the correct AndLink or Jump_With_Register signals. LUI (0x37) and AUIPC (0x17) generate the expected ALU control values and link behavior, and HALT (0x73) cleanly asserts Halt = 1.

[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.

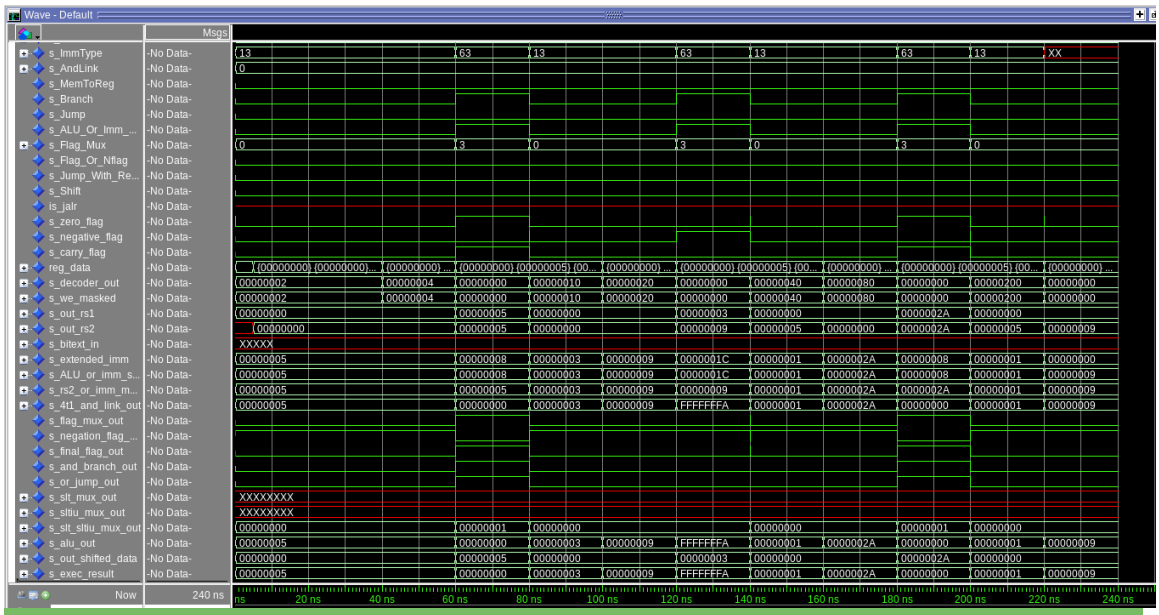
Some control flow possibilities are:
- Sequential (R/I/loads/stores/LUI/AUIPC)
  PC = PC+4
- Conditional Branch
  PC = PC + B-imm. This compares two registers and branches based on the comparative
- Unconditional Branch
  PC = PC + J-imm. This jumps to a value specified, does not have to compare any values to jump.

[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?



We had to add ALU_Or_Imm_Jump to facilitate jumping to an address specified by an immediate, or if changing the PC relies on the ALU's output. We added Flag_Mux to pick between any of the four different flags generated by the ALU: neg, ovf (even though it's never used in RISCV), carry and zero. We added Flag_Or_Nflag for some branch statements that rely on the negation of a flag for function. Finally, we added Jump_With_Register for the sole purpose of jalr, so that it can properly distinguish what address to jump to from a register.

[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.



The waveform shows the **beq** instruction, and how the processor's control flow responds to this specific instruction. When s_Branch goes high, the PC selects the branch target using the ALU output. When s_Jump asserts, the PC jumps to the immediate target, and s_Jump_With_Register indicates a register-based JALR jump. The alternating patterns in s_ImmType, s_AndLink, and s_ALU_Or_Imm_Jump align with instruction decoding, confirming that control signals switch correctly between sequential, branch, and jump operations. Corresponding updates in s_out_rs1, s_out_rs2, and s_exec_result verify that the datapath and control unit work together.

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

IMPORTANT: OUR BARREL SHIFTER IS OUTSIDE THE ALU. The difference between these two is that srl fills the leftmost bits with zero when shifting, while sra fills the leftmost bits with the sign bit, thus keeping the original sign of the value. There is no sla because the leftmost bits are lost anyways, and the rightmost bits are filled with zeros anyways as well.

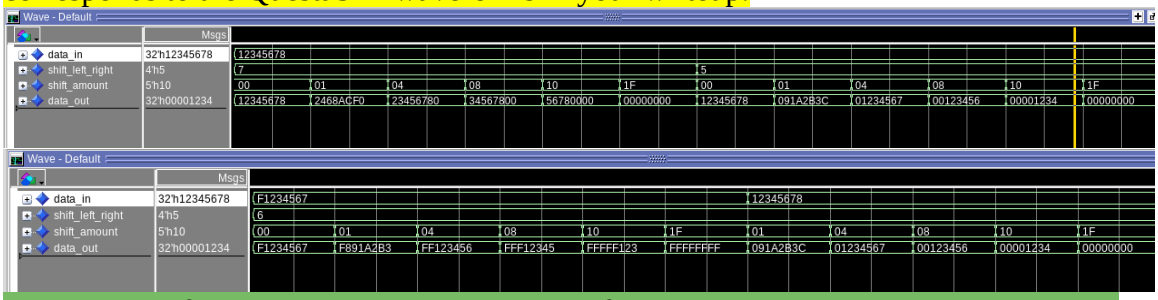
[Part 3.3.1.(b)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

Our vhdl module implements the shifter by utilizing a five stage shifting network controled by five bits of the shift_amount input. It precomputes what the data would look like if it were shifted by that amount, and a mux either passes the data through or applies the shift depending on the corresponding bit in shift_amount. This can do any shift from 0 to 31 bits and can do both arithmetic (fill with sign bit) and logical (fill with zeros)

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

We would mirror the bit movement, but instead of moving bits right they would move toward the most significant end.

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

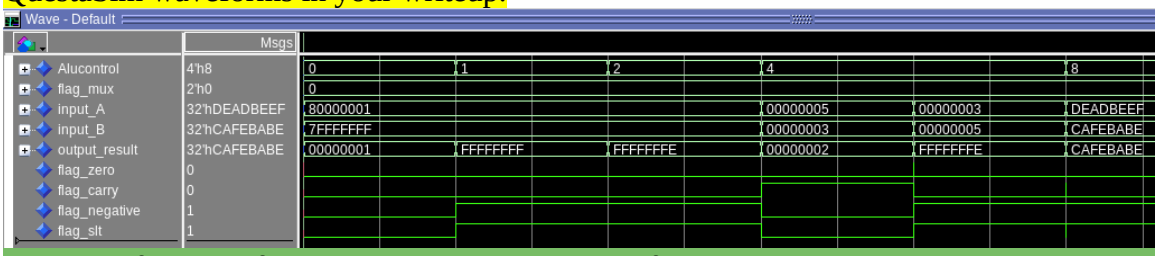


The barrel shifter behaves exactly as expected for SLL, SRL, and SRA. With data_in = 0x12345678 and SLL selected, data_out advances left by the programmed amounts (e.g., sh=0 → 12345678, sh=1 → 2468ACF0, sh=4 → 23456780, sh=8 → 34567800, sh=16 → 56780000, sh=31 → 00000000). Switching to SRL on the same input shows zero-fill right shifts (sh=0 → 12345678, sh=1 → 091A2B3C, sh=4 → 01234567, sh=8 → 00123456, sh=16 → 00001234, sh=31 → 00000000). For SRA, using a negative operand (data_in = 0xF1234567, MSB=1) demonstrates correct sign extension (sh=1 → F891A2B3, sh=4 → FF123456, sh=8 → FFF12345, sh=16 → FFFFF123, sh=31 → FFFFFFFF), while a positive operand (0x12345678, MSB=0) matches the SRL results (e.g., sh=1 → 091A2B3C, sh=4 → 01234567, sh=8 → 00123456, sh=16 → 00001234, sh=31 → 00000000).

[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.

The shared bus + final mux keeps the critical path short and predictable, The small pre-mux lets us encode control compactly which simplifies the controller, and the component blocks are easy to unit test. It is fixed at 32 bits because riscv only handles 32 bit words, so using an N bit is unneeded. Zero is computed with a vector compare. The barrel shifter is external to the ALU.

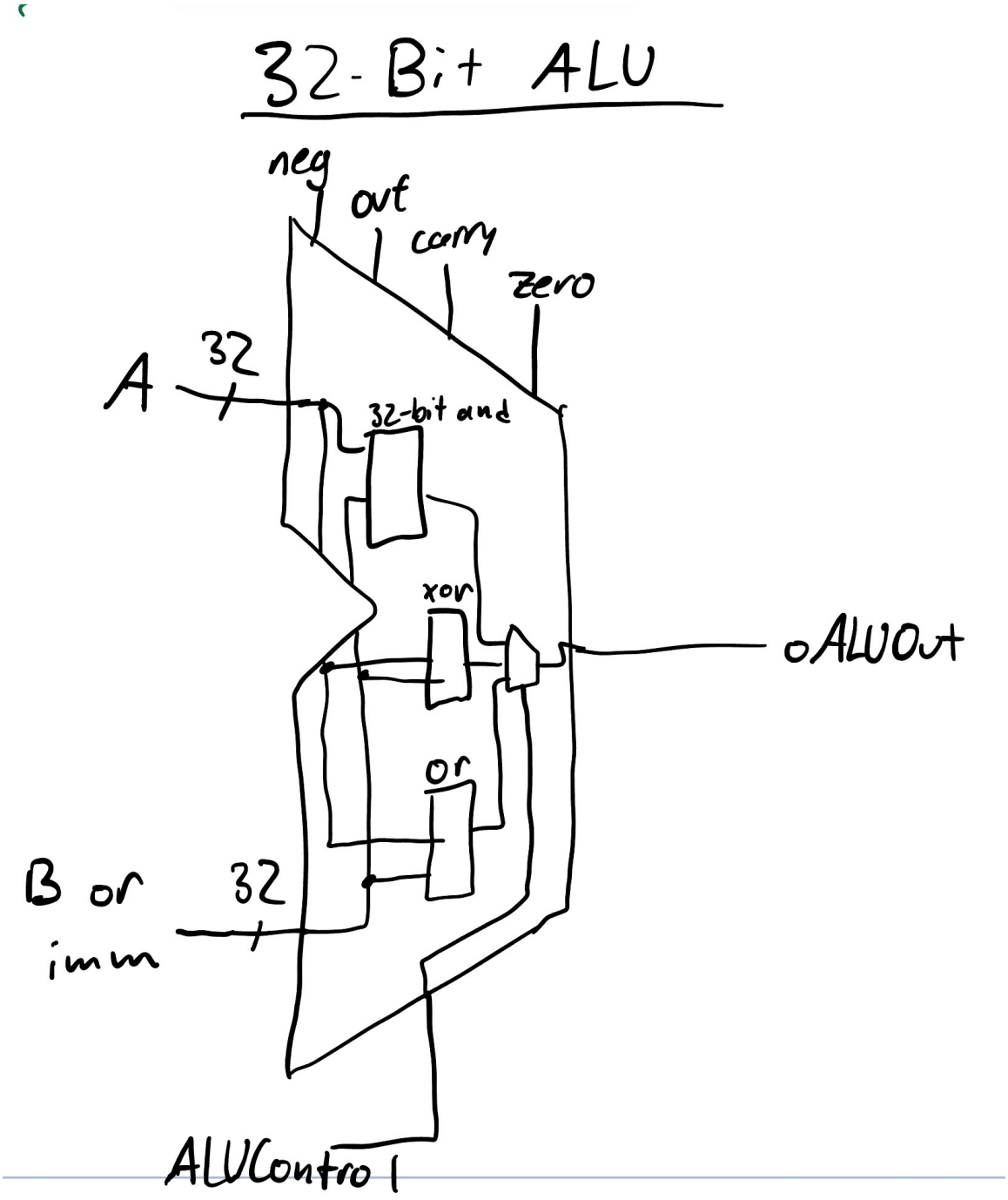
[Part 3.3.2.(b)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.



The waveform confirms correct ALU operation for all tested control codes. When the Alucontrol signal changes, the output_result and flag outputs respond accordingly within a 10 ns delay. Logic operations (AND, OR, XOR) correctly produce bitwise results and update the negative and zero flags based on the output's sign and magnitude. The

subtraction cases show proper arithmetic behavior, with flag_carry high when no borrow occurs and flag_slt indicating signed less-than. Finally, the pass-through instruction (Alucontrol = 1000) bypasses the ALU and outputs input_B directly, verifying control path selection.

[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?



The zero flag is calculated by subtracting one value from the other, and if the result is zero, the the zero flag is made to be ‘1’. Slt is implemented by doing the same thing, but if the result of rs1 – rs2 is negative, then rs1 is less than rs2 and the slt flag is made to be ‘1’. If not, it is set to ‘0’.

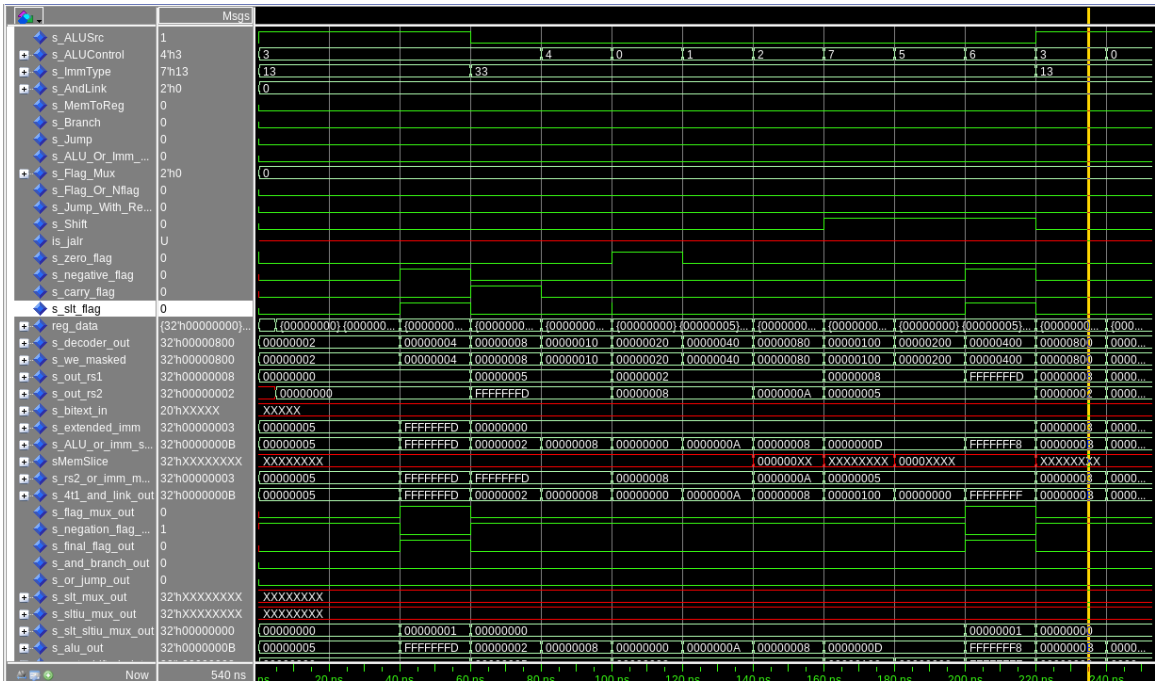
[Part 3.3.5] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

The ALUControl steps through **0000, 0001, 0010, 0100 and 1000,** thereby hitting all of the functions that the ALU can complete. First, AND (0000) is tested, with the output being the and of 80000001 and 7FFFFFFF: 00000001. Then, OR is tested, with the output being FFFFFFFF. After that, XOR is tested, with the output being FFFFFFFE which is correct because the only bit that is similar is the LSB, so the output for that bit is 0. Then SUB is tested, doing 5-3=2. Then the negative version is tested, 3-5=-2. Finally, the LUI/passthrough is tested. This just makes sure that when **1000** is selected, the input of the alu matches the output, having no operations done on it.

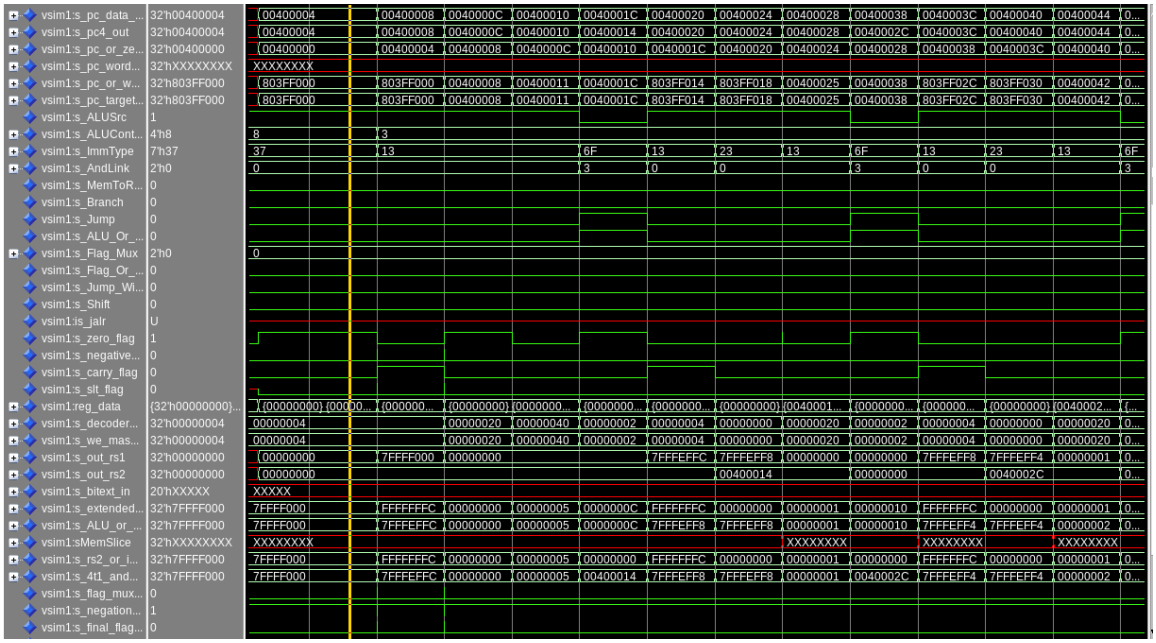
[Part 3.3.8] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.
The above test plans are comprehensive because they test that all (slt, sra, srl, add, xor, sub, or) functionality works as expected and as designed.

[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.

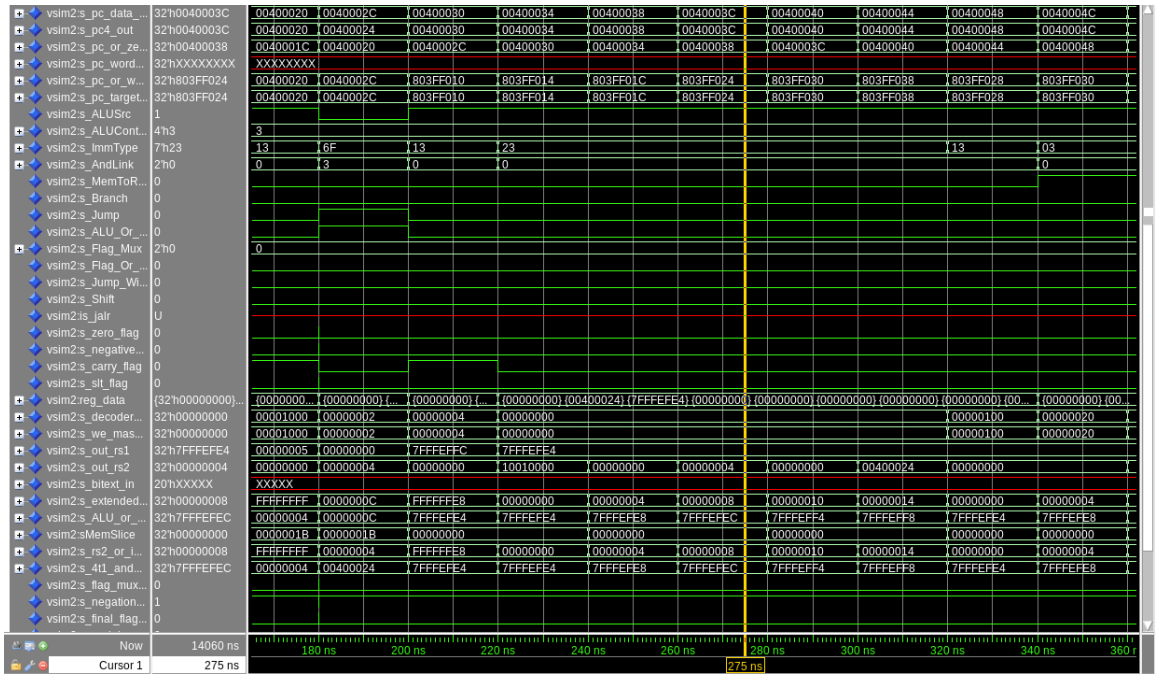
[Part 4.a] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.



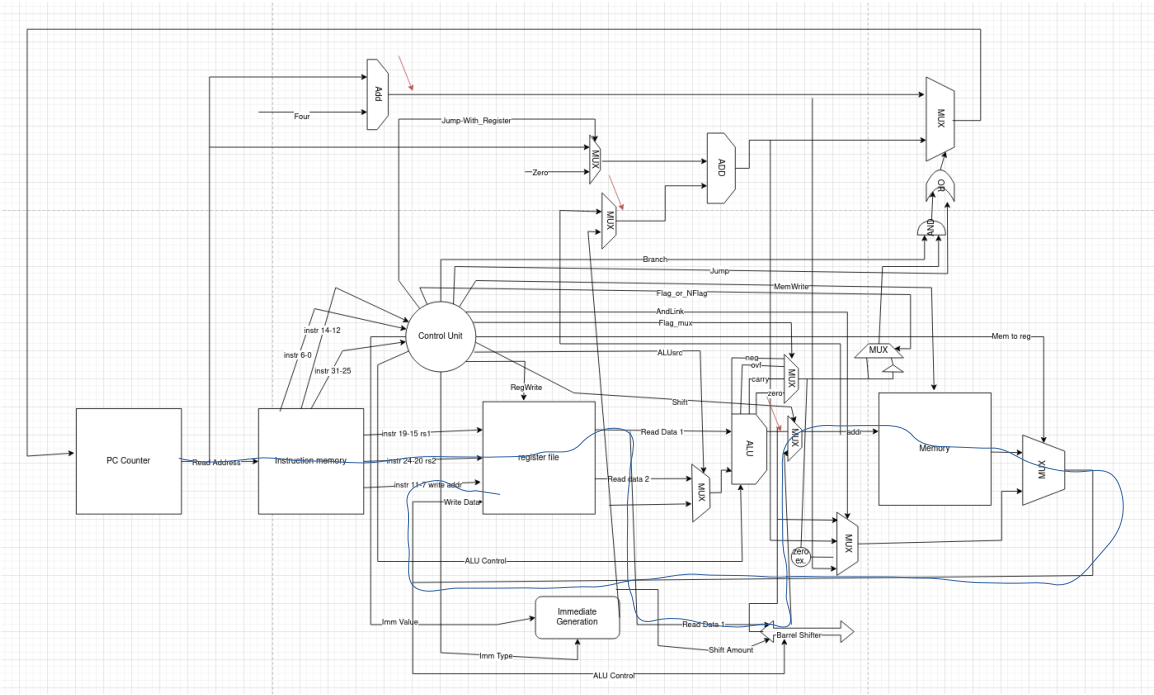
[Part 4.b] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.



[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?



The maximum frequency is 23.1 mhz.
The critical path is seen in blue above^^
We would focus on the barrel shifter to increase the frequency, it had the slowest portion of the critical path as it stands.