CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: ____Clay Kramper_____

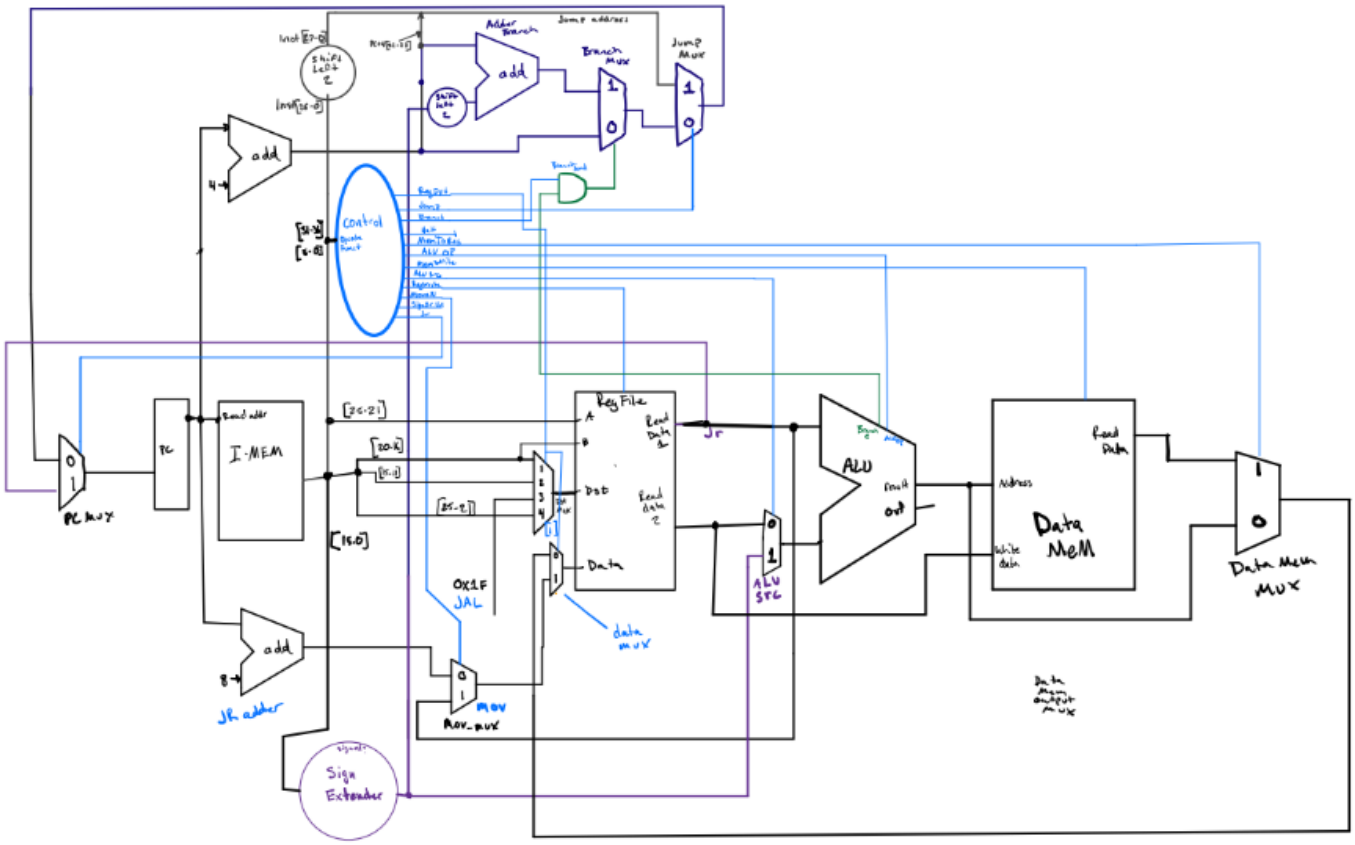
____Blake Carlson_____

____Ben Johnson_____

Project Teams Group #:____4-3_____

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

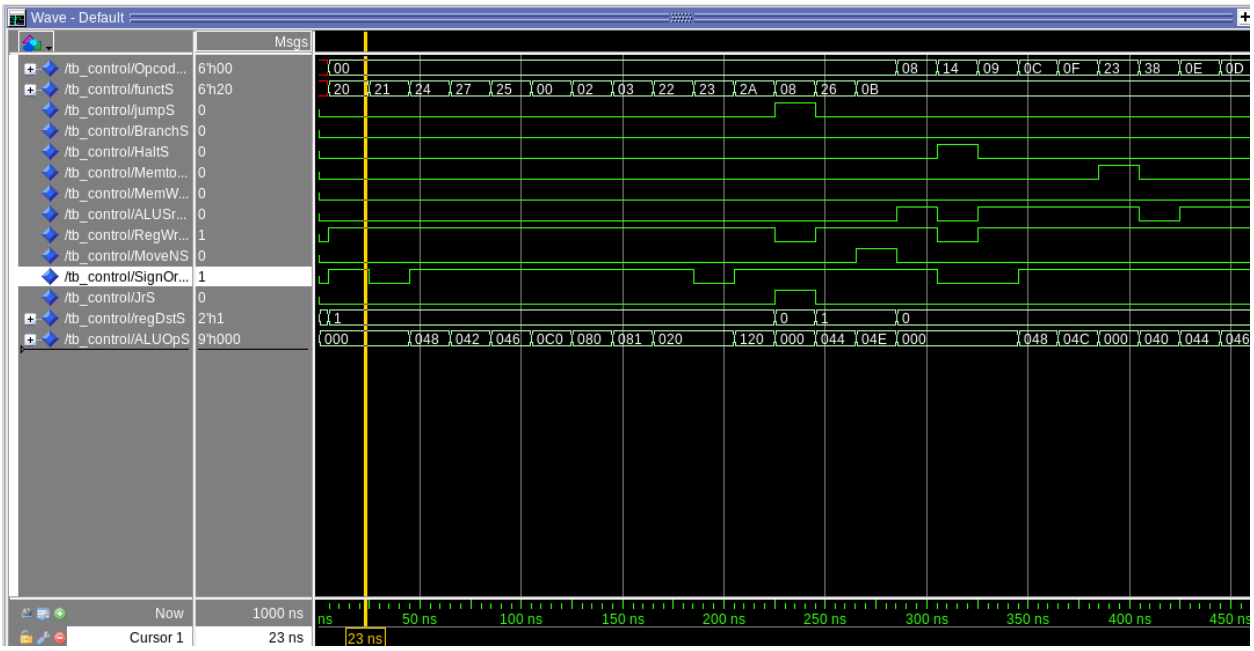
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



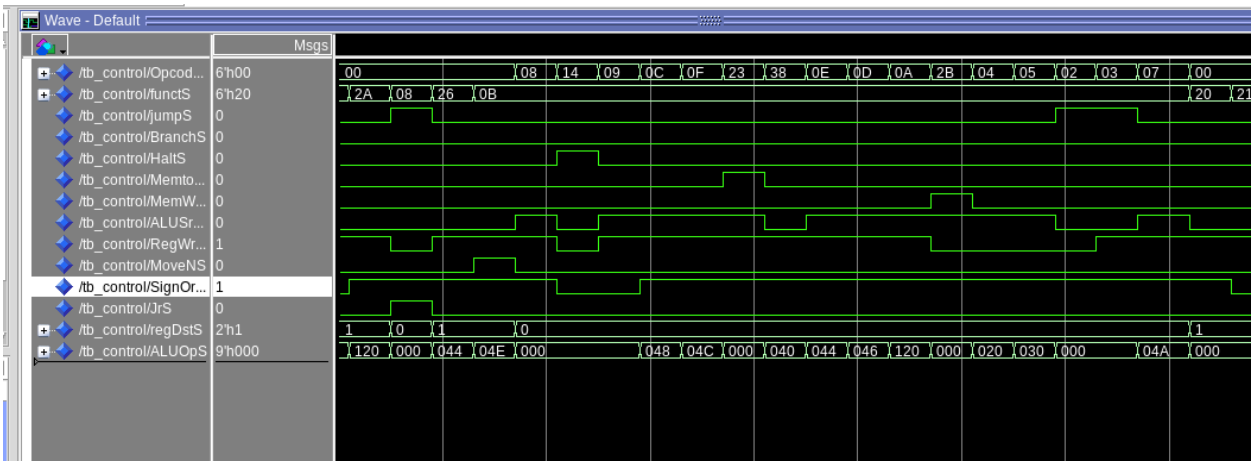
[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your 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.

Included in zip file

[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).



From this testbench it is able to show that the outputs that were expected for when the opcode is 000000. As we can see from the testbench results we are getting the values we should expect from the control unit.



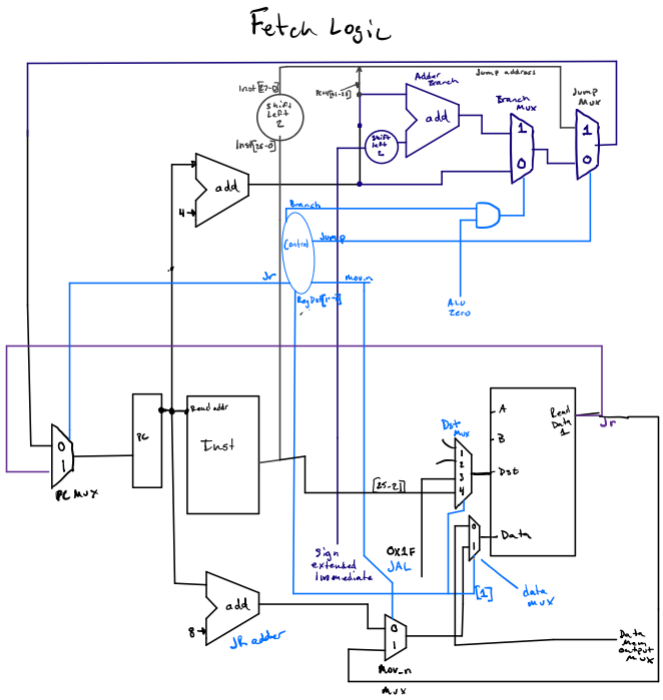
This part of the testbench shows the outputs for when the opcode changes for using all of the I-type instructions. As we can see from the test bench all of the expected outputs for I-type instructions match.

[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.

Fetch logic must support sequential instructions via a PC = PC + 4 circuit, also must support branching instructions with an adder that adds the immediate input * 4 to the next PC value. Finally we must support Jump instructions with jump address parsed

from Instruction memory * 4 to compute a new jump address and wired backing the PC input.

[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?



[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.

Fetch logic is demonstrated via the execution of MIPS code. This shows a test bench for simplebranch.s. The processor still needs some debugging but the fetch logic is shown to be successful via branch not equal output setting when the branch address route needs to be taken.

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

SRL is a logical shift, meaning that it will shift the bits by the specified amount, and put zero's in where they were shifted.

SRA is an arithmetic shift, meaning that it will shift the bits by the specified amount, and then it will put the sign bit (MSB) in the space that opened up.

There is no SLA instruction in mips because the shift left logical and shift left arithmetic do the same thing. Therefore there is no need for an SLA instruction in MIPS.

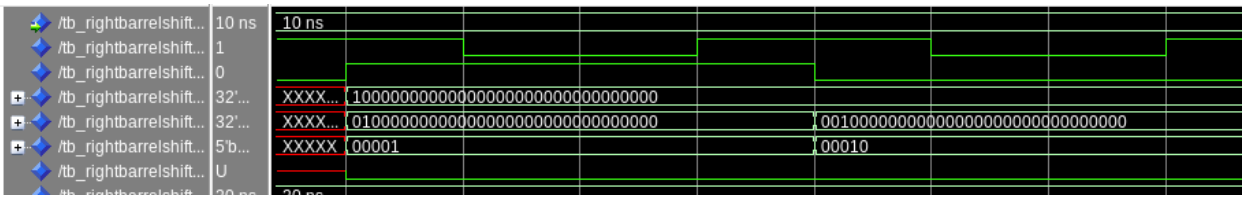
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

We do not need to handle both in shift left because SLA and SLL are the same. For SRL and SRA, we take an input bit into the barrel shifter to specifiy if we are replacing the shifted values with either a zero or with the sign bit from the input.

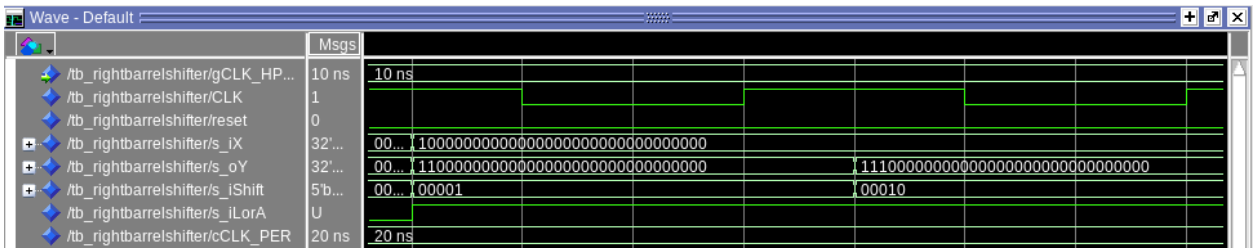
[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

The right barrel shifter could be enhanced to handle left shifting operations with 2 changes. The first would be to add an input bit to specify if we are shifting left or right. Depending on this bit, we would then either shift left by adding zeros to the end, or shift right by adding them to the beginning of the input.

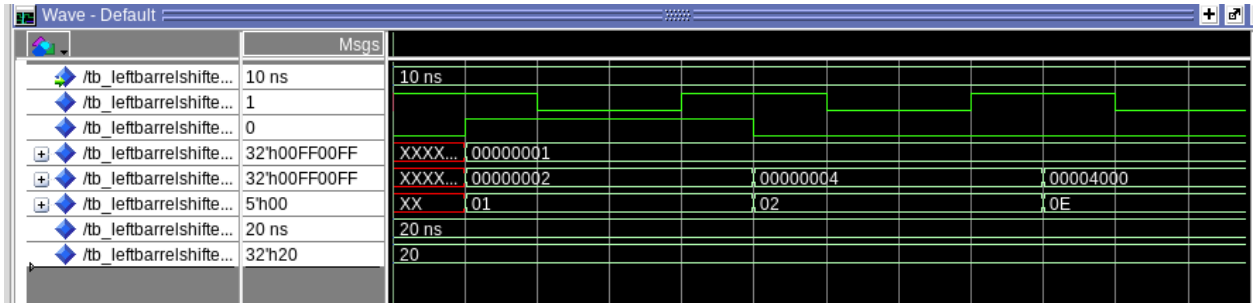
[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the Modelsim waveforms in your writeup.



The above screen shot shows tests for SRA by 1 and SRAby 2. Both outputs show the expected result. Additonal test cases can be found in the testbench. (22 total for right shifts)



The above screenshot shows tests for SRL by 1 and SRL by 2. Both outputs show the expected result. Further test cases can be found in the testbench.

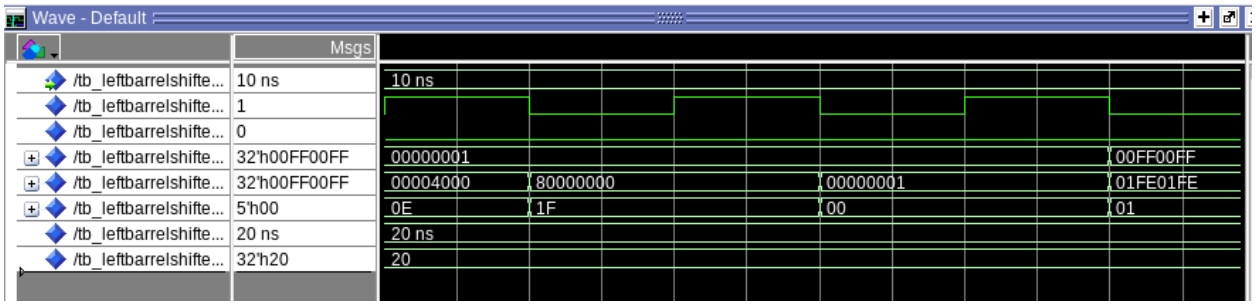


The above screenshot shows tests for SLL by 1, 2, and by 14. Each output behaves as expected. There are 10 additional test cases in the testbench.

[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.

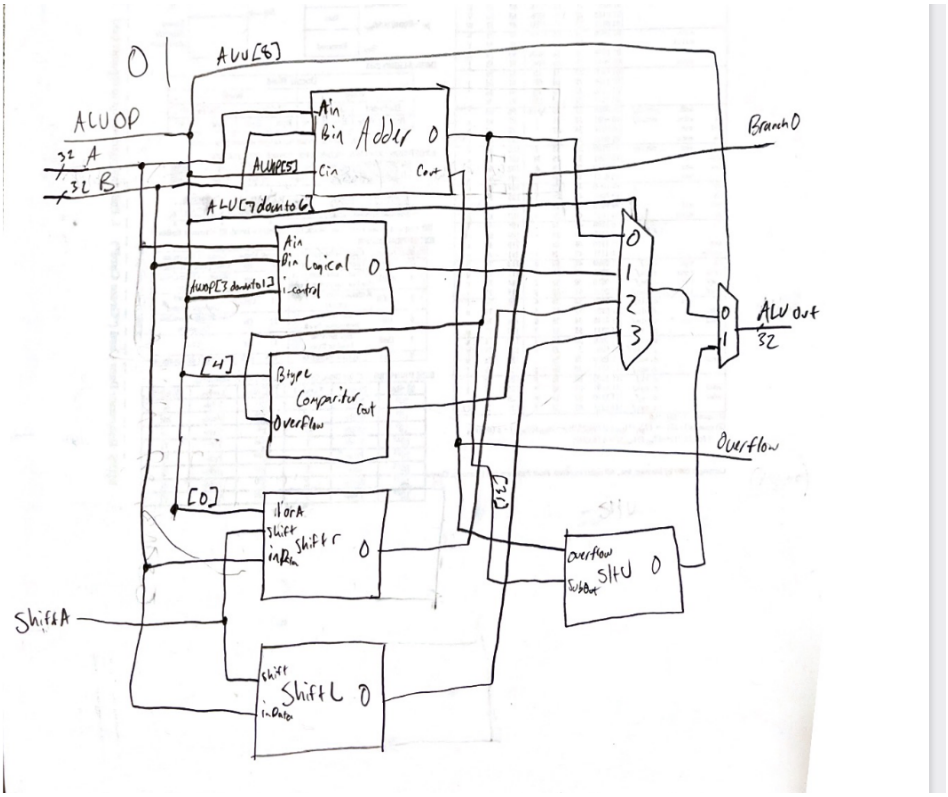
We began our design approach by working from the base schematic in the project documents. As we progressed into the project, we had to decide how to handle the additional instructions. We had to create multiple MUX's to handle inputs of different sizes. In earlier labs we had created a variety of multiplexors that we could use in the project, but our design needed a few more to work properly. Some of the MUX's we implemented are a 5 bit 4 to 1 MUX and a 32 bit 4 to 1 MUX. Another design decision we made was to use a left and right barrel shifter instead of just one. We did this to simplify the debugging process and keep each component as simple as possible for testing purposes.

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



This is a screenshot of some of the results for our Left Barrel Shifter. It is working as expected, in this image we are shifting by 14, 31 and 0. Each of the results that we get in this part are exactly what we expected. We did not implement full test benches for the MUX's that we created as they were fairly simple designs.

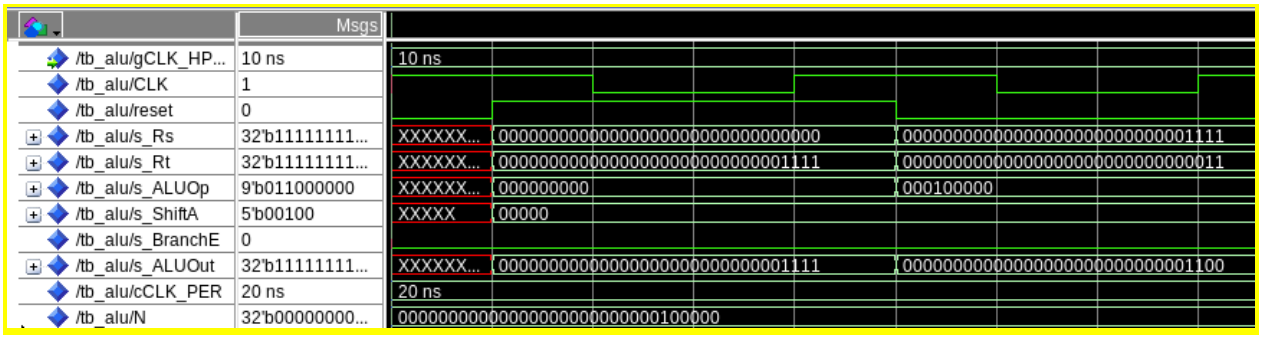
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?



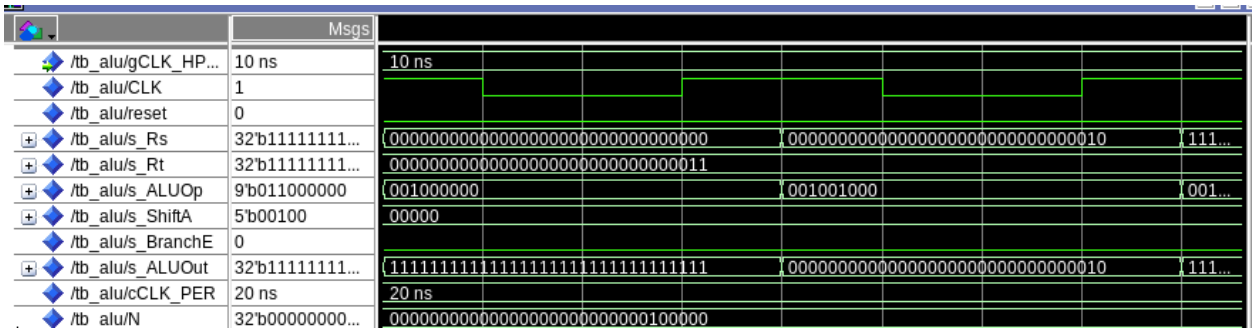
Overflow is coming from the output of our adder/subtractor which is the labeled port Cout. The Zero is renamed to BranchO because it is only one when the conditions of the branch are true. This is calculated from our comparator that we built that utilized the output of our subtractor. Slt

instruction uses its own separate unit to calculate by taking the input of the 31st bit of the subtractor output as well as the overflow from the subtractor. These values are XORed to fit the conditions of having any value, including multiple negative values. Then we have a bit to designate whether we are calling the slt instruction that is fed to a mux that takes the output of out 4to1 mux and the slt unit and returns the value of the slt when 1.

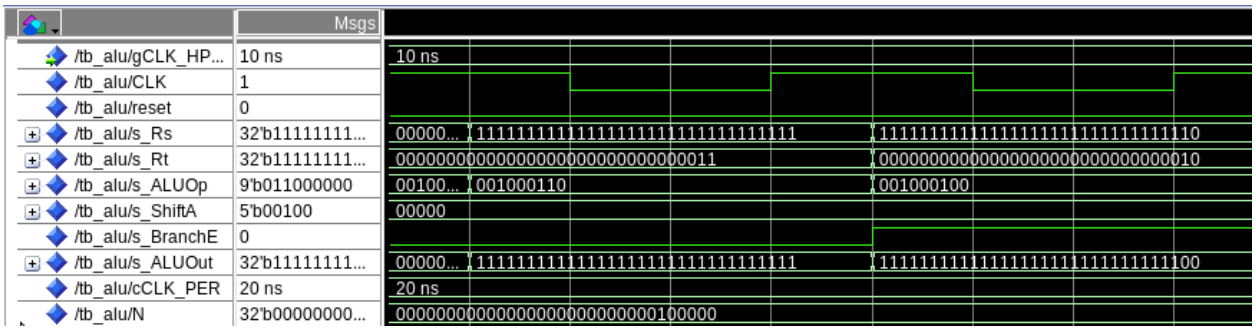
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



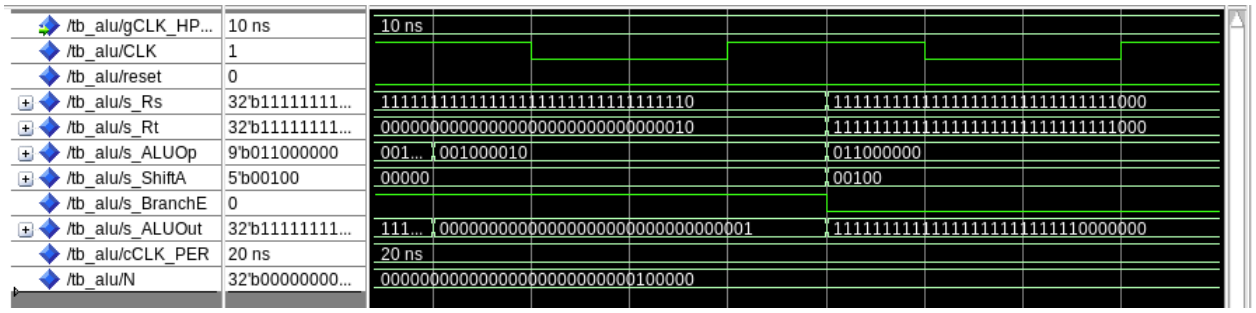
The above screenshot shows the output of an add instruction and then a sub instruction. ALUOut is the expected value for each of them. Additional tests found in testbench file.



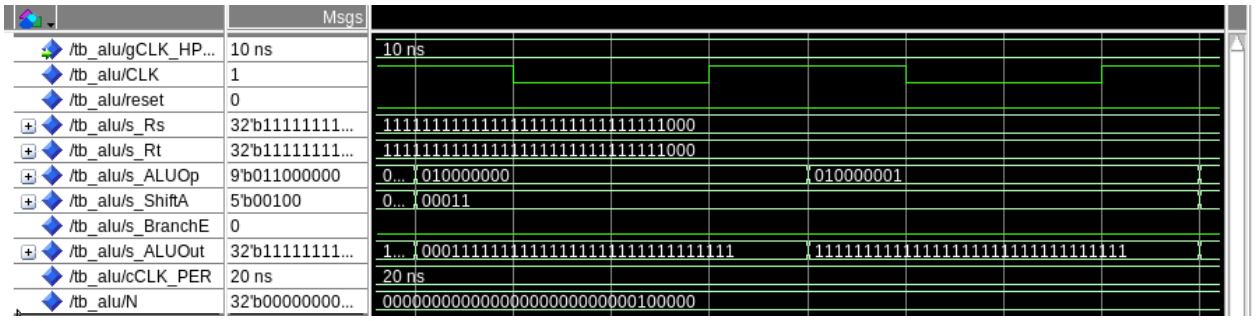
The above screenshot shows the output of tests for the not and the and instructions. Both behave as expected.



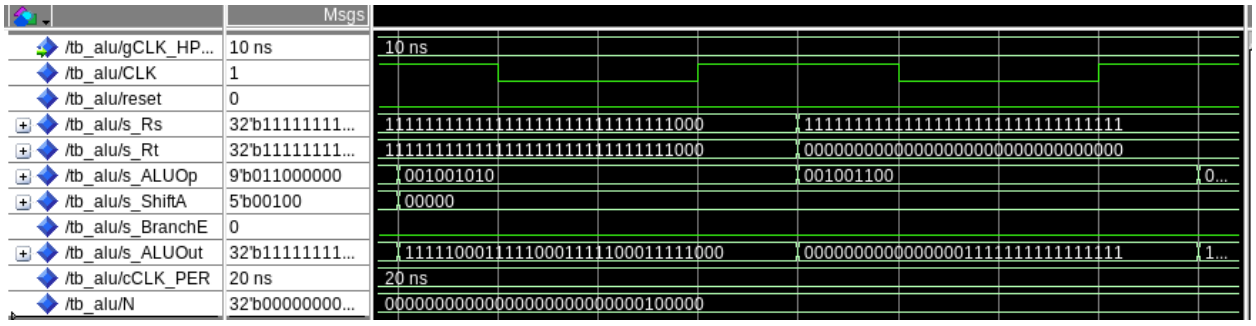
The above screenshot shows the output of an or instruction and an xor instruction. Both behave as expected.



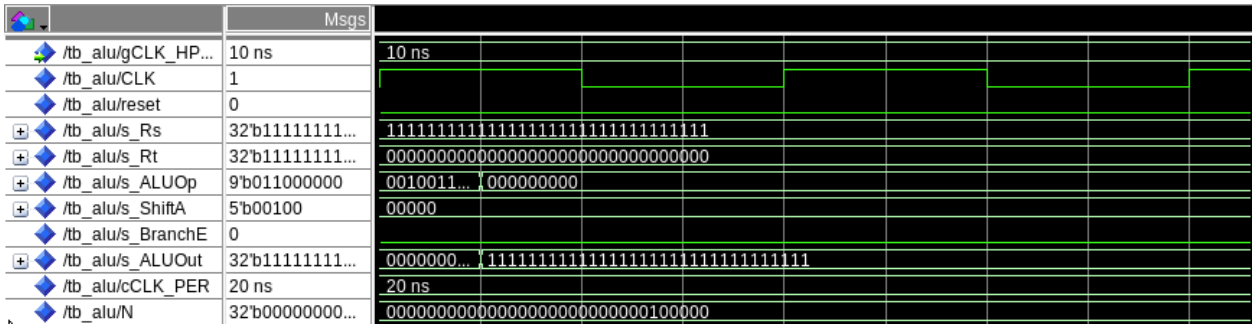
The above screenshot shows the output of a nor instruction and a SLL by 5 instruction. Both behave as expected.



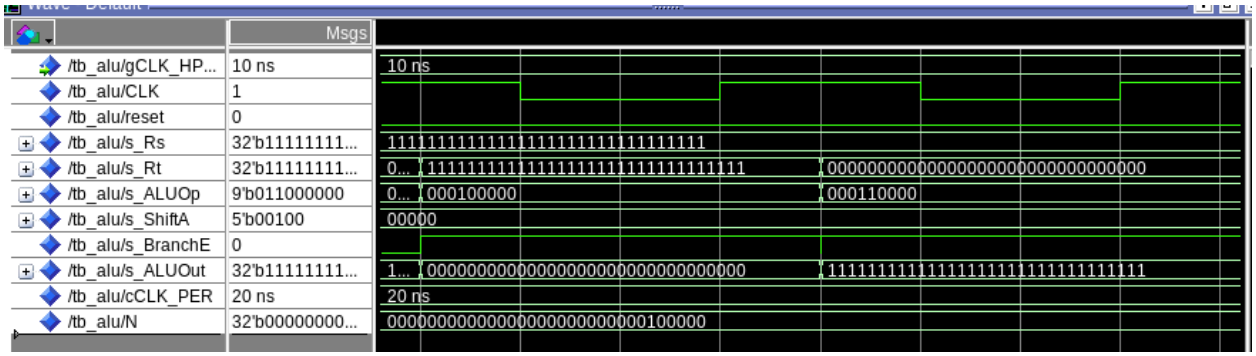
The above screenshot shows the output of a SRL by 3 instruction and an SLA by 3 instruction. Both should return all 1's and they do.



The above screenshot shows the output of a repl.qb instruction, which is behaving as expected by concatenating the 8 bits. We also test lui, which is also behaving as expected by combining the 16 bits from each input.



This screenshot is the output from lw and sw. The opcode is the same and so is the output for both of them. Both behave as expected.



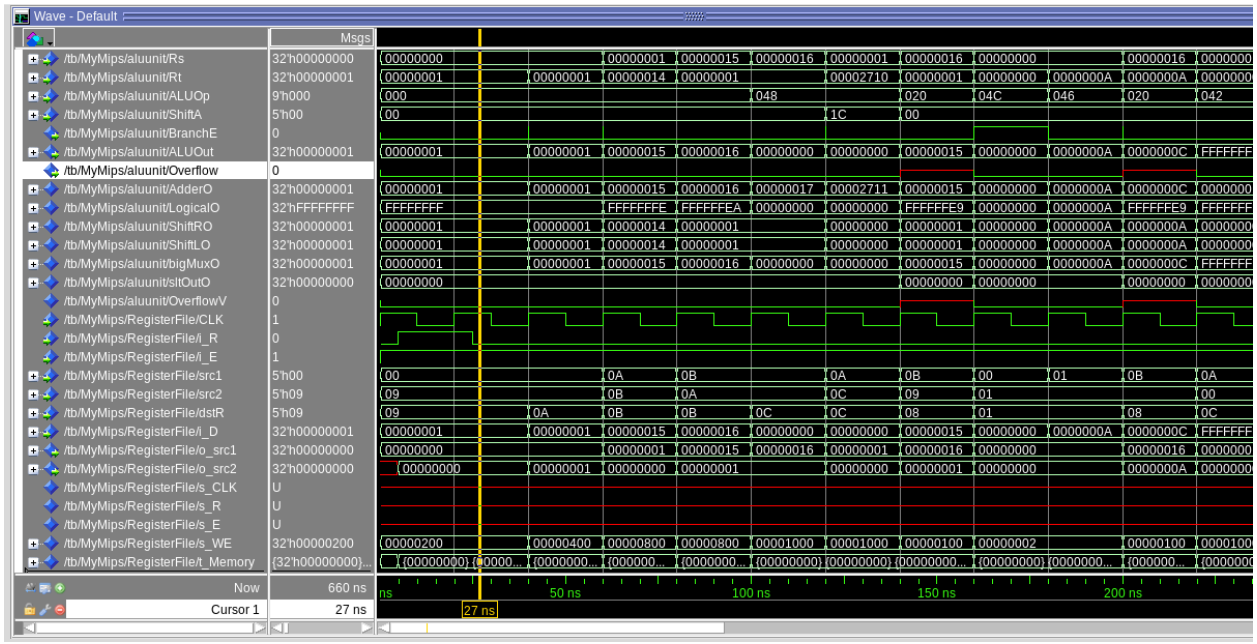
These tests are for beq, which sets the BranchE output to 1 when the two inputs are equal. Then we test bne, which sets the BranchE output to 1 when the two inputs are not equal. Both behave as expected.

[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

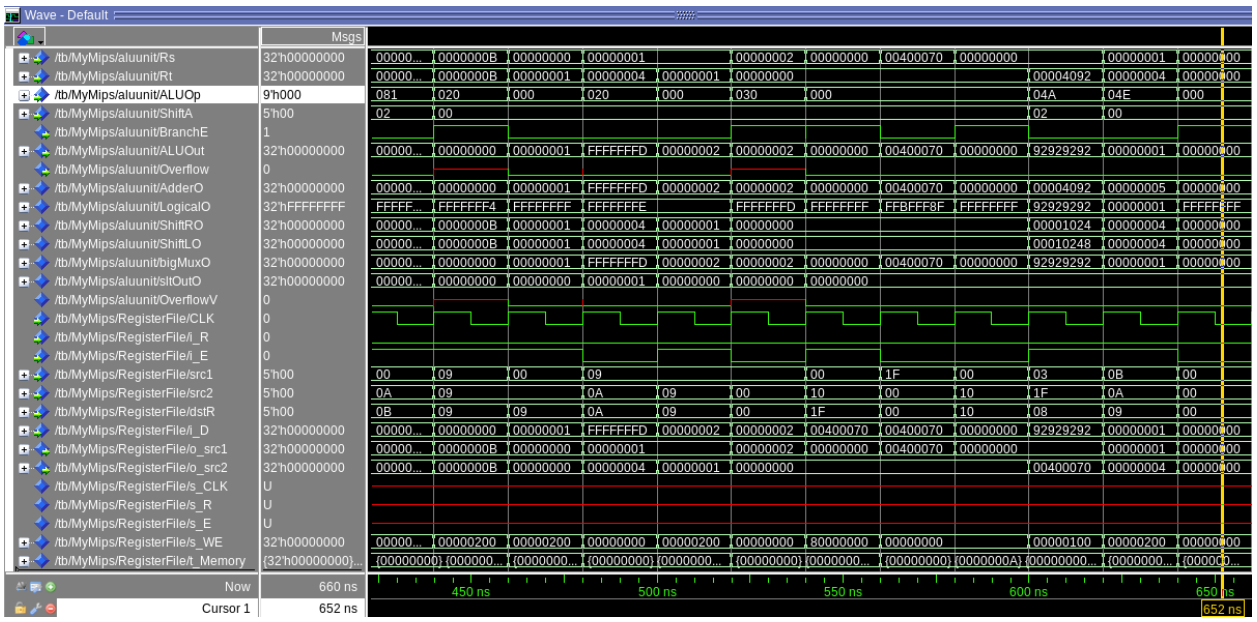
Our test plan is comprehensive because we test every instruction supported by the ALU. The ALU is passing every single test. We also have thorough testing in each component of the ALU, giving us great coverage of the overall component. Screenshots of 1 test for each type of instruction are included above in the solution for part 2.c.v.

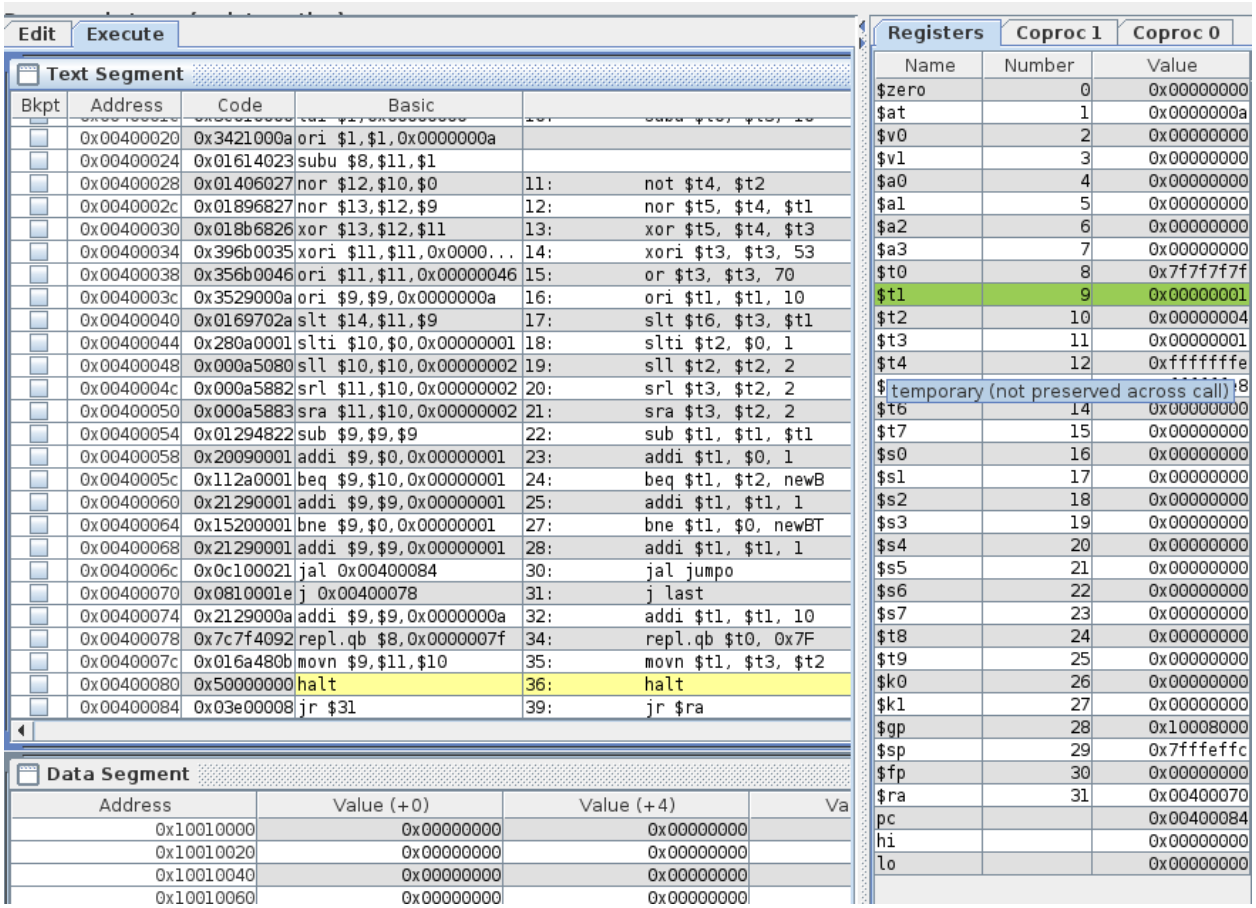
[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[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.



This shows our expected values for the first instruction of addi. This is shown with an ALUOp of 000. This also show our last instructions terminating as expected with a halt instruction shown by the ALUOP 000

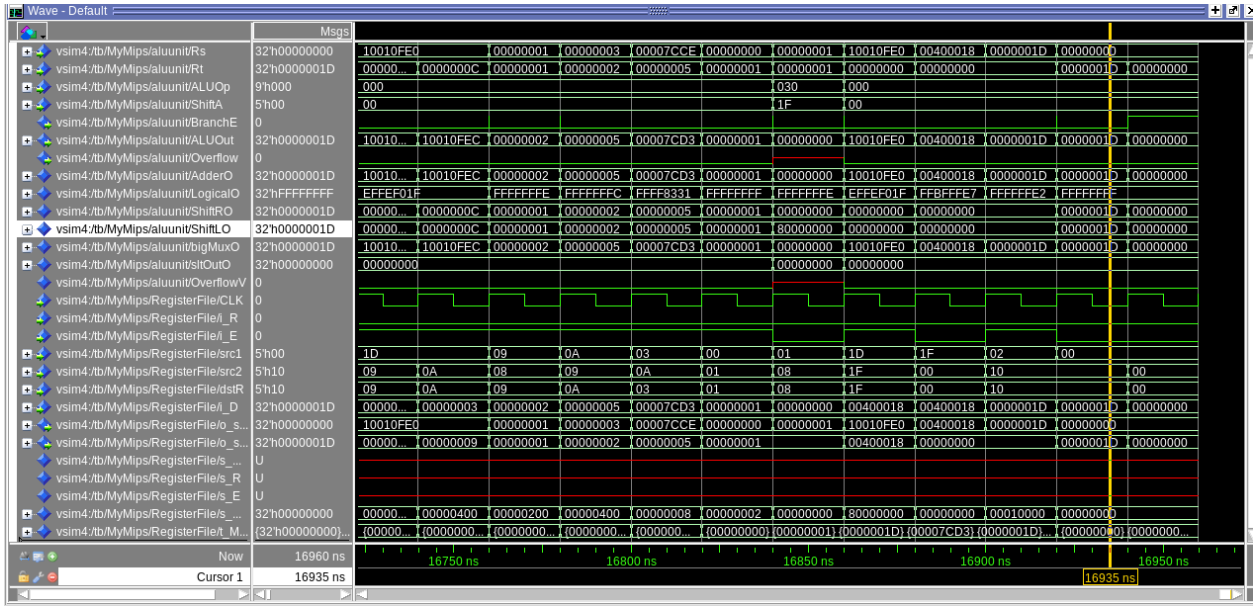




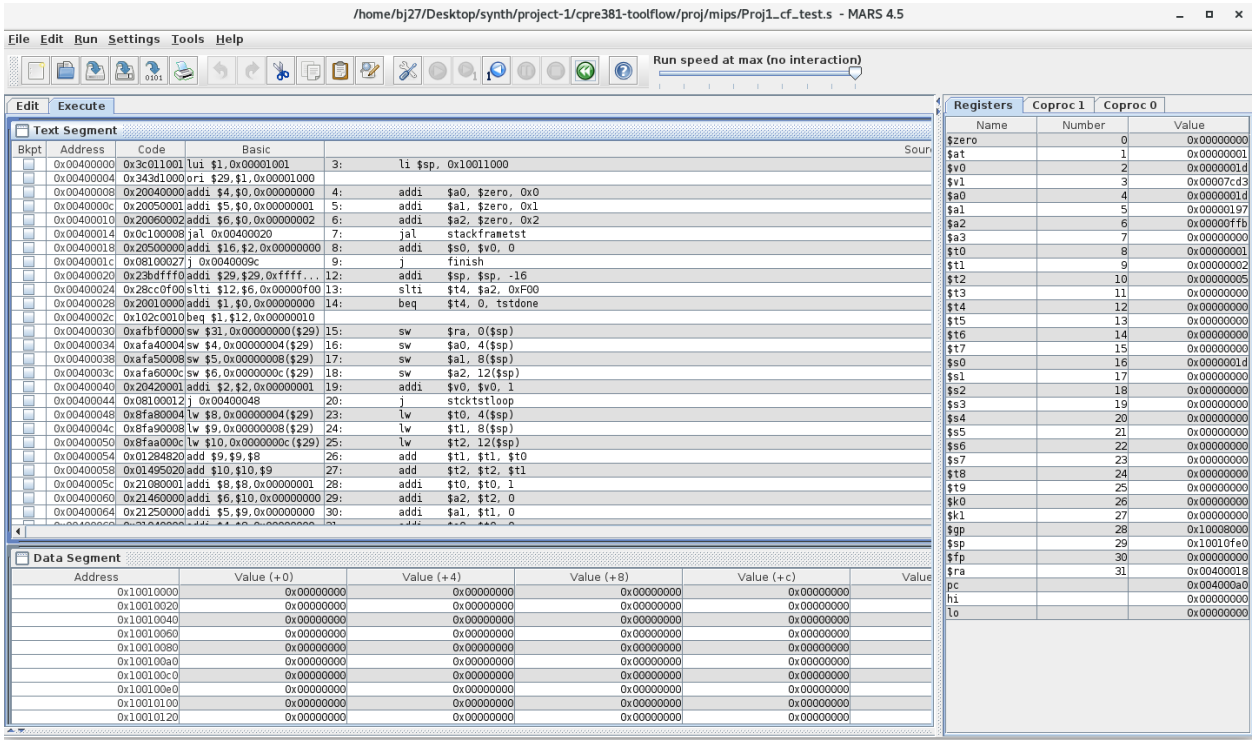
This is our expected output observed in mars

Output of CF

[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.

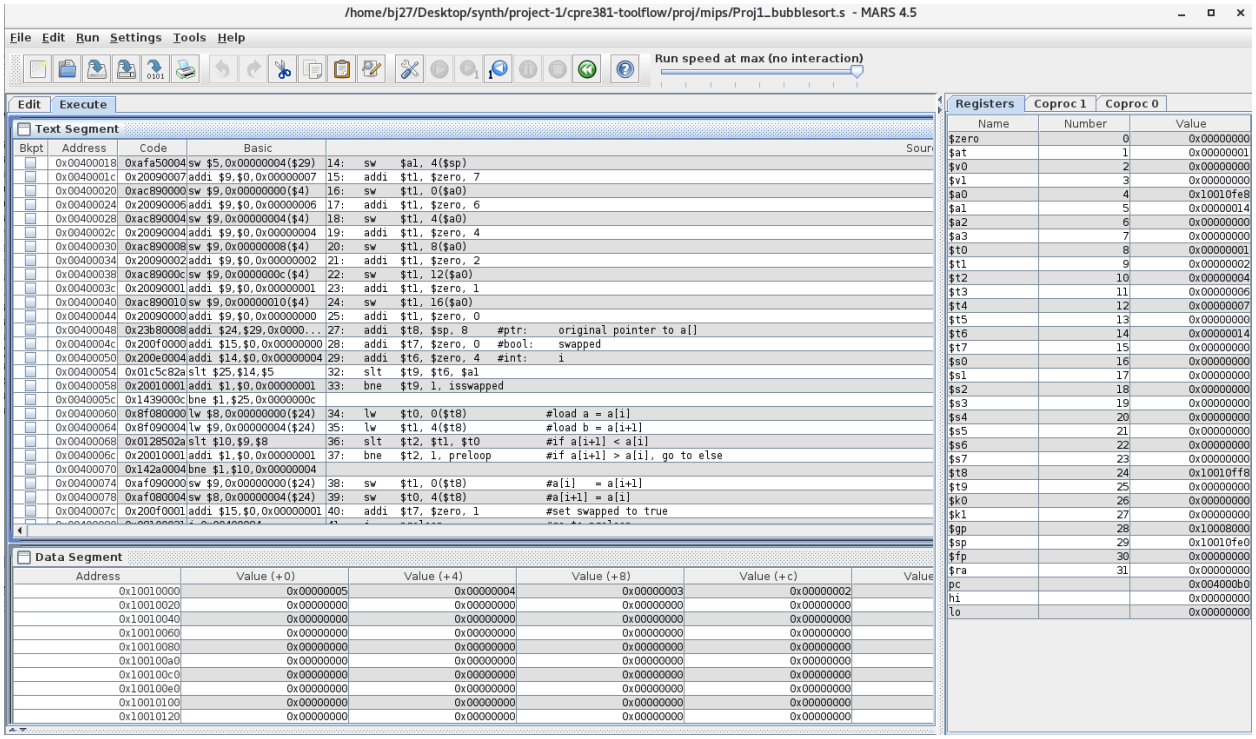


This is the waveform output of cf_tests. The file is included with our submission. You can see the halt instruction at the end with the OPcode of 000. We have also included the register status at the end of running the file in Mars for comparison.



[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.

This is the waveform output of Bubblesort. The file is included with our submission. At the end of the file, you can see the OPcode of 000 for the halt instruction to stop the program. In the MARS output below, you can see the status of each register at the end of the program running.



[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?