CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 1 Report

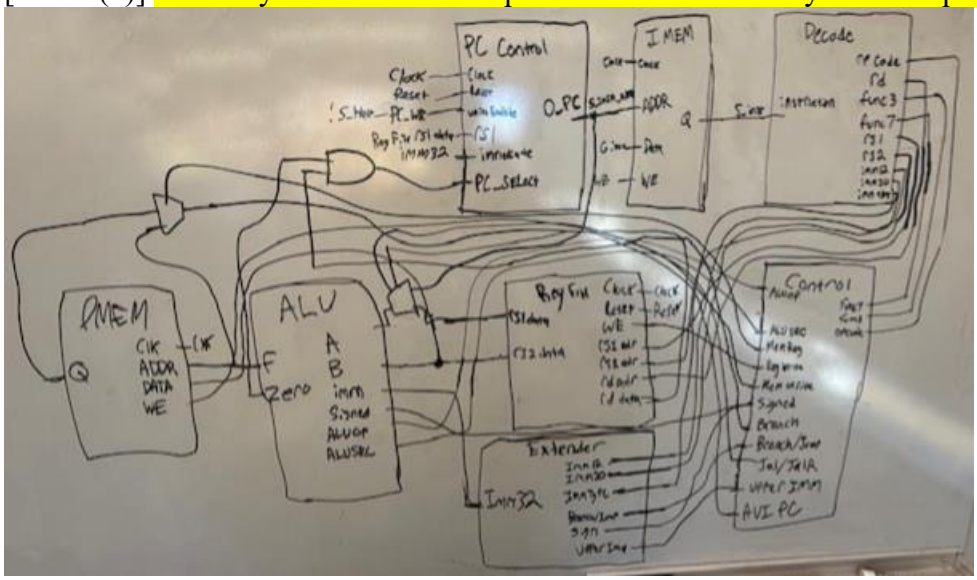
Team Members: **Connor Moroney**

Owen Gilbertson

Project Teams Group #: **D07**

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

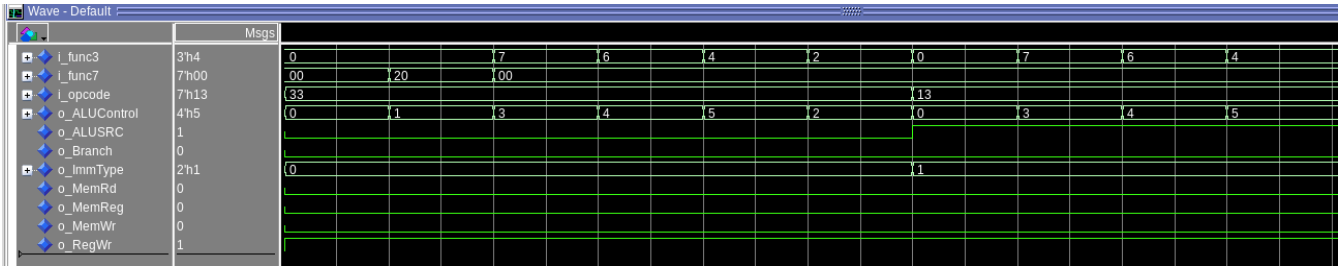
[Part 2 (d)] Include your final RISC-V processor schematic in your lab report.



[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.

Link to spreadsheet: Proj1_control_signals.xlsx

[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).



Test 1: Add: ALUSRC – 1, RegWr – 1, Else = 0 and is true
Test 2: Sub: ALUSRC – 1, RegWr – 1, ALUControl – 1, Else = 0 and is true
Test 3: And: ALUSRC – 1, RegWr – 1, ALUControl – 3, Else = 0 and is true
Test 4: Or: ALUSRC – 1, RegWr – 1, ALUControl – 4, Else = 0 and is true
Test 5: Xor: ALUSRC – 1, RegWr – 1, ALUControl – 5, Else = 0 and is true
Test 6: Slt: ALUSRC – 1, RegWr – 1, ALUControl – 2, Else = 0 and is true
Test 7: Addi: ALUSRC – 1, RegWr – 1, ImmType – 1, Else = 0 and is true
Test 8: Andi: ALUSRC – 1, RegWr – 1, ImmType – 1, ALUControl – 3, Else = 0 and is true
Test 9: Ori: ALUSRC – 1, RegWr – 1, ImmType – 1, ALUControl – 4, Else = 0 and is true
Test 10: Xori: ALUSRC – 1, RegWr – 1, ImmType – 1, ALUControl – 5, Else = 0 and is true

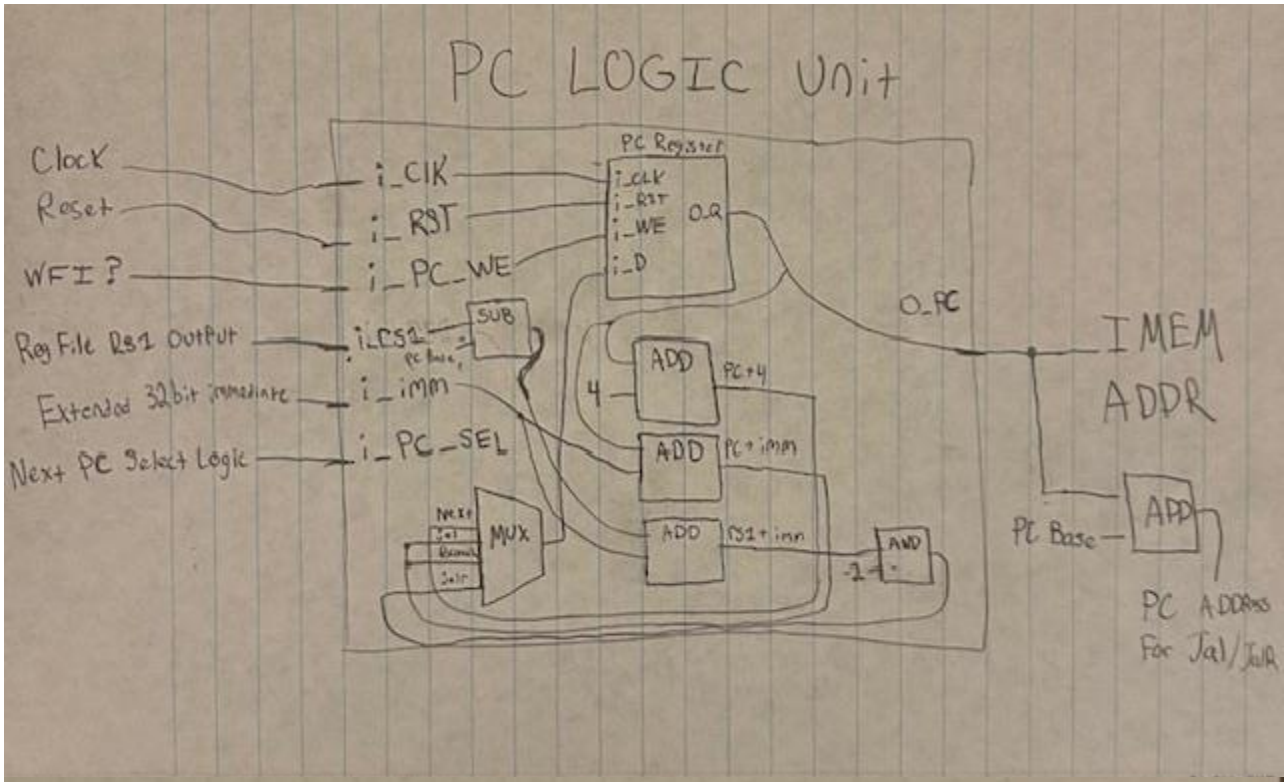
[Part 3.2. (a)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

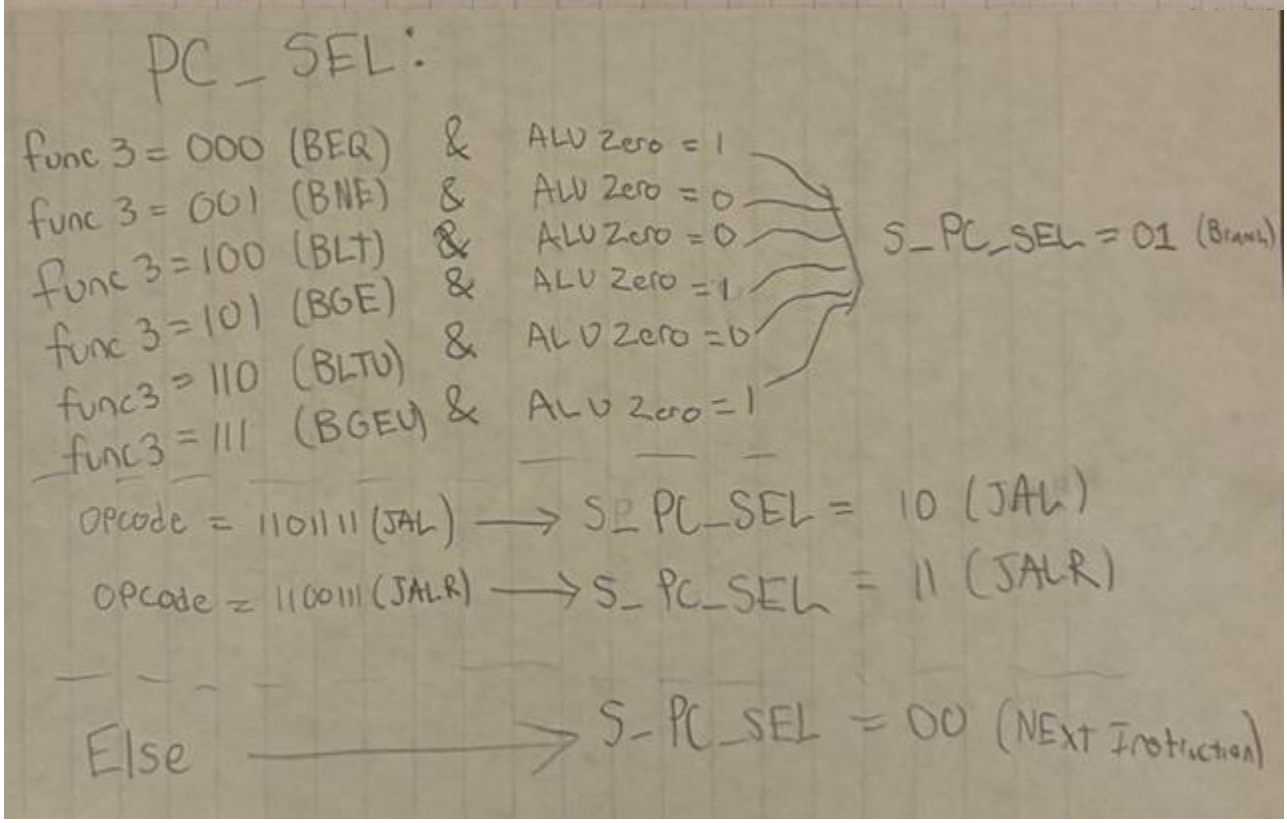
- **Unconditional Jumps (JAL/JALR)**
  Unconditional jumps set the PC to a specific address using an immediate (JAL) and/or a register value (JALR).
  pc = pc + signExtended(imm) (JAL)
  pc = reg[rs1] + signExtended(imm) (JALR)

- **Conditional Branches (BEQ/BLE, etc.)**
  Compares the values of two registers.
  If the condition is true, then pc = pc + signExtended(imm)
  If the condition is false, then pc = pc + 4

- **Default/Next Instruction**
  For all non-control flow instructions, pc = pc + 4.

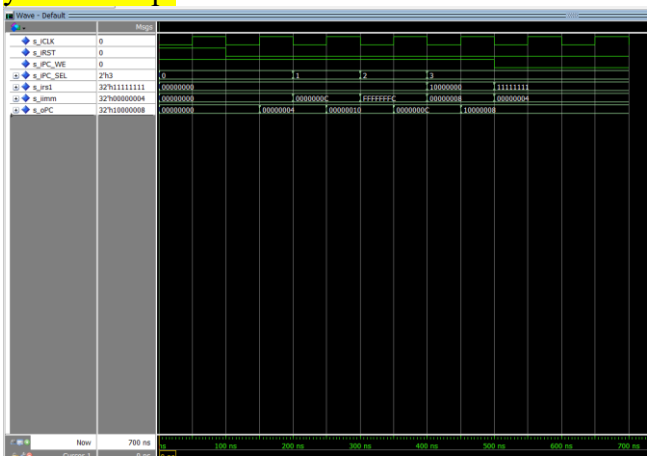
[Part 3.2. (b)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?





Additional control signals are needed for halting (WFI), PC next state selection (Outlined above), and Immediate type selection (because immediate for JALR needs 0 added to the end and not all immediates are 12-bits)

[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: Resetting the PC. s_oPC should be 0x00000000 and it is.
Test 2: Default case (PC + 4). s_oPC should be 0x00000004 and it is.
Test 3: Branch with a 10 immediate value. s_oPC should be 0x00000010 and it is.
Test 4: JAL with a -4 immediate value. s_oPC should be 0x0000000C and it is.
Test 5: JALR with an 8 immediate value and a 0x10000000 rs1 value. s_oPC should be 0x10000008 and it is.

[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 (Shift Right Logical) - fills necessary high-order bits with 0 when completing a shift.

SRA (Shift Right Arithmetic) - fills necessary high-order bits with the sign value (1 when negative, 0 when positive) when completing a shift

RISC-V does not include an SLA (Shift Left Arithmetic) because low-order bits do not affect the sign of a value.

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

Using the input signal "i_arith" the barrel shifter can distinguish a logical shift from an arithmetic shift. When "i_arith" is 1 (and "i_dir" is 0 for a right shift), any bits that get shifted from out of bounds into the output data will be set to the value of the MSB (1 if negative, 0 if positive). When "i_arith" is 0 (or "i_dir" is 1 for a left shift), all bits that get shifted from out of bounds into the output data will be set to 0.

[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 updated the barrel shifter to include an input signal "i_dir" that can control which direction the shifter shifts the bits. When "i_dir" is 0, a right shift occurs (data(i) = data(i+N)). When "i_dir" is 1, a left shift occurs (data(i) = data(i-N)).

[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 (SLL). (Shift 0x00000001 by 1 to the left) s_odata should be 0x00000002 and it is
Test 2: Logical Right Shift (SRL). (Shift 0x80000000 by 1 to the right) s_odata should be 0x40000000 and it is
Test 3: Arithmetic Right Shift (SRA). (Shift 0x80000000 by 1 to the right) s_odata should be 0xC0000000 (0x40000000 + 0x80000000) and it is**.**
Test 4: Shift by 0. (Shift 0xDEADBEEF by 0 to the right) s_odata should be 0xDEADBEEF and it is
Test 5: Arithmetic Left Shift (SLA). SHOULDN'T WORK (SHOULD DEFAULT TO SLL) (Shift 0x8000000F by 4 to the left) s_odata should be 0x000000F0 and it is

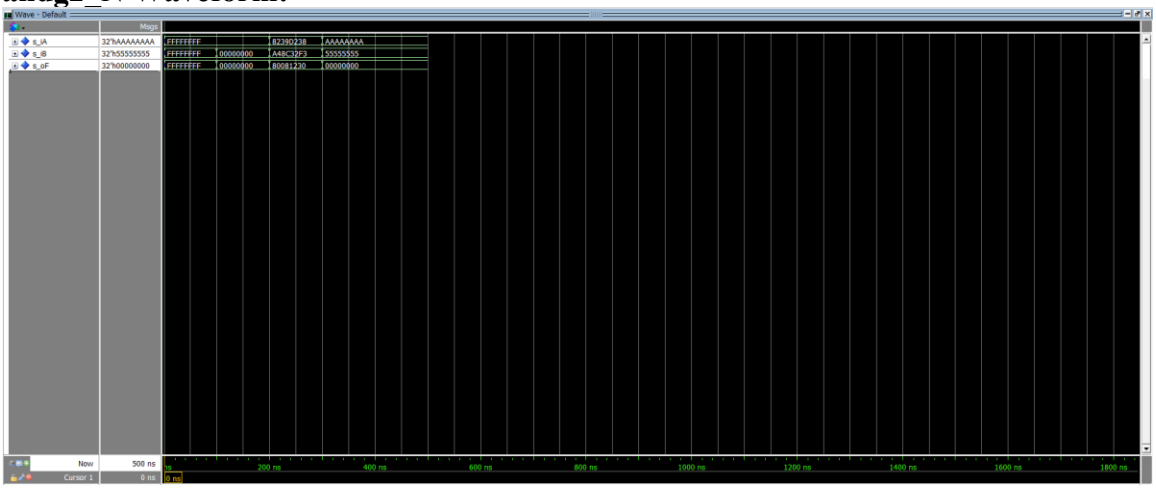
[Part 3.3.2.(a)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

After implementing the barrel shifter, we needed to create components for the AND/OR/XOR operations. To complete this, we simply created N-bit versions of the gates that were already provided to us (i.e. for andg2 we created andg2_N using structural VHDL that used the andg2 for each bit of the input. Then, when the ALU is performing an AND operation, the i_ALUOp would select the output of the AND gate to be used as the output of the ALU). We followed the same process for org2, xorg2, and invg.

The more interesting component we had to create was a component for SLT. Technically speaking, we could have used the add component to implement SLT, but we thought that approach would be a little more confusing. So, we created an slt_N.vhd file that could carry out an SLT operation. Then, using the i_ALUOp, we were able to select the output of the slt when the alu is performing an slt operation.

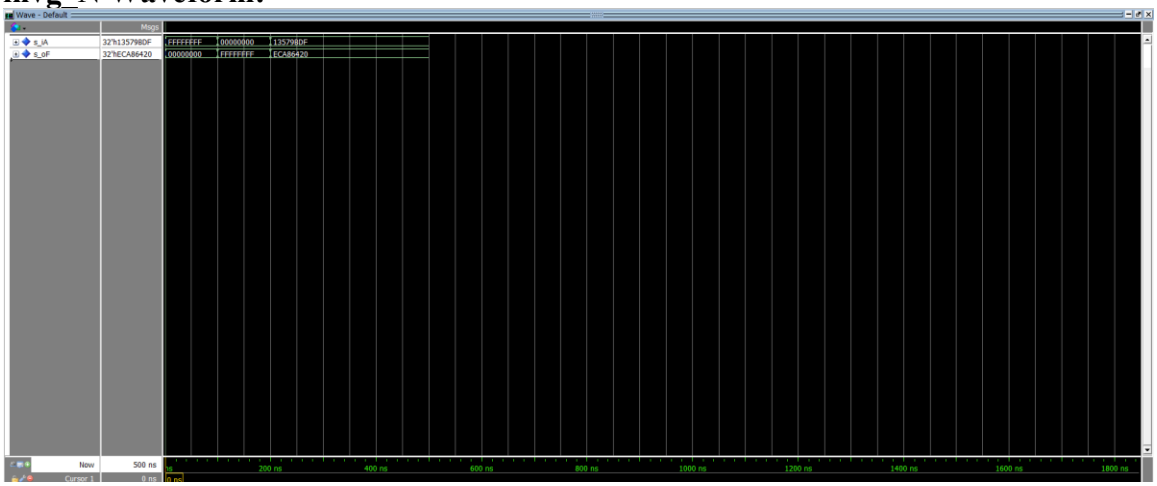
[Part 3.3.2.(b)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

andg2_N Waveform:



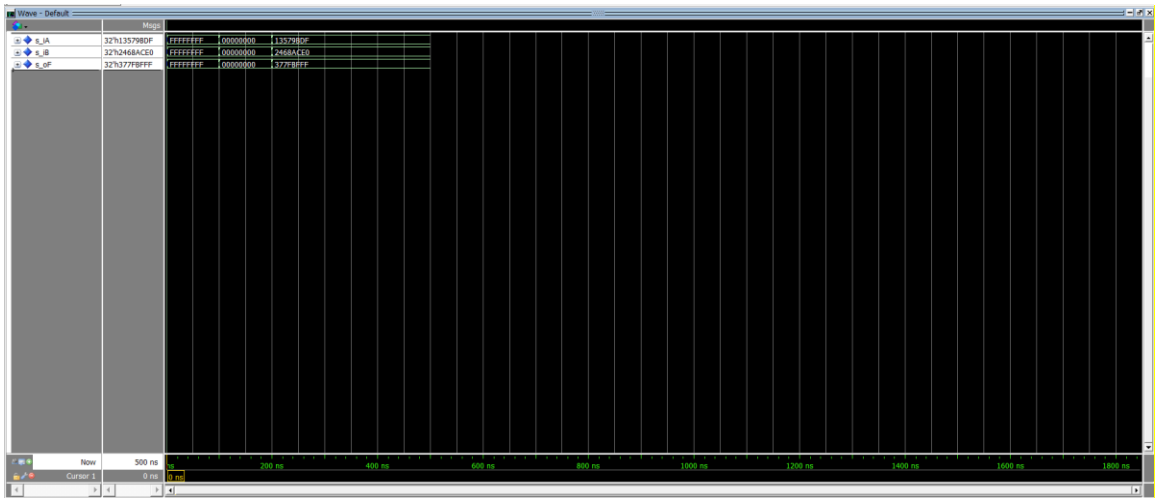
Test 1: 0xFFFFFFFF AND 0xFFFFFFFF. Expect o_F to be 0xFFFFFFFF and it is.
Test 2: 0x00000000 AND 0x00000000. Expect o_F to be 0x00000000 and it is.
Test 3: 0x8239D238 AND 0xA4BC32F3. Expect o_F to be 0x80081230 and it is.
Test 4: 0xAAAAAAAA AND 0x55555555. Expect o_F to be 0x00000000 and it is.

invg_N Waveform:



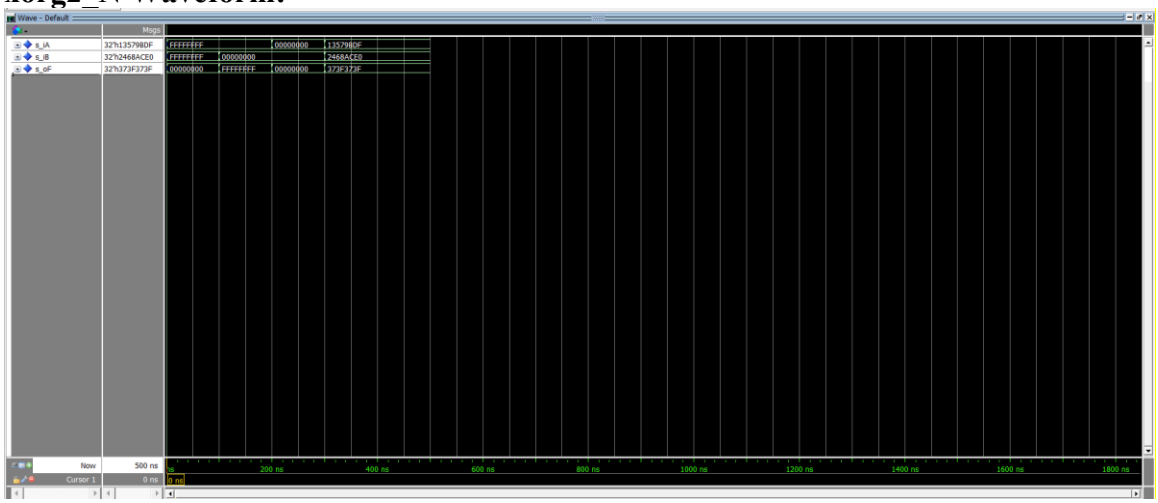
Test 1: NOT 0xFFFFFFFF. Expect o_F to be 0x00000000 and it is.
Test 2: NOT 0x00000000. Expect o_F to be 0xFFFFFFFF and it is.
Test 3: NOT 0x13579BDF. Expect o_F to be 0xECA86420 and it is.

org2_N Waveform:



Test 1: 0xFFFFFFFF OR 0xFFFFFFFF. Expect o_F to be 0xFFFFFFFF and it is.
Test 2: 0x00000000 OR 0x00000000. Expect o_F to be 0x00000000 and it is.
Test 3: 0x13579BDF OR 0x2468ACE0. Expect o_F to be 0x377FBFFF and it is.

xorg2_N Waveform:

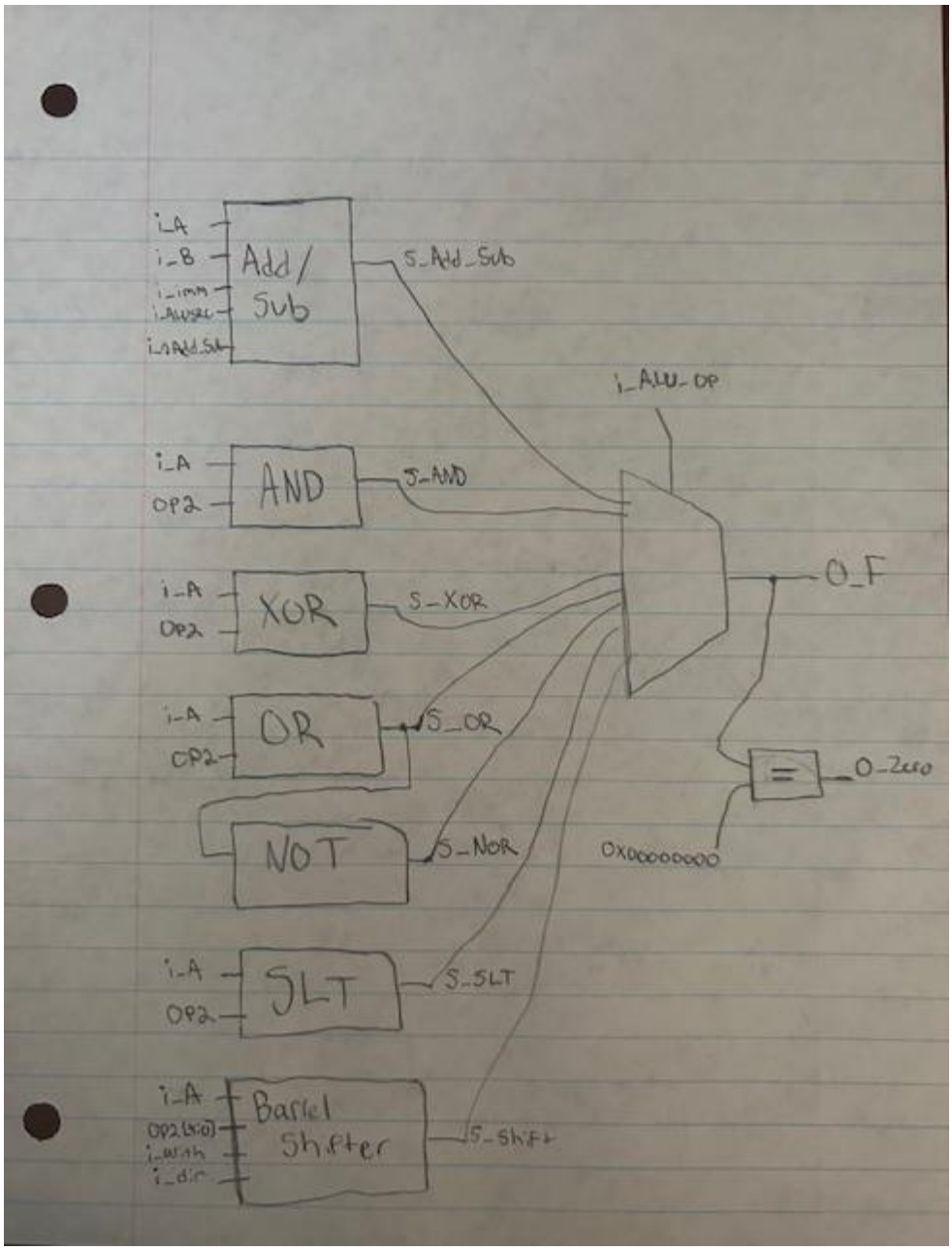


Test 1: 0xFFFFFFFF XOR 0xFFFFFFFF. Expect o_F to be 0x00000000 and it is.
Test 2: 0x00000000 XOR 0xFFFFFFFF. Expect o_F to be 0xFFFFFFFF and it is.
Test 3: 0x00000000 XOR 0x00000000. Expect o_F to be 0x00000000 and it is.
Test 4: 0x13579BDF XOR 0x2468ACE0. Expect o_F to be 0x373F373F and it is.

slt_N Waveform:

Test 1: 0xFFFFFFFF SLT 0xFFFFFFFF. Expect o_F to be 0x00000000 and it is.
Test 2: 0xFFFFFFFF SLT 0x00000000. Expect o_F to be 0x00000001 and it is.
Test 3: 0x00000000 SLT 0xFFFFFFFF. Expect o_F to be 0x00000000 and it is.
Test 4: 0x13579BDF SLT 0x2468ACE0. Expect o_F to be 0x00000001 and it is.

[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?



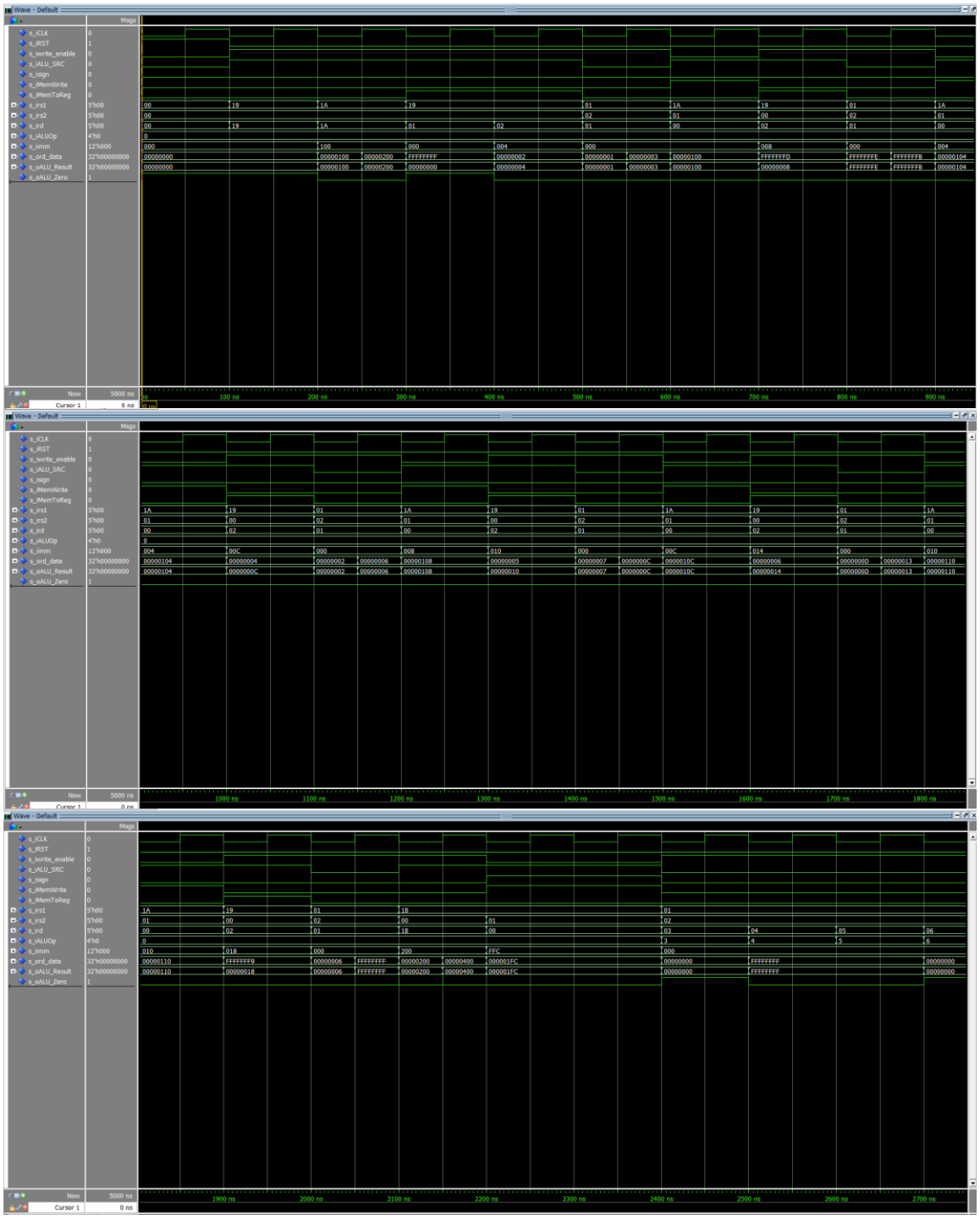
o_Zero is 0 when o_F does not equal 0x00000000
o_Zero is 1 when o_F equals 0x00000000

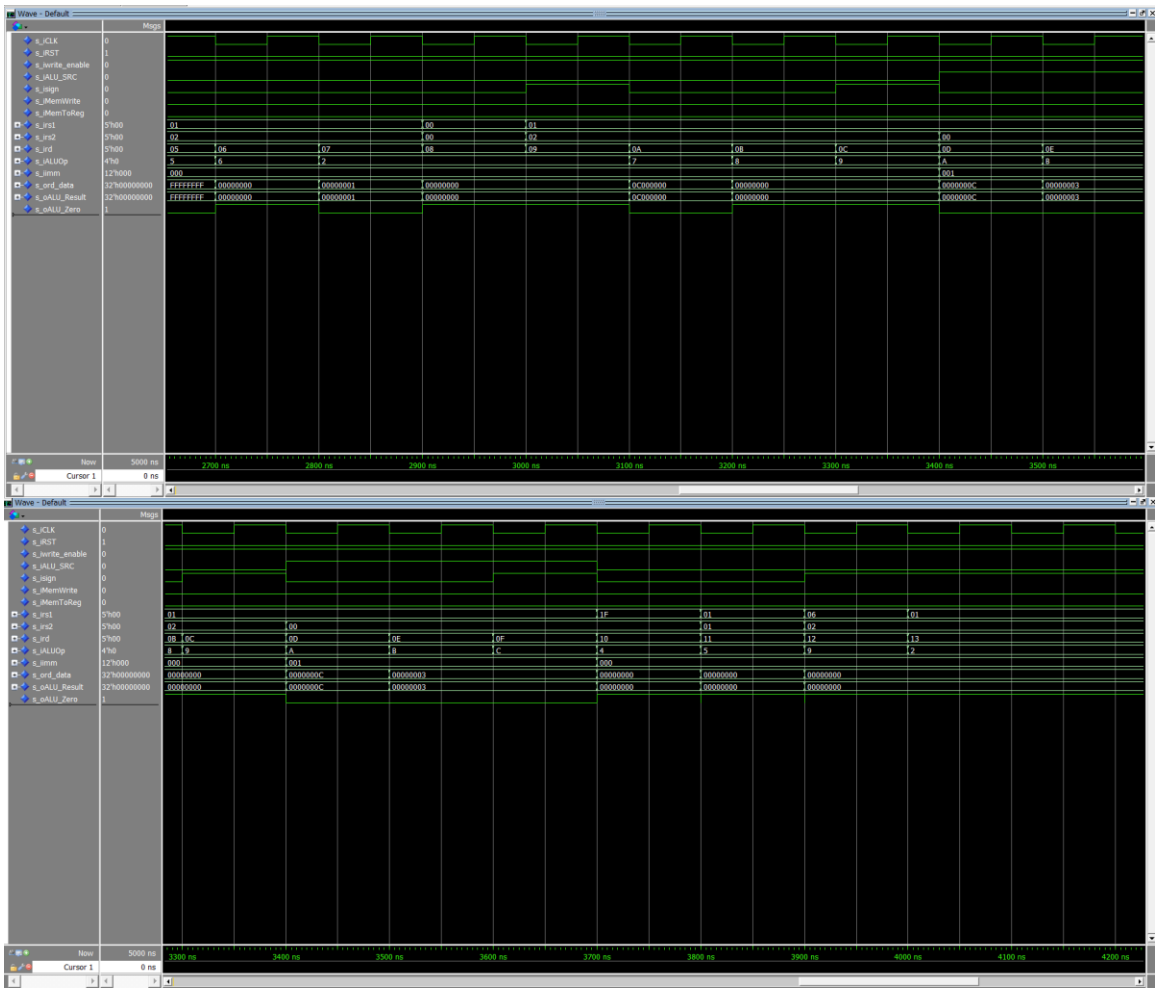
SLT is implemented in our case by a separate SLT unit. The output is then selected by i_ALU_Op if it is supposed to be an SLT operation.

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

Test 1: 0x00001234 + 0x00005678. Expect o_F = 0x000068AC, o_Zero = '0' and it is.
Test 2: 0x0000ABCD - 0x00004567. Expect o_F = 0x00006666, o_Zero = '0' and it is.
Test 3: 0x00001000 + 0x00000010 (ADDI). Expect o_F = 0x00001010, o_Zero = '0' and it is.
Test 4: 0x0000AAAA - 0x0000000F (SUBI). Expect o_F = 0x0000AA9B, o_Zero = '0' and it is.
Test 5: 0xFFFFFFFE < 0x00000005 (SLT). Expect o_F = 0x00000001, o_Zero = '0' and it is.
Test 6: 0x1234ABCD AND 0x0F0F0F0F. Expect o_F = 0x02040B0D, o_Zero = '0' and it is.
Test 7: 0x12340000 OR 0x0000ABCD. Expect o_F = 0x1234ABCD, o_Zero = '0' and it is.
Test 8: 0x55AA55AA XOR 0x0F0F0F0F. Expect o_F = 0x5AA55AA5, o_Zero = '0' and it is.
Test 9: 0x12345678 NOR 0x87654321. Expect o_F = 0x688AA886, o_Zero = '0' and it is.
Test 10: 0x00000011 << 4 (SLL). Expect o_F = 0x00000110, o_Zero = '0' and it is.
Test 11: 0xF0000000 >> 4 (SRL). Expect o_F = 0x0F000000, o_Zero = '0' and it is.
Test 12: 0x80000000 >> 2 (SRA). Expect o_F = 0xE0000000, o_Zero = '0' and it is.
Test 13: 0x00001234 << 3 (SLLI). Expect o_F = 0x000091A0, o_Zero = '0' and it is.
Test 14: 0x80001234 >> 3 (SRLI). Expect o_F = 0x10000246, o_Zero = '0' and it is.
Test 15: 0xF0001234 >> 3 (SRAI). Expect o_F = 0xFE000246, o_Zero = '0' and it is.
Test 16: 0x00000001 + 0xFFFFFFFF. Expect o_F = 0x00000000, o_Zero = '1', and it is.
Test 17: 0xABCDEF01 - 0xABCDEF01. Expect o_F = 0x00000000, o_Zero = '1', and it is.
Test 18: 0x00000005 < 0x00000003 (SLT). Expect o_F = 0x00000000, o_Zero = '1', and it is.
Test 19: 0xFFFF0000 AND 0x0000FFFF. Expect o_F = 0x00000000, o_Zero = '1', and it is.

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


Wave - Default
Msgs
s_iCLK
s_iRST
s_iwrite_enable
s_iALU_SRC
s_isign
s_iMemWrite
s_iMemToReg
s_irs1
s_irs2
s_ird
s_iALUOp
s_iimm
s_ord_data
s_oALU_Result
s_oALU_Zero
Now
Cursor 1
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1000 ns
1100 ns
1200 ns
1300 ns
1400 ns
1500 ns
1600 ns
1700 ns
1800 ns
1900 ns
2000 ns
2100 ns
2200 ns
2300 ns
2400 ns
2500 ns
2600 ns
2700 ns

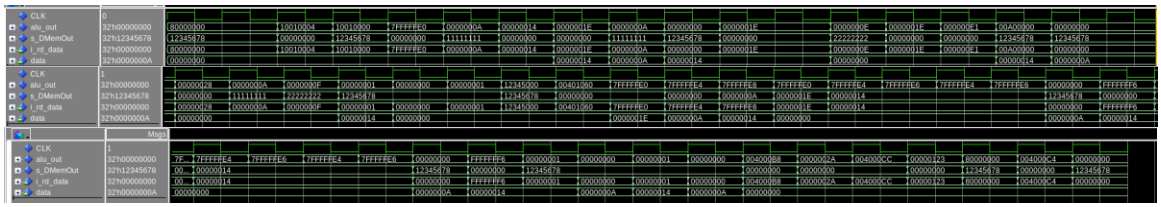


We had 40 tests cases in order to make sure that the Datapath worked with the updated ALU. We made sure to test edge cases and found that the Datapath output worked exactly the same as the previous version, except now it has access to more operations (SLT, AND, OR, SLL, etc.)

[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.

Done (see below)

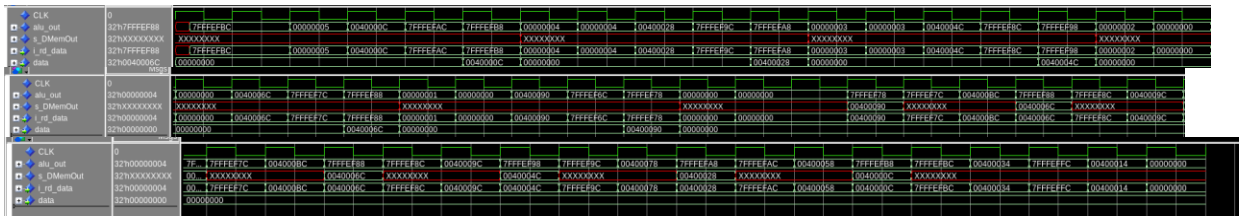
[Part 4.a] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially,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



alu_out = Output from ALU
s_DMemOut = Output from DMEM
i_rd_data = Data being written to register
data = Data being written to Memory

The base tests works as expected. The outputs matched the expected outputs in RARS.

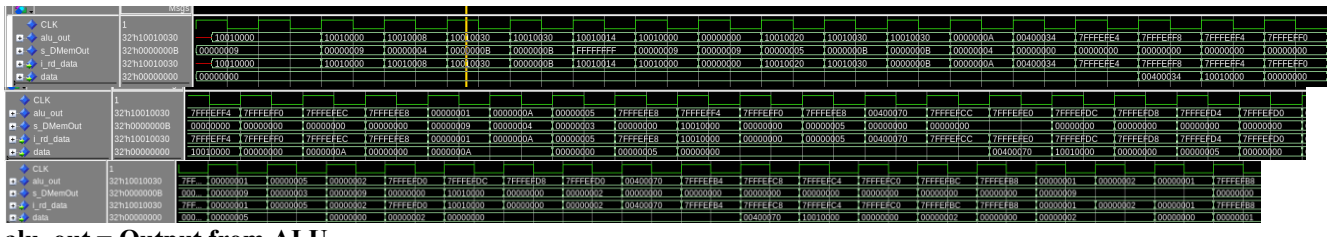
[Part 4.b] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.



alu_out = Output from ALU
s_DMemOut = Output from DMEM
i_rd_data = Data being written to register
data = Data being written to Memory

The Control Flow test works as expected. The outputs matched the expected outputs in RARS.

[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.

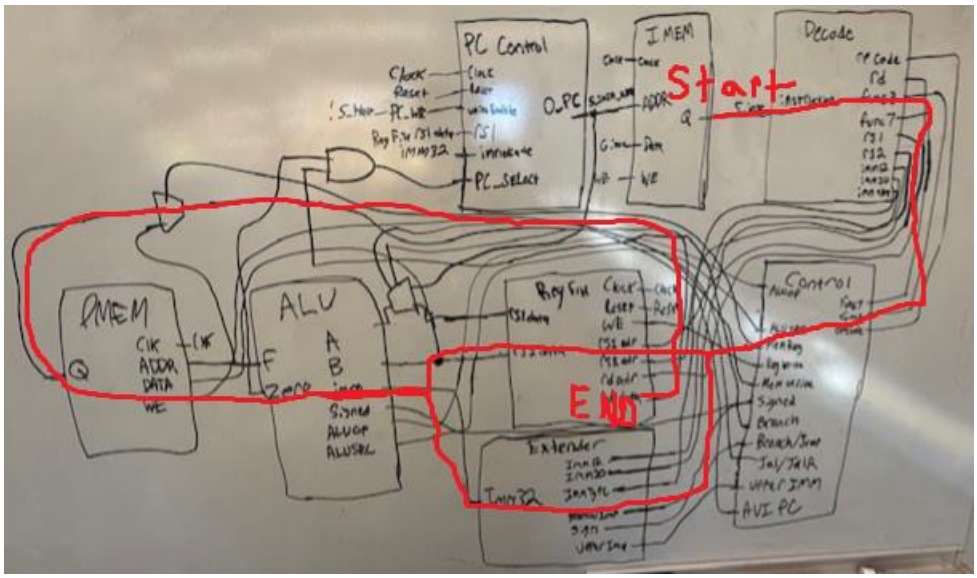


alu_out = Output from ALU
s_DMemOut = Output from DMEM
i_rd_data = Data being written to register
data = Data being written to Memory

The Merge Sort test works as expected. The outputs matched the expected outputs in RARS.

[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?

Fmax = 25.49 MHz



We noticed that the DMEM unit seemed to take a long time to complete. That might be something we aim to improve in the future.