CprE 381: Computer Organization and Assembly-Level Programming

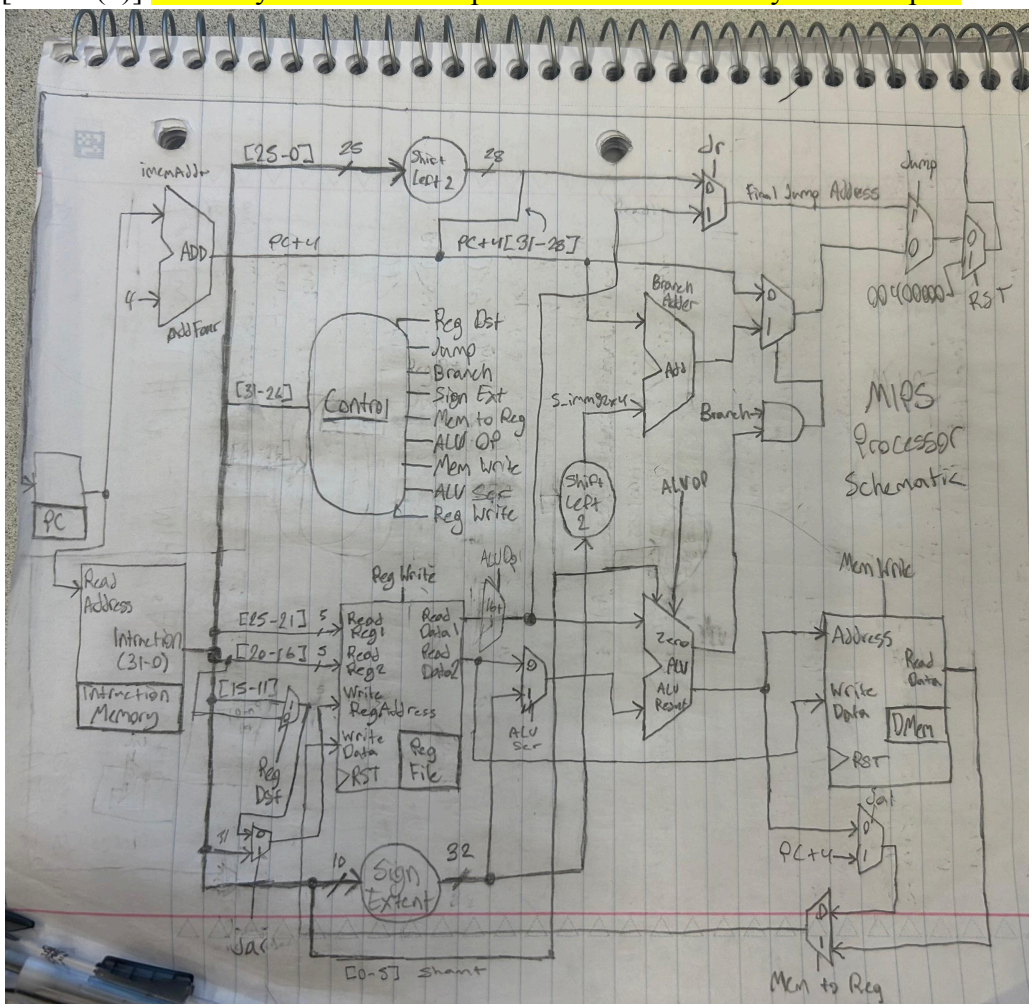
Project Part 1 Report

Team Members: Brandon LeVake
Spencer Opitz
Charlie Wood

Project Teams Group #: Section G, Group 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

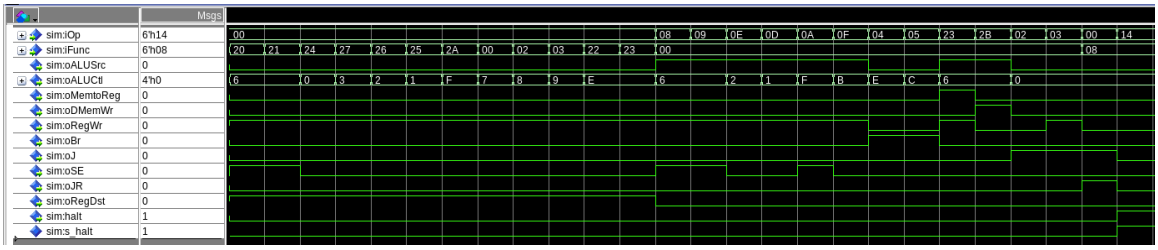


[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the

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.

In Files

[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).

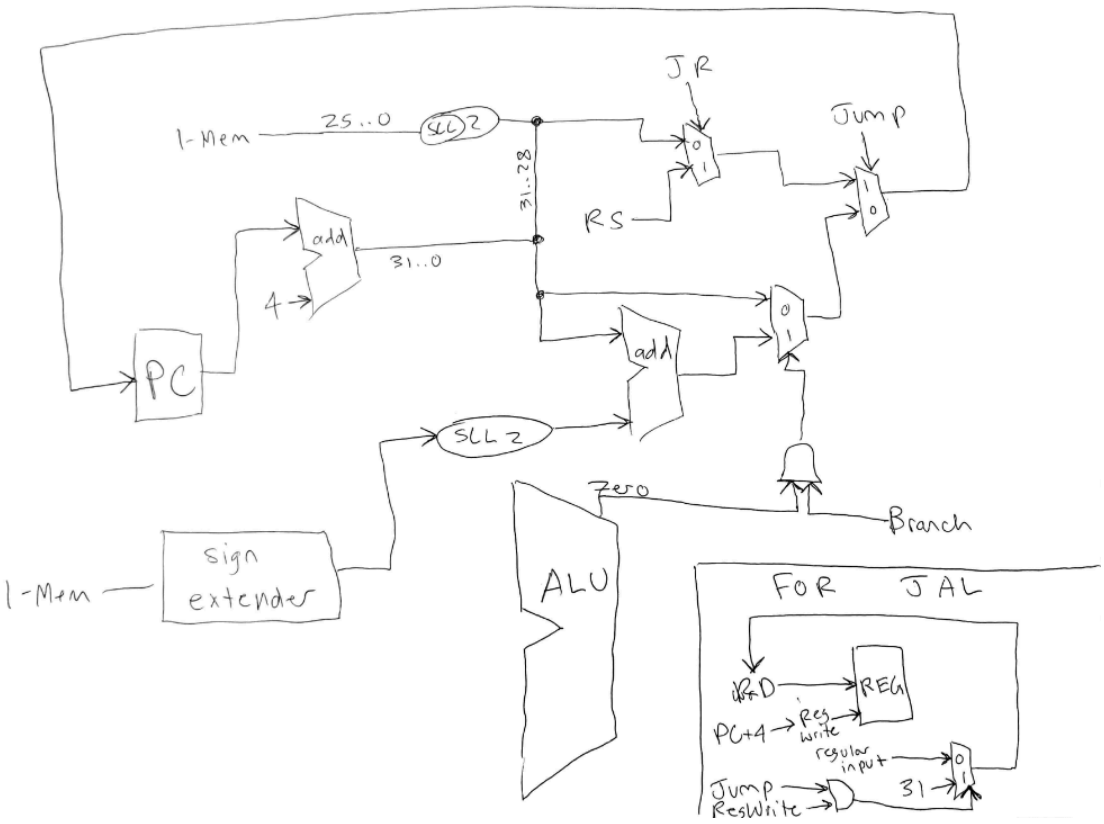


[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

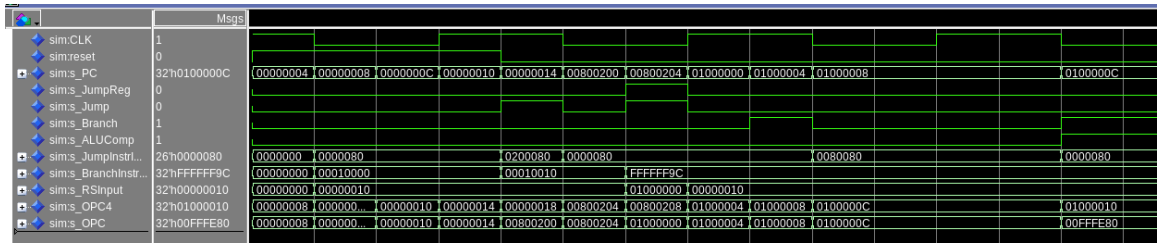
beq - break to a 16 bit immediate
bne - break to a 16 bit immediate
j - jump to a 26 bit immediate
jal - current PC + 4 becomes $ra ($31) and jumps to 26 bit immediate
jr - jump to a 32 bit address stored within a register

[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?

Need signals : JR, Jump, and Branch
JAL can be derived from Jump and Jump and regWrite



[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.



Each of the instructions feed into the next instruction's PC value, adding 4 every time. When there is a jump or branch, it jumps to the correct PC value, and sets bits when it needs to.

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

SRL will replace incoming bits from the left side with 0. This could result in a negative number switching signs when you are trying to divide it.

SRA will replace incoming bits from the left with the same number as the most significant bit. This means that if you are trying to divide a negative value it will not have its sign changed.

SLA is unnecessary because arithmetically multiplying an integer by any power of 2 will result in an even number/0 in the least significant bit. SLA would thus do the same thing as SRL.

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

The barrel shifter component imports i_shifttype which is a standard logic input. After the shift amount is determined by the muxes (both arithmetic and logical separately) the outputs are sent to another set of multiplexers that uses the i_shifttype as the select bit.

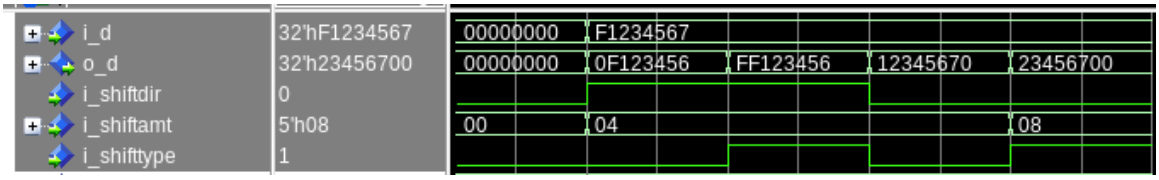
[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

The output from the previous question is then sent to the direction muxes which select between left shifted and right shifted using i_shiftdir as the select bit.

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

input is 0xF1234567

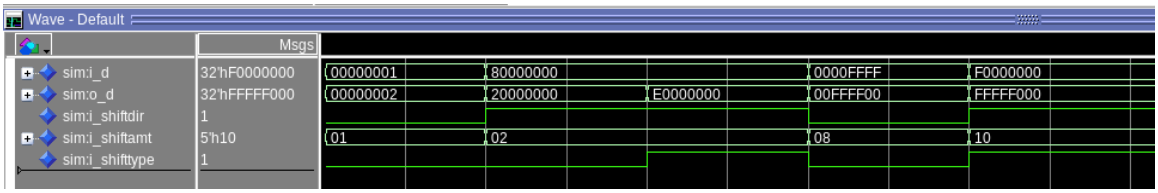
first test is shift 4 bits right logical
second test is shift 4 bits right arithmetic
third test is shift 4 bits left
fourth test is shift 8 bits left with i_shifttype set to the arithmetic option



[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.

Our approach was to take each module and see how would could simplify them. For example instead of adding a nor module we added a invg to or. One design decision we made was to separate the zero calculation from the equality module as the mux would require aluOp. Separating the two modules causes testing to be an easier process.

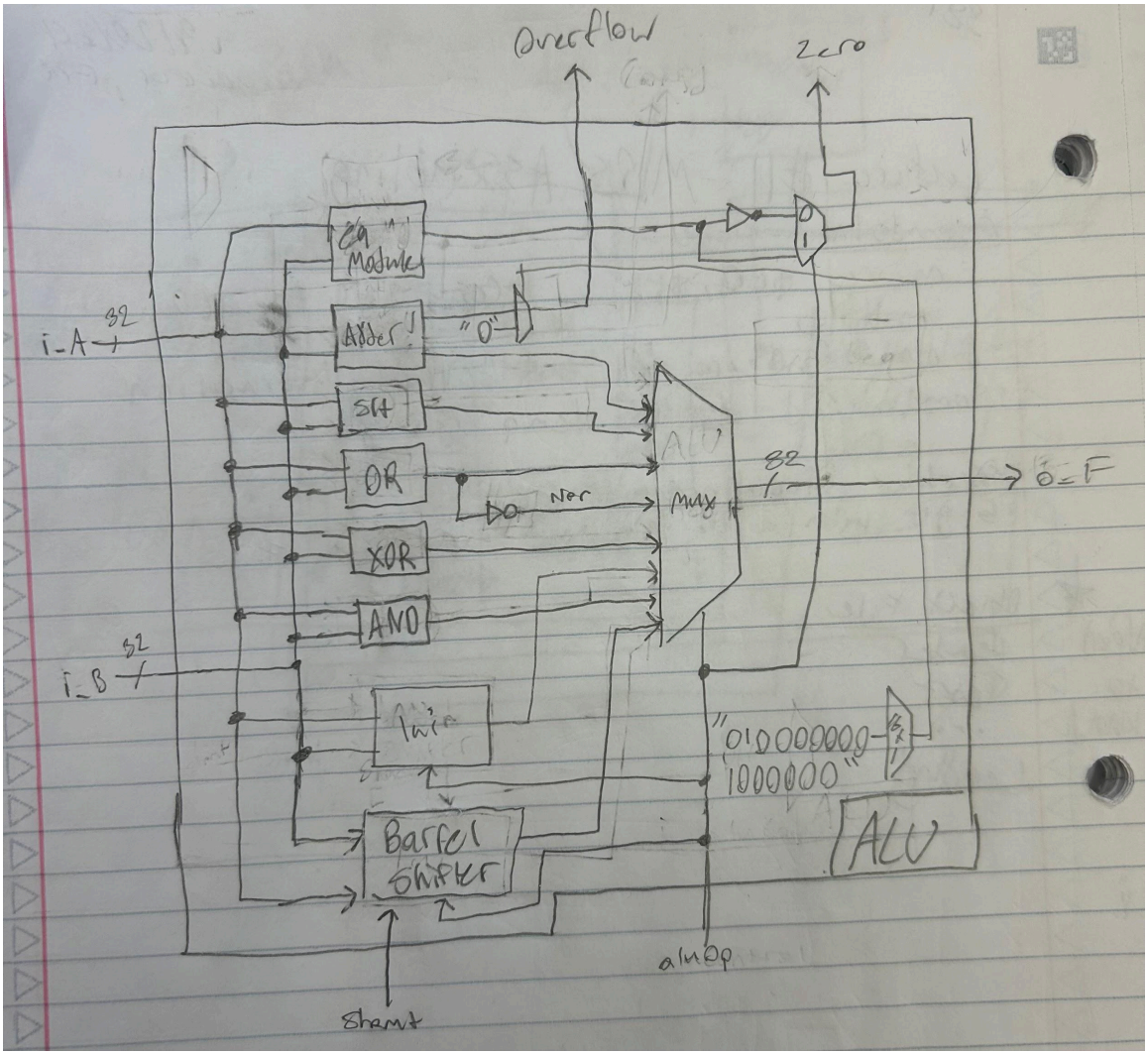
[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



In the first test its asked to shift 1 and as you can see if accomplished that goal. Similarly all of the following testbenches match there expected values.

[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?

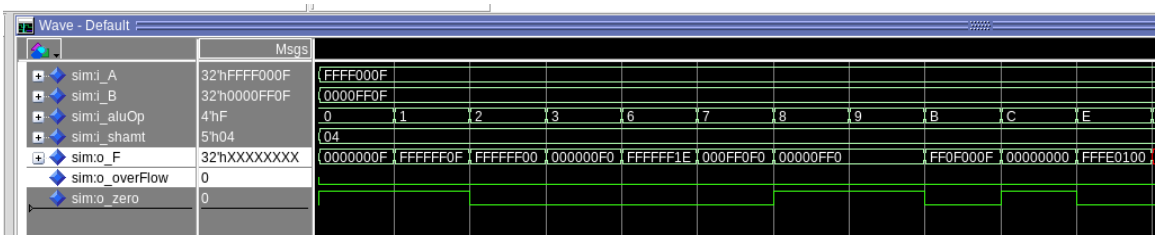
Overflow is calculated by taking the 31st bit of the Adder(and sub) and xoring the 31st and 32nd bits of the carry signal. Zero is calculated by xoring the output of out control and then using or to see if it equals zero. Slt is calculated by subtracting A and B and then taking the signed bit of the output.



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

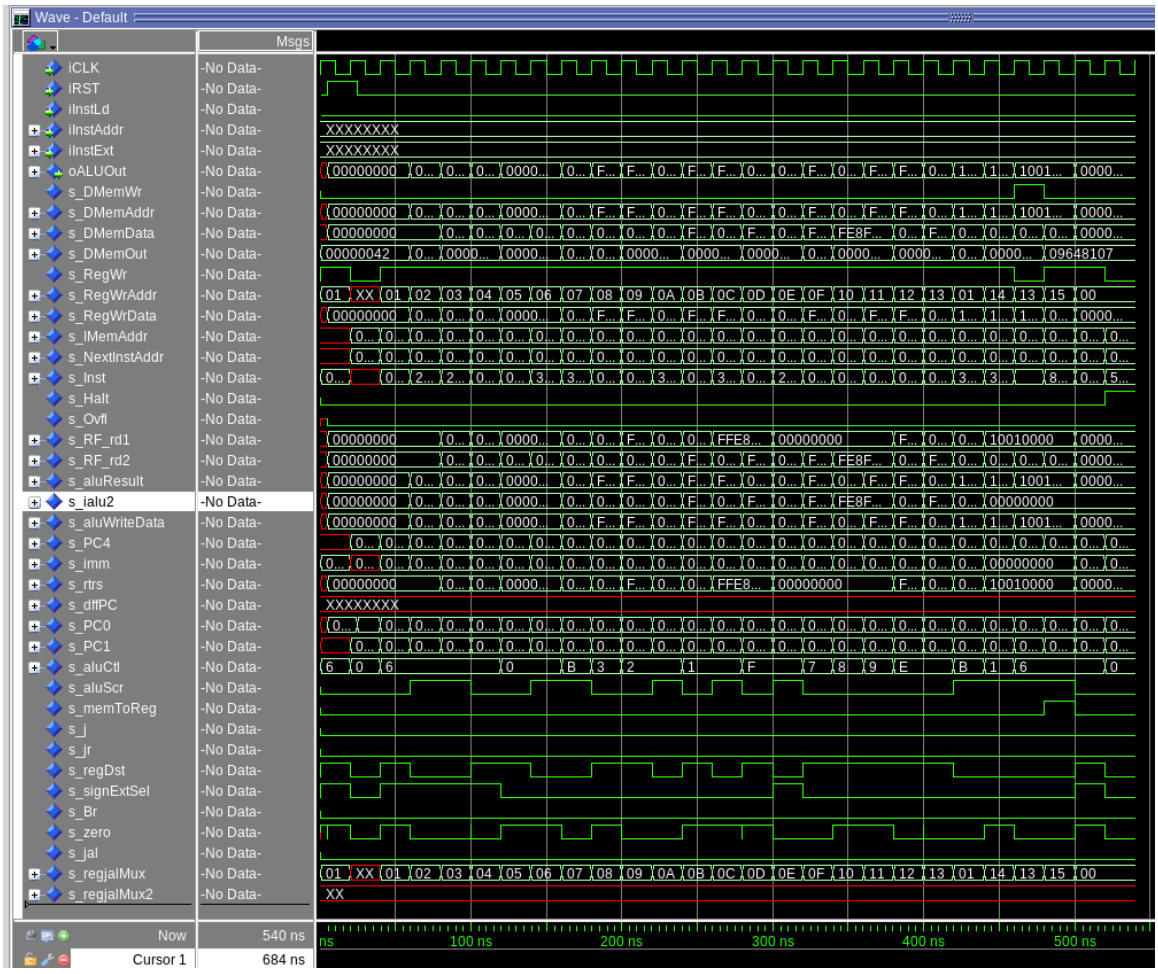
The instruction input for the different operations is different for every operation. In the waveform, this corresponds to the signal that shows the instruction, but it is easier to see the control bits that are different for every instruction.

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

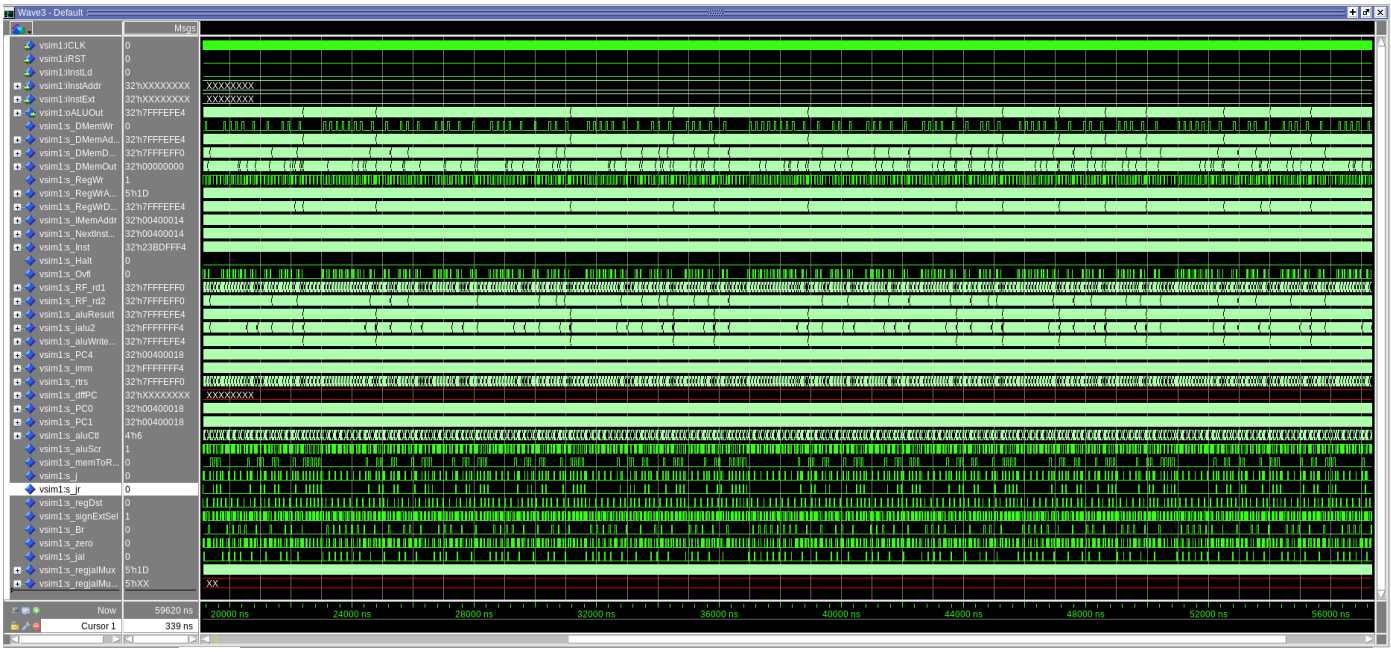


This test plan is comprehensive because it goes over every instruction with similar input. This shows how the output for every instruction changes based on what the ALU op input is.

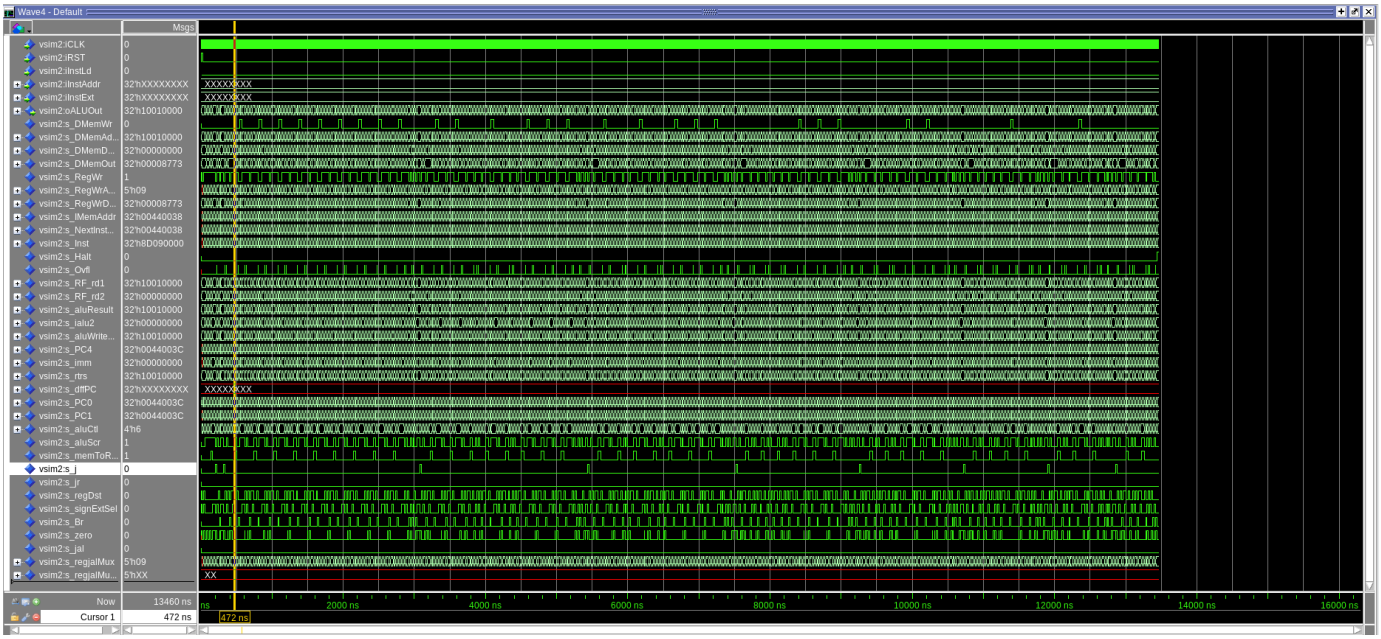
[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.



Base Test



CF Test



Bubble Sort

[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.

Done in mips folder

[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.

Done in mips folder

[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.

Done in mips folder

[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?

Max Frequency was 25.50mHz, the critical path took 42.659 ns
The critical path came from load word instructions, where you need to go through regfile, the alu, memory, and then write back to reg file.