CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: _____Simon_____

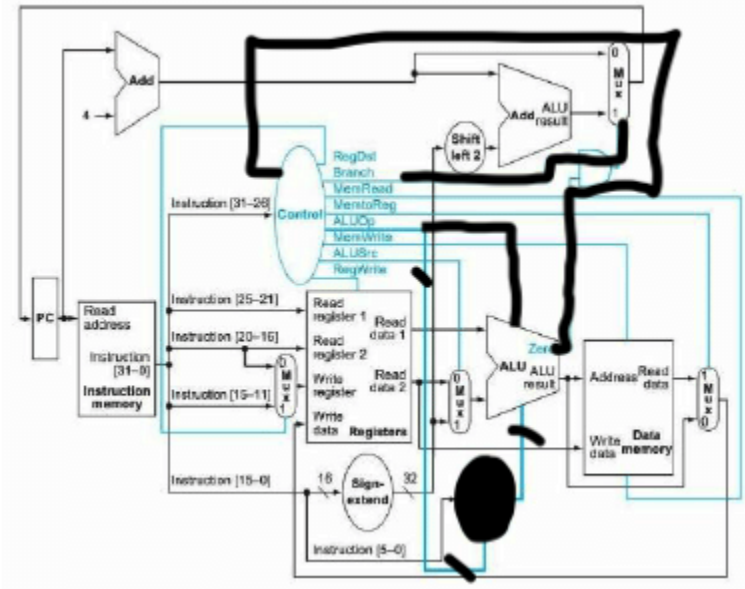
_____Evan_____

_____Mitch_____

Project Teams Group #:_____This does not matter__

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.



Instead of creating a seperate ALU control unit (which we have blotted out in this general form of a single cycle processor), we created an additional four bit control signal in the main control unit and ran it directly to the ALUs control block. We slashed unnecessary signals in the general schematic which were discarded because of this change.

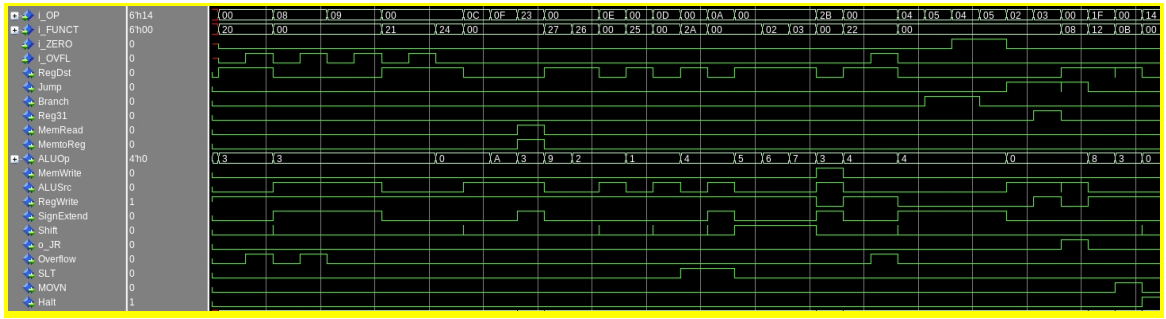
Additionally, we removed the and gate which connects zero and branch, instead choosing to route both signals directly from the control to the ALU results mux (in the case of branch) and the ALU itself (in the case of zero.)

Other than those two slight changes, our signal cycle processor is quite similar to the general form.

[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.

https://iowastate-my.sharepoint.com/:x:/g/personal/wmhudson_iastate_edu/EQ_GqhvRcBdIgbQmkyFFG8QBRf539neiumIc0J4sUIT-sQ?e=BQoC6R

[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).



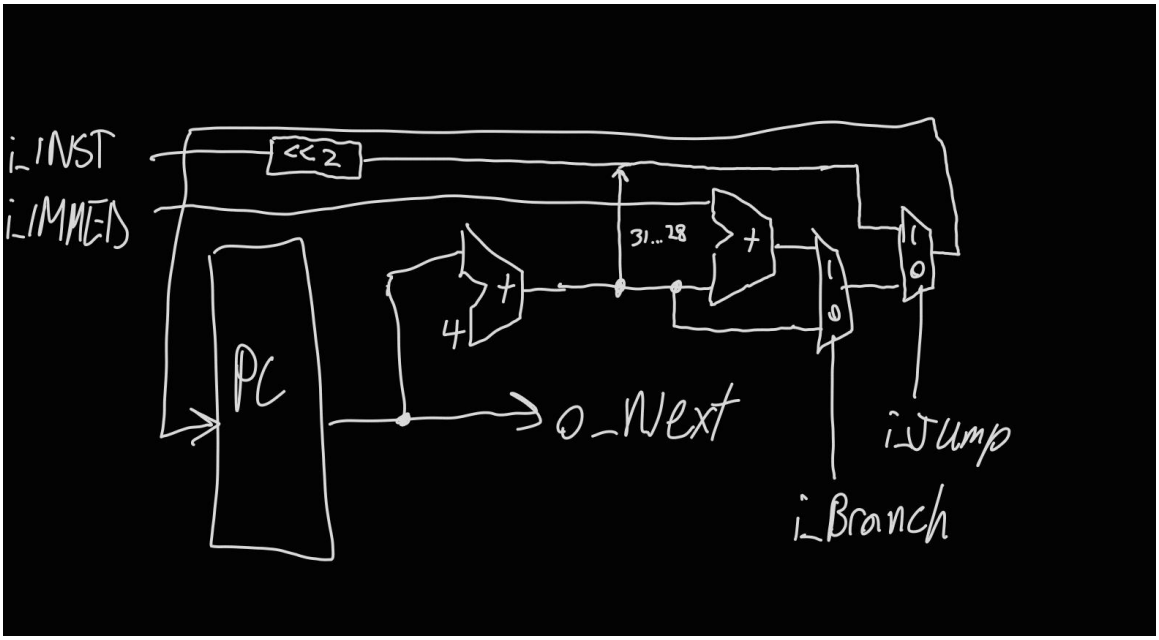
Each of the control signals from the above spreadsheet is shown in the above image, with accurate values for each opcode and function code. The zero input is used to calculate bne or beq, and the i_OVFL is used to calculate the overflow output signal. If the operation can produce overflow, and the alu outputs overflow, then the overflow signal is outputted high.

[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.

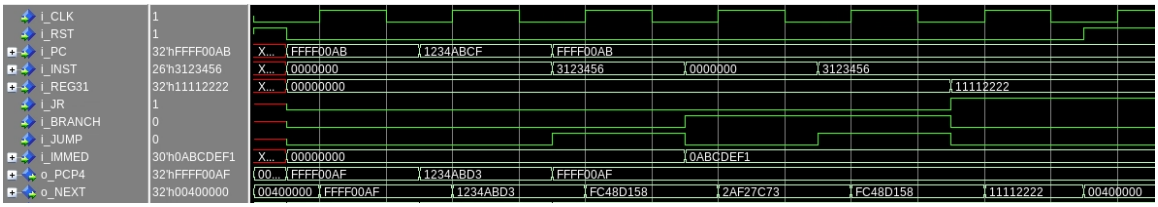
Branch, Jump, JR, Jump and Link and just regular plus 4.

[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?

Branch, jump, jr control signals, as well as a reg31 signal for the $ra address



[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.



First, the i_PC is the input PC for the next clock cycle. o_NEXT is the current pc to fetch with. Whenever i_RST is high, the pc is reset to 0x00400000 which is necessary for mars. Most of the time the next value is just the old value plus 4, which you can see in o_PCP4. Whenever jr is high, the i_REG31 value is used directly for the jump, because that is the $ra register. When branch is high, the i_IMMED value is sign extended, shifted, and added to PCP4 to get the next value. Finally, whenever jump is high, the instruction input is shifted, and the top 4 bits from PCP4 is appended to the top of it.

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

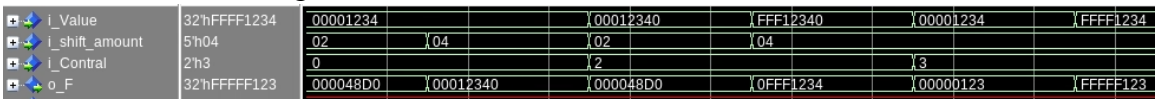
srl is a logical shift, meaning the bits appended to the end are 0s, whereas sra is an arithmetic shift, meaning the bits appended to the end are the sign bit (preserves signing). MIPS doesn't have sla because an arithmetic shift left does the same thing as a logical shift left, appending 0s to the lsb

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

It does it with a control signal that tells it to use arithmetic or logical shifting. by telling it to use zeros fro arithmetic and what ever is the sing bit is for the logical.

[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 barrel shifter has a control signal which determines whether or not it will shift left or shift right. This is implemented by the inclusion of an additional barrel shifter that has been reversed to support shifting left.

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



i_Value is the input value, i_shift_amount is the amount of shift, and o_F is the output
i_Contral is the control value:
00: o_F = i_Value << i_shift_amount
10: o_F = i_Value >> i_shift_amount
11: o_F = i_Value >>> i_shift_amount

step 0: 0x1234 << 2 = 0x48d0
step 1: 0x1234 << 4 = 0x12340
step 2: 0x12340 >> 2 = 0x48D0
step 3: 0xFFF12340 >> 4 = 0x0FFF1234
step 4: 0x1234 >>> 4 = 0x123
step 5: 0xFFFF1234 >>> 4 = 0xFFFFF123

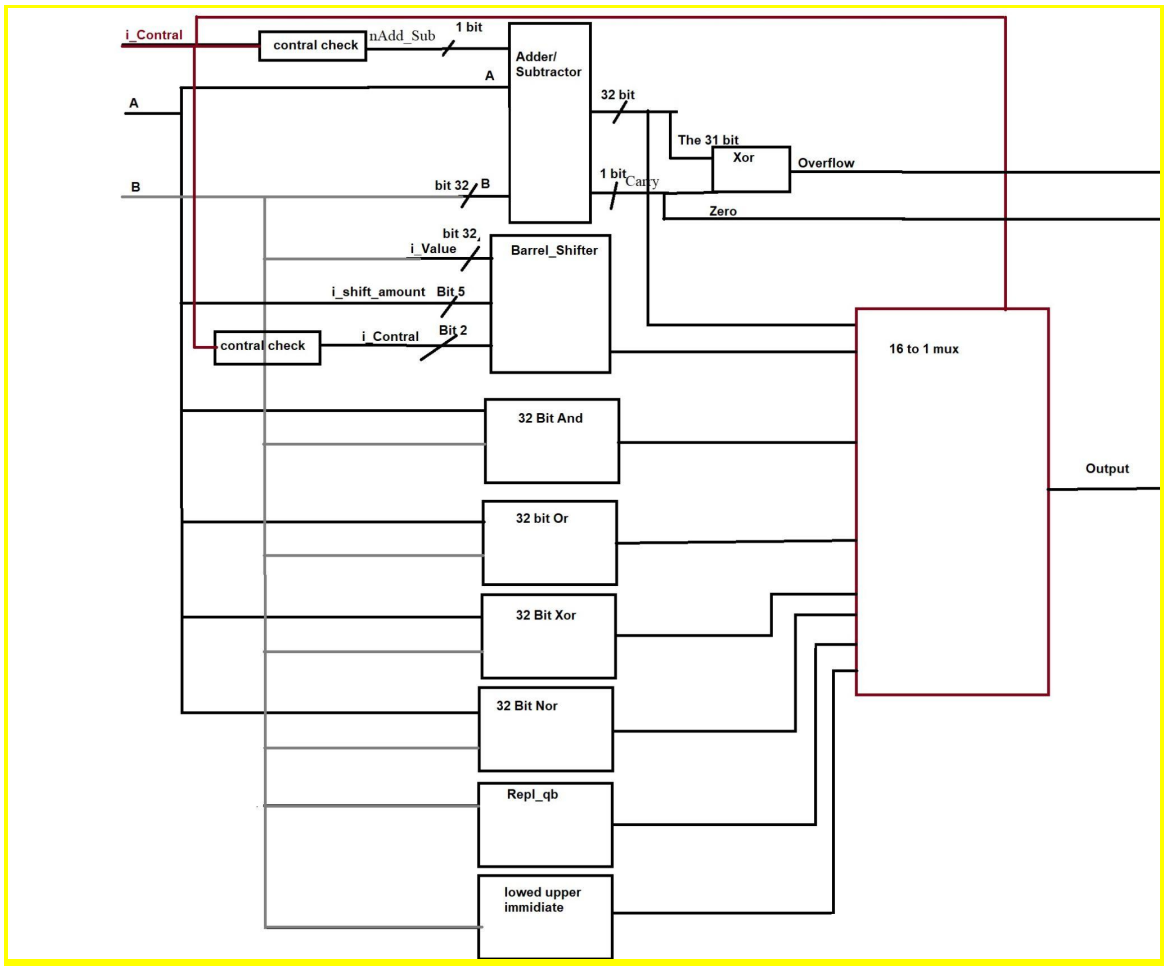
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

One of the design decisions we needed to make was on what each control value would be for which instruction, which took a long time to coordinate between each of the components.
Certainly the authors referenced images of a single-cycle processor schematic while designing their own - these schematics were a good place to start while trying to understand the way that the different components interacted with each other.

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

The only other functional units in the ALU is the AddSub, which is from previous labs, and the bitwise logical operations, which are just arrays of each logic unit.

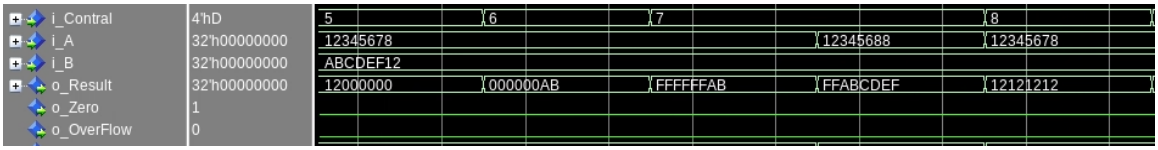
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?

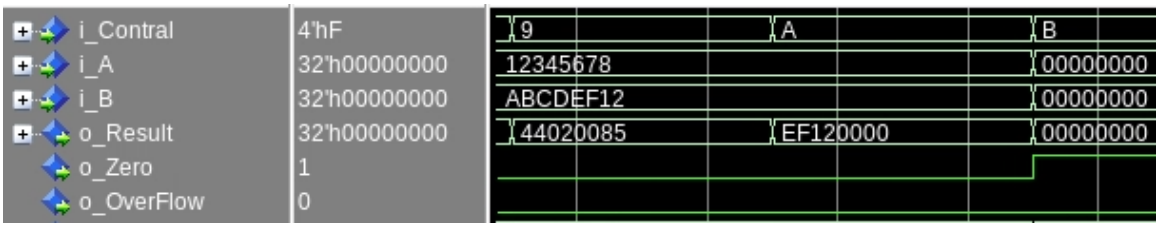


There is a special component that accomplishes SLT

Zero is when the output is zero, and the overflow is calculated from the carry out and second to last carry out being xored together

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



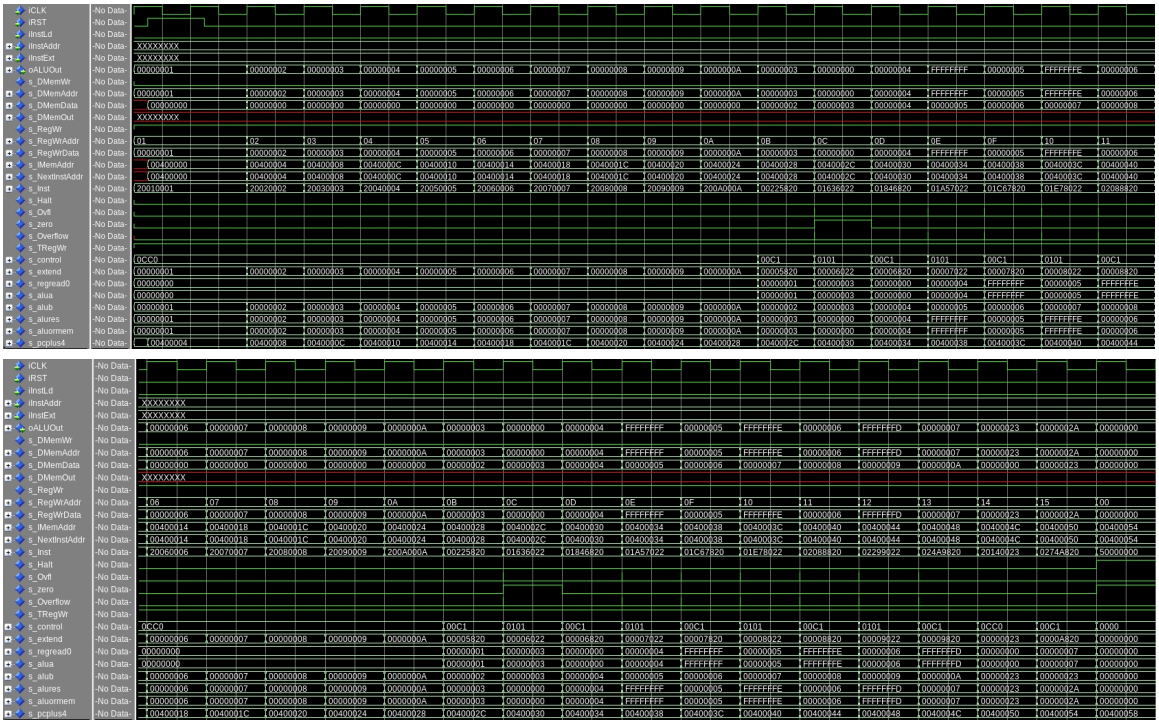


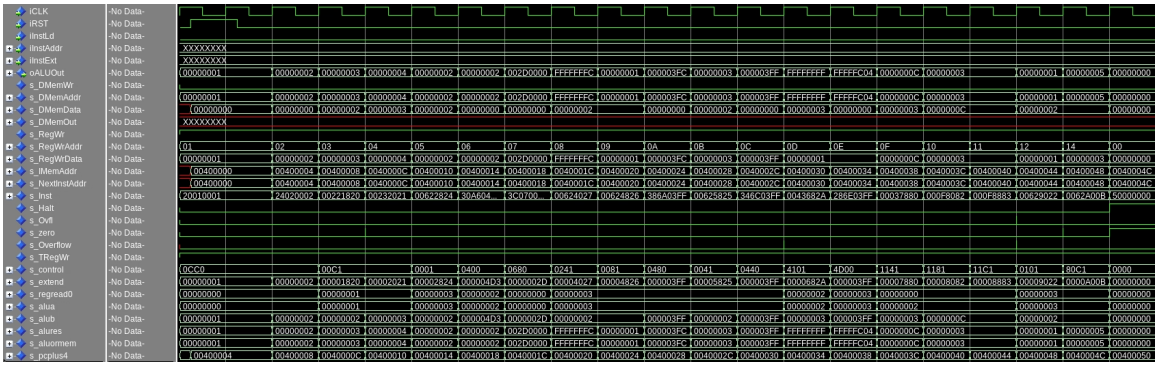
i_Contral is the control values:
0 = a & b, 1 = a | b, 2 = a xor b, 3 = a + b, 4 = a - b, 5 = a << b, 6 = a >> b, 7 = a >>> b, 8 = repl.qb, 9 = a nor b, A = lui

Manually doing the math with each operation provides the same result as o_Result

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

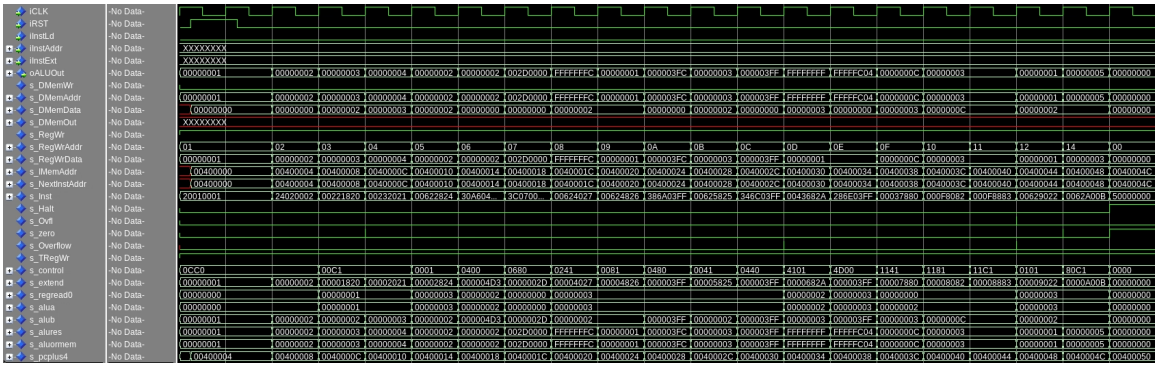
Our test plan is comprehensive because it tests positive and negative values for most of the operations that need it, as well as works for a large program with large numbers.





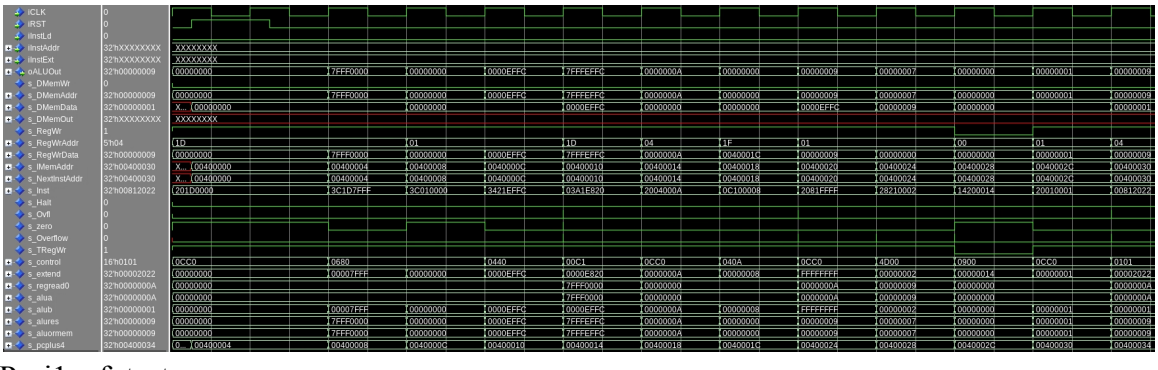
This is the modelsim for the given lab3Seq.s and the Proj1_base_test.s that runs every instruction at least once.

[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.



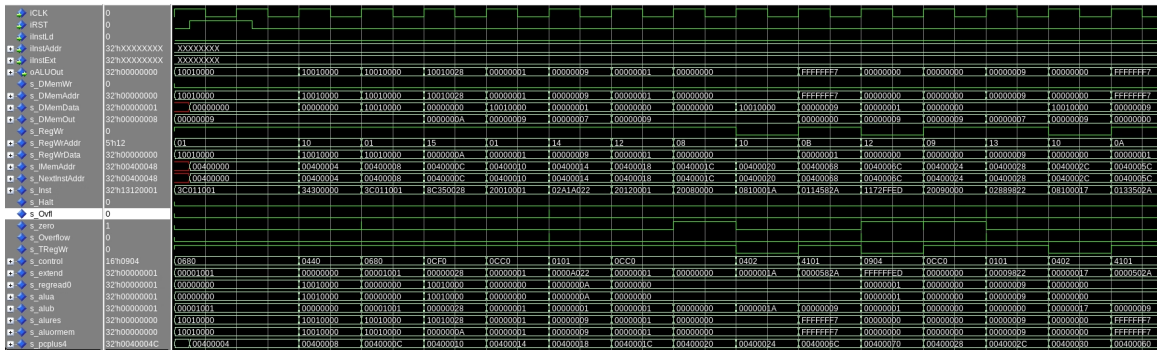
Proj1_base_test.s

Each step provides the correct output for the instruction as designed.



Proj1_cf_test.s

The fibonacci result is correct, ex: fib 10 is 34.



Proj1_bubblesort.s

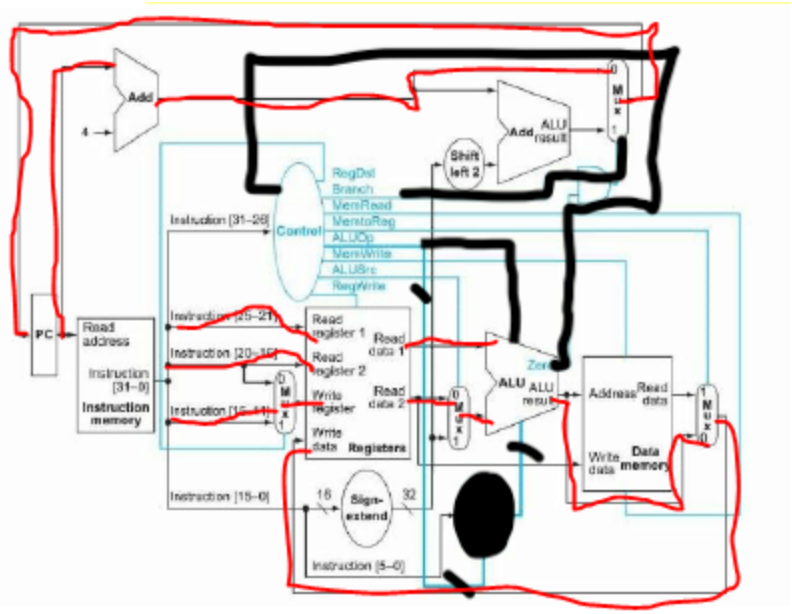
The bubblesort correctly sorts the memory, giving a sorted array at the end, from least to greatest.

[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.

[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.

[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.

[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 highest frequency the processor can run at is 24.81 MHz

The critical path of the processor, or the longest possible instruction, has been detailed in red. To increase the maximum frequency of the processor, we should focus on the ALU and memory modules