CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: ____Eli von Nordheim_____

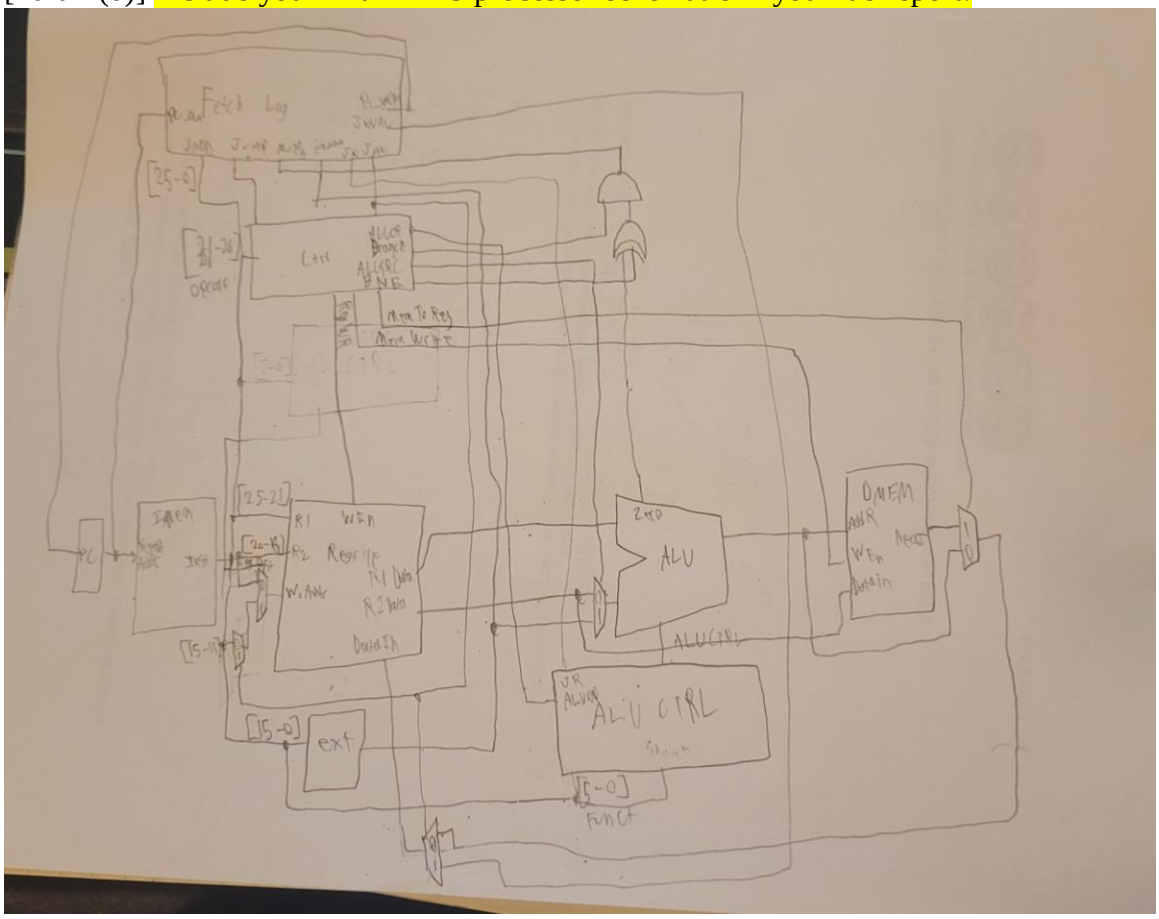
____Sam Forde_____

_____

Project Teams Group #:__Sec B 02_____

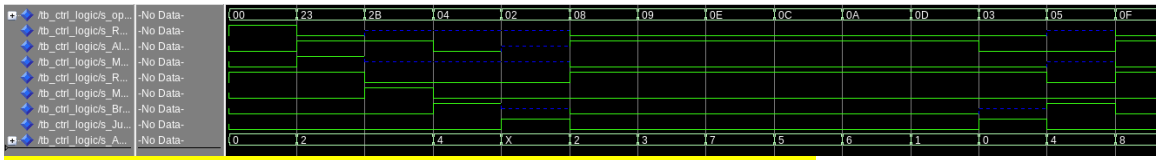
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[Part 2 (a.i)] 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

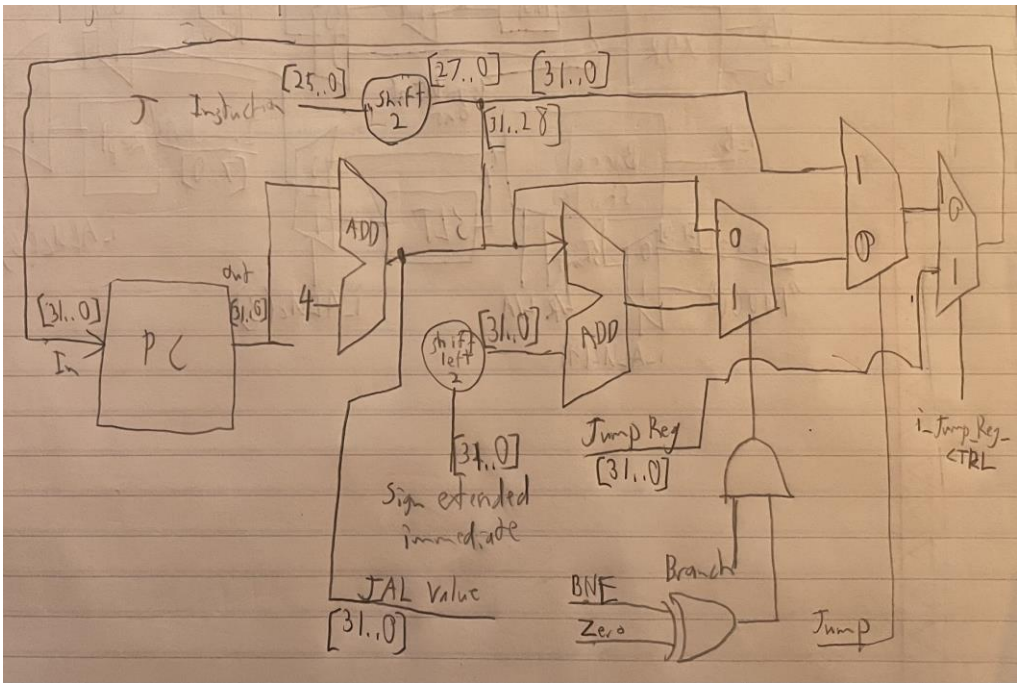
separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your datapath implementation. The end result should be an *N***M* table where each row corresponds to the output of the control logic module for a given instruction.
Spreadsheet in docs folder
[Part 2 (a.ii)] 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).

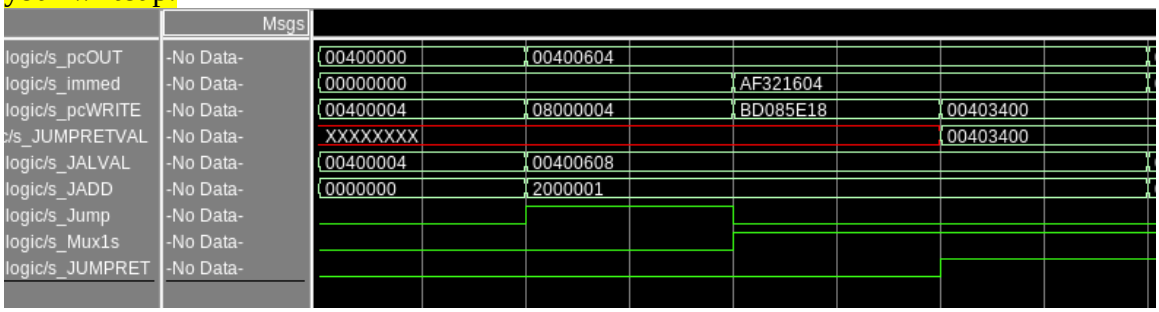
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.
The fetch logic must be able to support standard incrementing by 4, jump and jal instructions, branch instructions, and jump register.

[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?
Signals needed are JAL, Jump, a bit that knows if the branch instruction is bne or beq, and the zero detect line.



[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use Modelsim 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 Modelsim waveforms in your writeup.



This simulation looks at the different functions of the fetch logic and tests each of them. The first 10 ns test the standard increment. The next 10 tests a jump, where the output is the shifted and augmented input from the j type instruction. The next part tests the branch path, where the immediate value is added to the current instruction counter. The final part is the Jump Register, where a the output from the registers is multiplexed into the fetch logic.

[Part 2 (c.i.1)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?

srl shifts bits to the right filling in 0s as bits are shifted out, sra shifts to the right but fills in the sign bit. There is no sla because sla can be accomplished by adding a number to itself per shift.

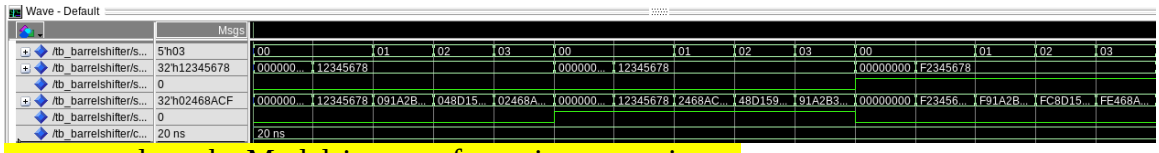
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

The code implements right shifting using a barrel shifter where the 0s are instead connected to a mux that can select between 0 and the sign bit. Left shifting is implemented by inverting both the input and the output of the barrel shifter.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

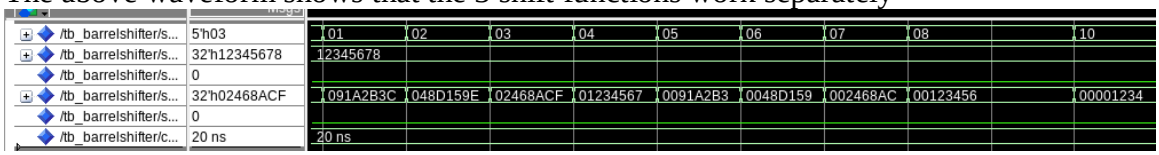
If both the input and output are inverted, the barrel shifter will shift left instead of right

[Part 2 (c.i.4)] Describe how the execution of the different shifting operations



corresponds to the Modelsim waveforms in your writeup.

The above waveform shows that the 3 shift functions work separately



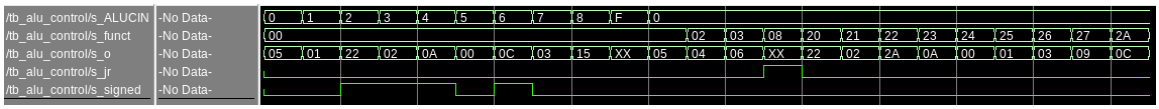
The above waveform shows that all of the control bits increase the shift amount by the correct amount.

[Part 2 (c.ii.1)] 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.

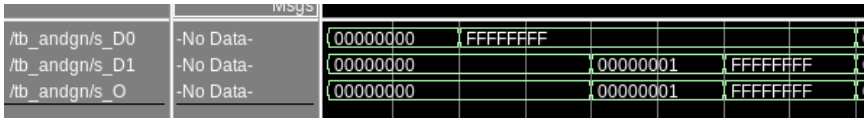
In order to get the ALU up to specification, I added an AND gate array(made out of single bit AND gates), a OR gate array(made out of single bit OR gates), a XOR gate array(made out of single bit xor gates), a SLT unit(a simple dataflow assignment of zeroes and the sign bit of the adder), and a zero detect functional unit(dataflow unit that checks if the ALU output is zero or not). I also built an ALU control unit that takes signals from the control logic and converts them into specific signals for the ALU to use.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

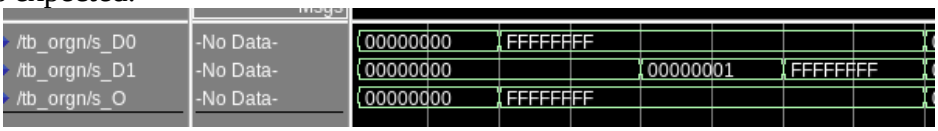
The ALU control takes the simplified signal from the control logic, and decodes those signals into specific bits to set for the ALU, using funct for the R-type instructions. This testbench cycles through all used configurations that the ALU control unit is designed to receive and send to the ALU.



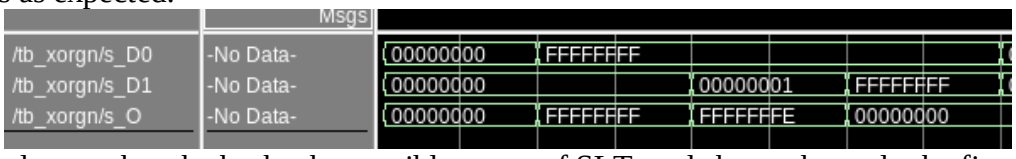
The below testbench checks the N-bit AND gate, showing that the AND is bit wise and works as expected.



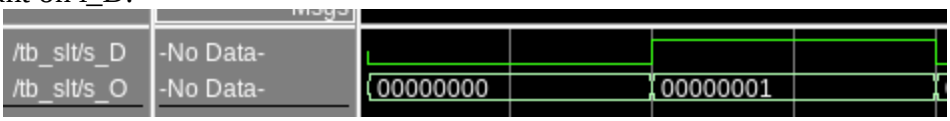
The below testbench checks the N-bit OR gate, showing that the OR is bit wise and works as expected.



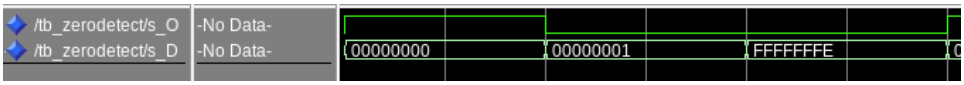
The below testbench checks the N-bit XOR gate, showing that the XOR is bit wise and works as expected.



The below testbench checks the possible states of SLT, and shows that only the first bit is dependant on i_D.

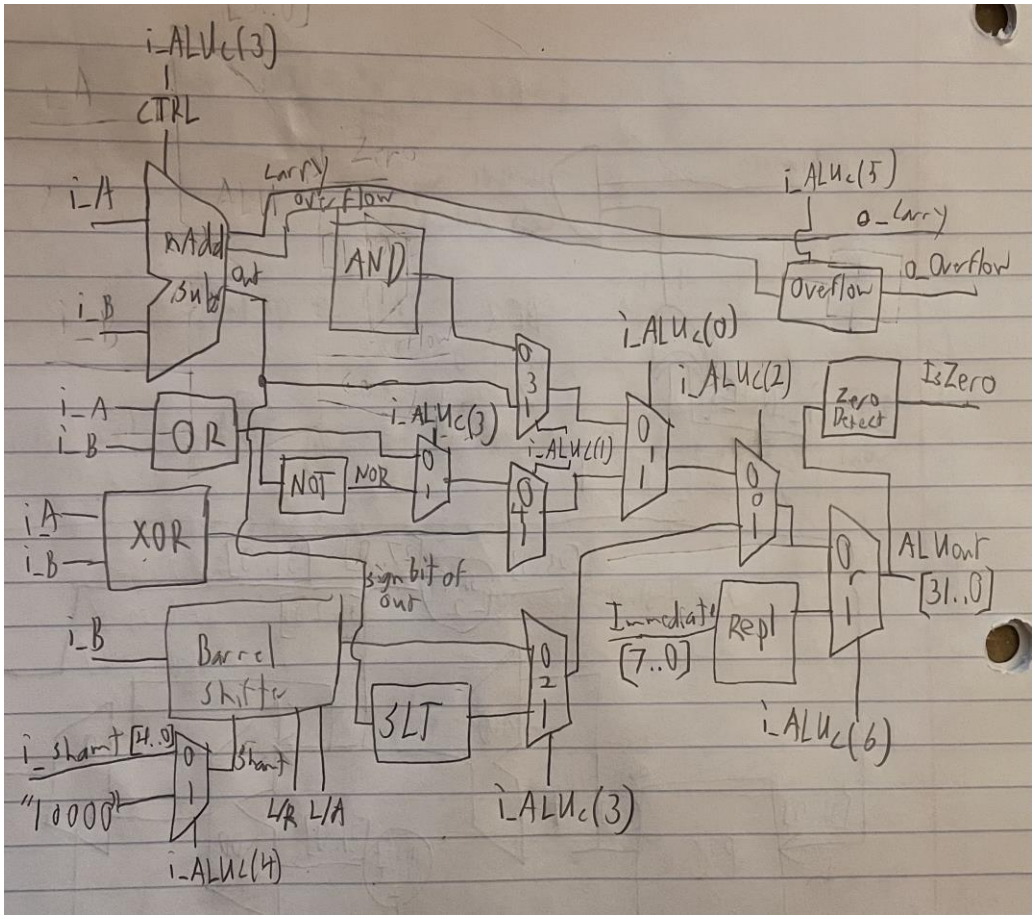


The below testbench checks the zero detect unit



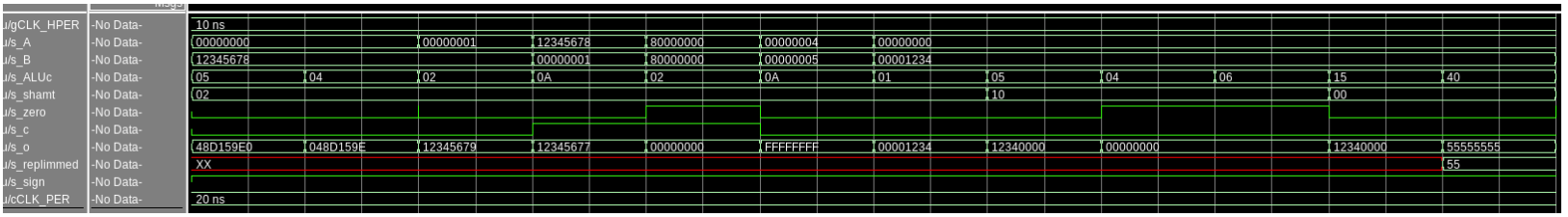
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?

Overflow is a not of the last bit of i_A and a not of the last bit of i_B ANDed with the last bit of the calculated output. This part is then ORed with i_A and i_B and a NOTed output to check if the sign bit of the output differs from the expected output sign. Zero is calculated by seeing if all of the bits of the output of the ALU are zero. SLT simply fills the first bit of the output with the sign bit from the ALU and fills the rest of the bits with



zero.

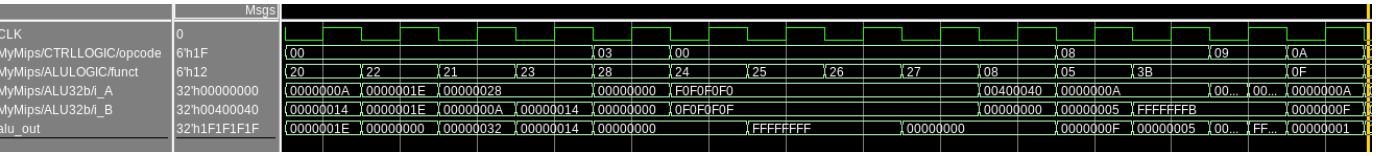
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



This testbench checks various possible states of the ALU using spoofed control signals from the ALU control unit, and sample inputs to i_A, i_B, and shamt just to verify the system outputs properly, since all of the individual units have already been tested. The ALUc input 0x5 is a test of sll, where the output is shifted left by two. ALUc input 0x4 is srl is a right shift by the same amount. 0x2 is the add function, and 0xA is subtraction. The next two 0x2 and 0xA's test that the adder has been hooked up correctly, and the zero detect line. After that comes 0x1, which is the or function. After that comes another test of sll, used for comparison to the lui function's shifting. Then another shift right logical to check another case with zero detect, and 0x6 to check sra. Then comes 0x15, which is the load upper immediate test. Finally 0x40 is a test of repl.qb.

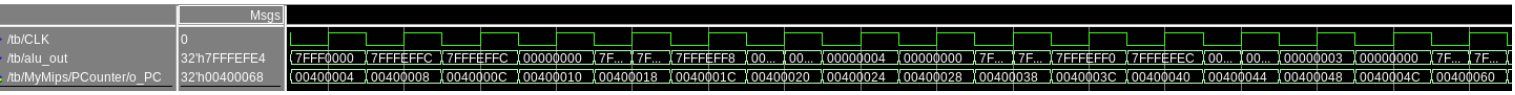
[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.
Instead of spending time on building an extra test bed, we opted to integrate the ALU into the MIPS processor design, and test with MIPS assembly.

[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.
Whatever you say boss
[Part 3 (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.



Add Sub Addu Subu Jal And Or Xor Nor Jr Addi Addi Addiu Slti
While I ran more instructions in this program than just these, this is most of the standard arithmetic/logical. This passed the MARS comparison, and if one calculates out each equation, they will find it all passes inspection.

[Part 3 (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.



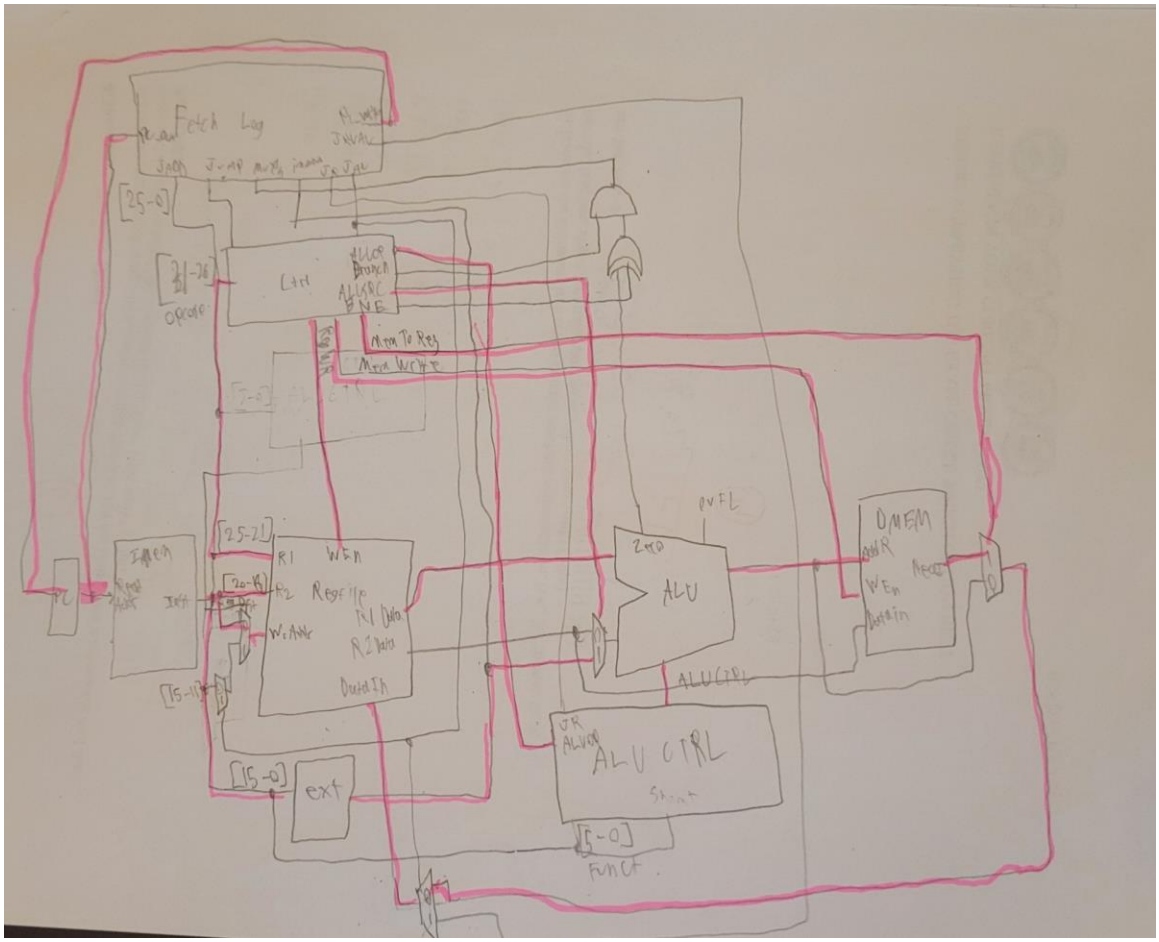
Here is a segment of the 54 instruction control-flow tests. It passed MARS comparison.

[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

The bubble sort algorithm also passed MARS comparison with no issues.

[Part 4] 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 schematics. What components would you focus on to improve the frequency?

The processor's maximum frequency is 23.25 MHz, with a critical path going from the program counter to the imem, which is then decoded by the ctrl logic and ALU logic, through the ALU's adder and into DMEM which then outputs data back to the register file. This is assumed, as the critical path in timing.txt is a load word instruction that goes through the barrel shifter rather than the adder, which is not implemented in any of our



instructions, with the greatest time sink being the DMEM.