CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

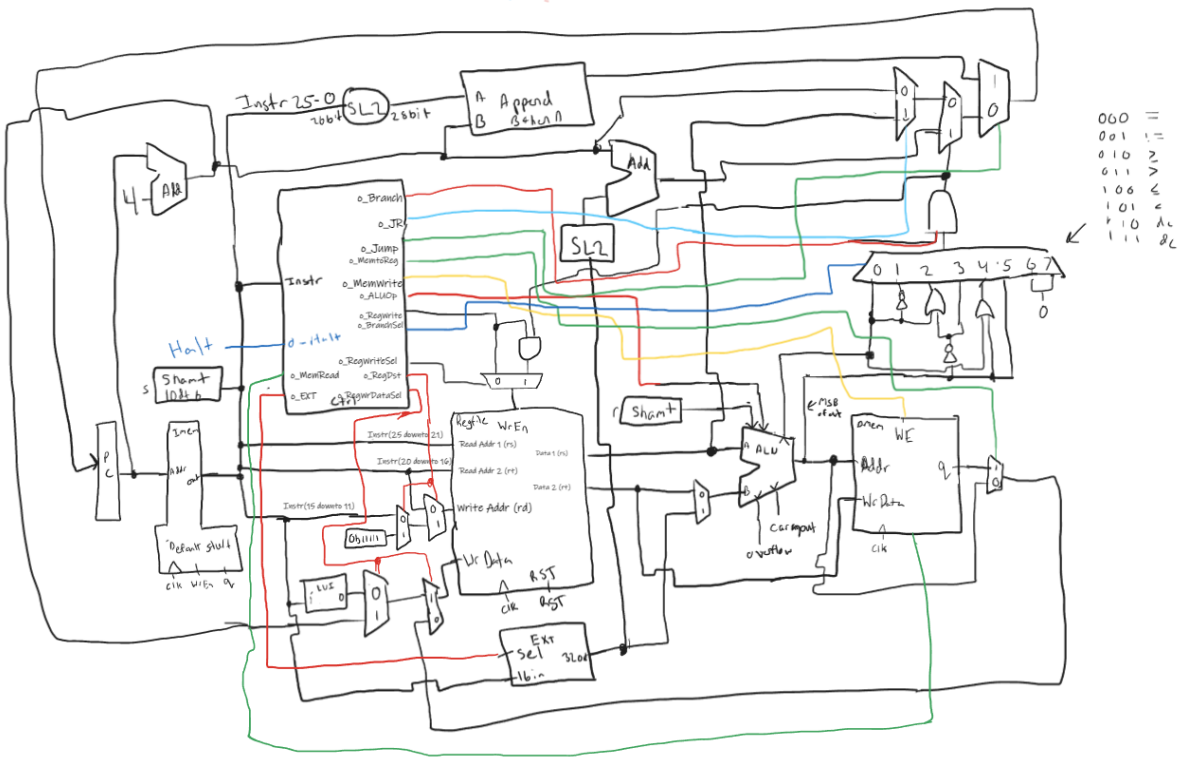
Team Members: Noah Ross _____

Alex Brown_____

Project Teams Group #:_____5_____

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

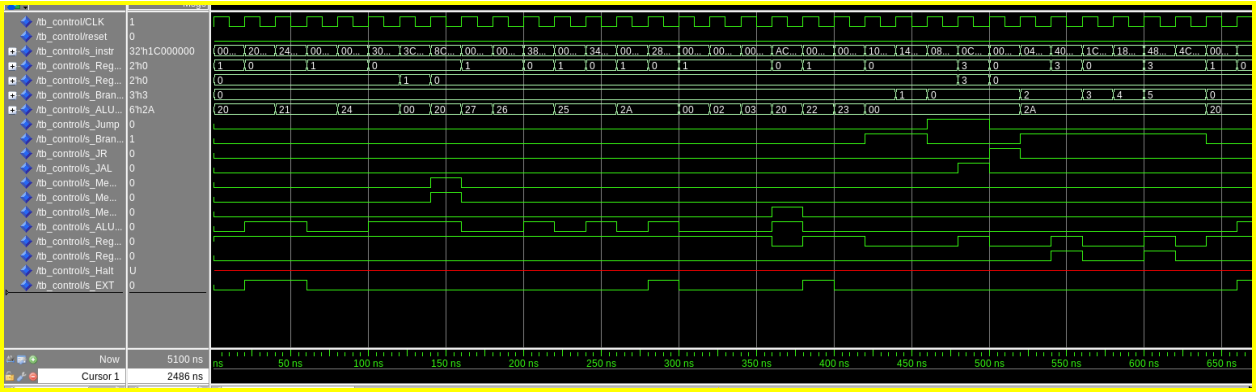
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[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.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Instructions:	RegDst	RegWrData	Jump	Branch	BranchSel	JR	JAL	MemRead	MemtoReg	MemWrite	ALUSrc	RegWrite	RegWriteSel	ALU Control	OP code	Function	Instructions:
2	add	01	00	0	0	000	0	0	0	0	0	0	1	0	100000	000000	100000	add
3	addu	01	00	0	0	000	0	0	0	0	0	0	1	0	100001	000000	100001	addu
4	addi	10	00	0	0	000	0	0	0	0	0	1	1	0	100000	001000	N/A	addi
5	addiu	10	00	0	0	000	0	0	0	0	0	1	1	0	100001	001001	N/A	addiu
6	sub	01	00	0	0	000	0	0	0	0	0	0	1	0	100010	000000	100010	sub
7	subu	01	00	0	0	000	0	0	0	0	0	0	1	0	100011	000000	100011	subu
8	and	01	00	0	0	000	0	0	0	0	0	0	1	0	100100	000000	100100	and
9	andi	10	00	0	0	000	0	0	0	0	0	1	1	0	100100	001100	N/A	andi
10	lui	00	01	0	0	000	0	0	0	0	0	1(N/A)	1	0	000000 (N/A)	001111	N/A	lui
11	lw	00	00	0	0	000	0	0	1	1	0	1	1	0	100000	100011	N/A	lw
12	nor	01	00	0	0	000	0	0	0	0	0	0	1	0	100111	000000	100111	nor
13	xor	01	00	0	0	000	0	0	0	0	0	0	1	0	100110	000000	100110	xor
14	xori	00	00	0	0	000	0	0	0	0	0	1	1	0	100110	001110	N/A	xori
15	or	01	00	0	0	000	0	0	0	0	0	0	1	0	100101	000000	100101	or
16	ori	00	00	0	0	000	0	0	0	0	0	1	1	0	100101	001101	N/A	ori
17	slt	10	00	0	0	000	0	0	0	0	0	0	1	0	101010	000000	101010	slt
18	slti	00	00	0	0	000	0	0	0	0	0	1	1	0	101010	001010	N/A	slti
19	sll	10	00	0	0	000	0	0	0	0	0	0	1	0	000000	000000	000000	sll
20	srl	10	00	0	0	000	0	0	0	0	0	0	1	0	000010	000000	000010	srl
21	sra	00	00	0	0	000	0	0	0	0	0	0	1	0	000011	000000	000011	sra
22	sw	00	00	0	0	000	0	0	0	0	1	1	0	0	100000	101011	N/A	sw
23	beq	00	00	0	1	000	0	0	0	0	0	0	0	0	000000	000100	N/A	beq
24	bne	00	00	0	1	001	0	0	0	0	0	0	0	0	000000 (N/A)	000101	N/A	bne
25	j	00	00	1	0	000	0	0	0	0	0	0	0	0	000000 (N/A)	000010	N/A	j
26	jal	11	11	1	0	000	0	1	0	0	0	0	1	0	000000 (N/A)	000011	N/A	jal
27	jr	00	00	0	0	000	1	0	0	0	0	0	0	0	000000 (N/A)	000000	001000	jr
28	bgez	00	00	0	1	010	0	0	0	0	0	0	0	0	101010	000001	N/A	bgez
29	bgezal	11	00	0	1	010	0	0	0	0	0	0	1	1	101010	010000	N/A	bgezal
30	bgtz	00	00	0	1	011	0	0	0	0	0	0	0	0	101010	000111	N/A	bgtz
31	blez	00	00	0	1	100	0	0	0	0	0	0	0	0	101010	000110	N/A	blez
32	bltzal	11	00	0	1	101	0	0	0	0	0	0	1	1	101010	010010	N/A	bltzal
33	bltz	00	00	0	1	101	0	0	0	0	0	0	0	0	101010	010011	N/A	bltz

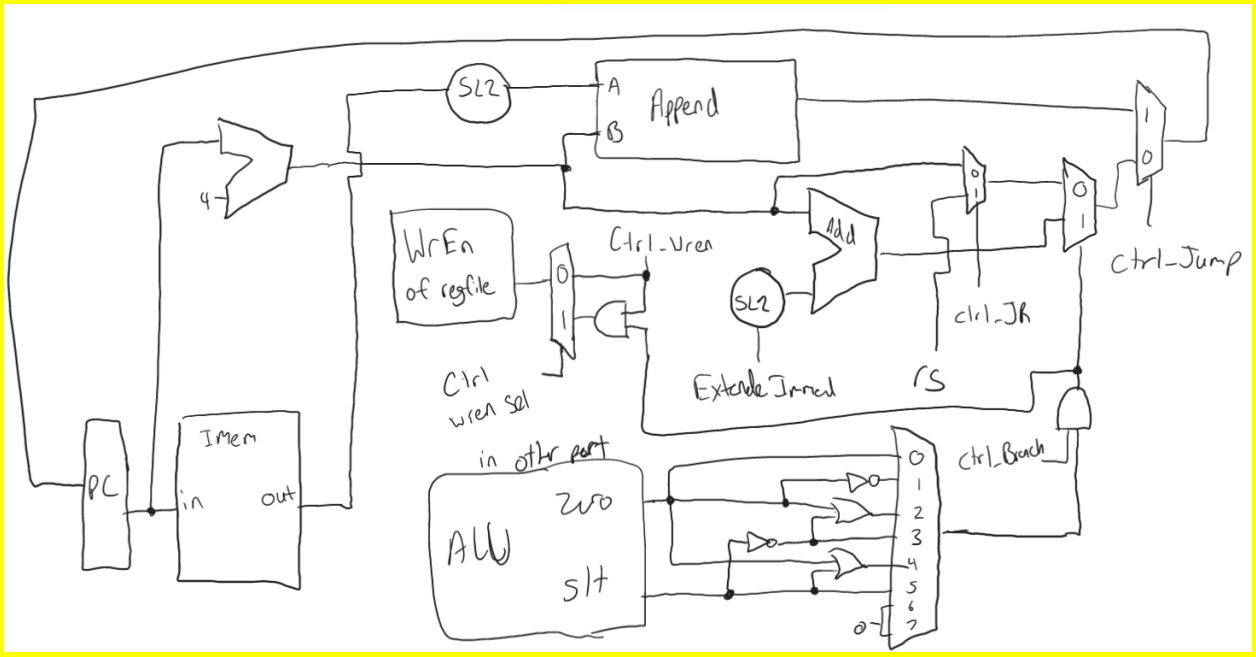
[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually and show that your output matches the expected control signals from problem 1(a).



[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.

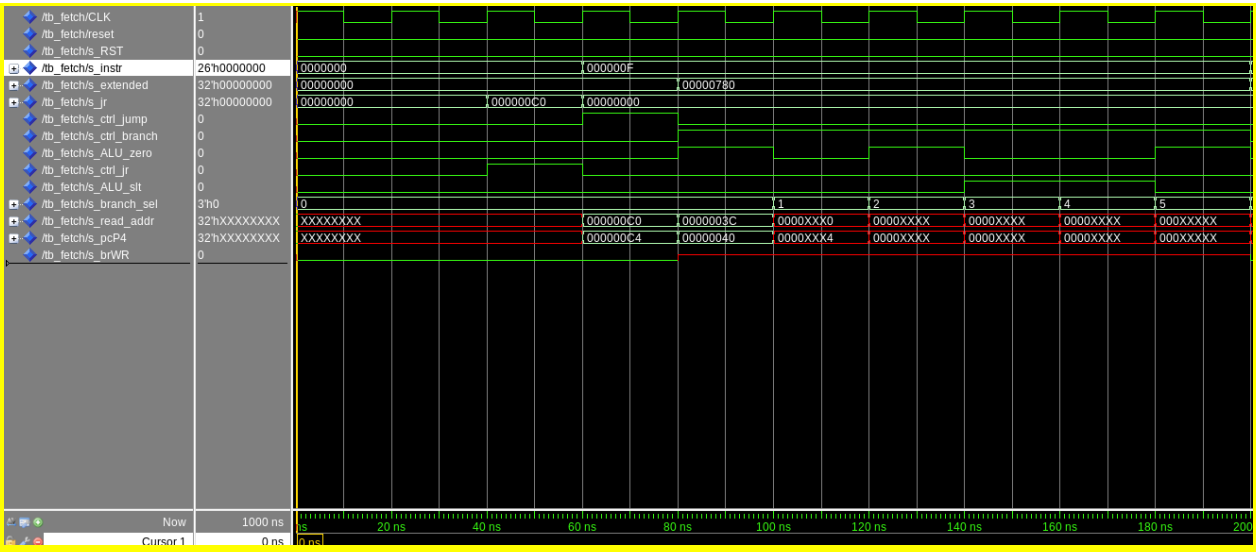
We have a bunch of bgez, bgtz type instructions to add and many of them have -al type variants. This can be achieved by receiving both zero (equal) and slt outs from the ALU and using various or gates and an 8to1 multiplexer to select between them. Combine this with the general branch logic and they should work. To get the and link part, use the logic behind jal and combine it with these branch instructions. Make sure to design a way to only enable the Write Enable of the regfile just incase the branch condition fails.

[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?



New control signals include a jr control, a branch type select, a control of the write enable selection and a new output from the ALU that is the slt. Other than that, it is essentially the same.

[Part 2 (b.iii)] 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.



The red failed outputs does appear to overlap with the instructions that have some failures. We are at the point where we need to move on and work on Project 2. Processor works for the most part and we are unsure why so much of this test bench fails, it is given valid input.

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

The srl shifts the values right, “dividing” the value by 2^however many bits it shifts. The sra does the same thing, but if the first bit of the data is 1, it will fill in 1s when the data is pushed right instead of zeros. This can be useful for negative numbers, by keeping the value negative. There is no sla instruction because a shift left is a shift left, and you do not need to keep track of it’s sign.

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

I have an arithmetic logical bit that is set to 1 for sra, but does nothing in my left shifter. By feeding a pre set bit in when a shift occurs, I can control what bit shifts in when shifting right and keep the sign of a number

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

By starting the shifter at bit 0 instead of 31, I can shift the bits starting at the opposite end and rolling over left instead of right

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

In the waveform above, there are two control bits: Shift_Right and Logical_ctl . Shift right decides which direction to shift: if it is set to 0, it will shift left. If it is 1, it will shift left. Logical control decides whether or not the data is arithmetically shifted if it is a right shift (it does nothing if it is a left shift).

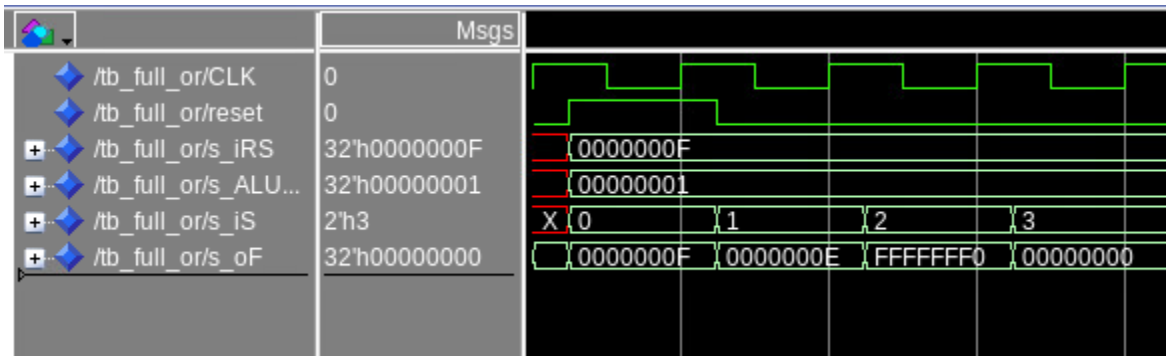
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

I decided to group similar components together for the sake of clarity. For example, I put the or, nor, and xor components together in a component I named “FULL OR”. This made it easier for me to direct data and choose which path I wanted to take, given instructions. The advantage of this decision was the organization of the project. A potential disadvantage to this strategy would be that it would be difficult to add on to, compared to a mux with many inputs, each with a separate component for a separate function. When finished, I had a full adder, a full shifter, an And component, and a Full or component.

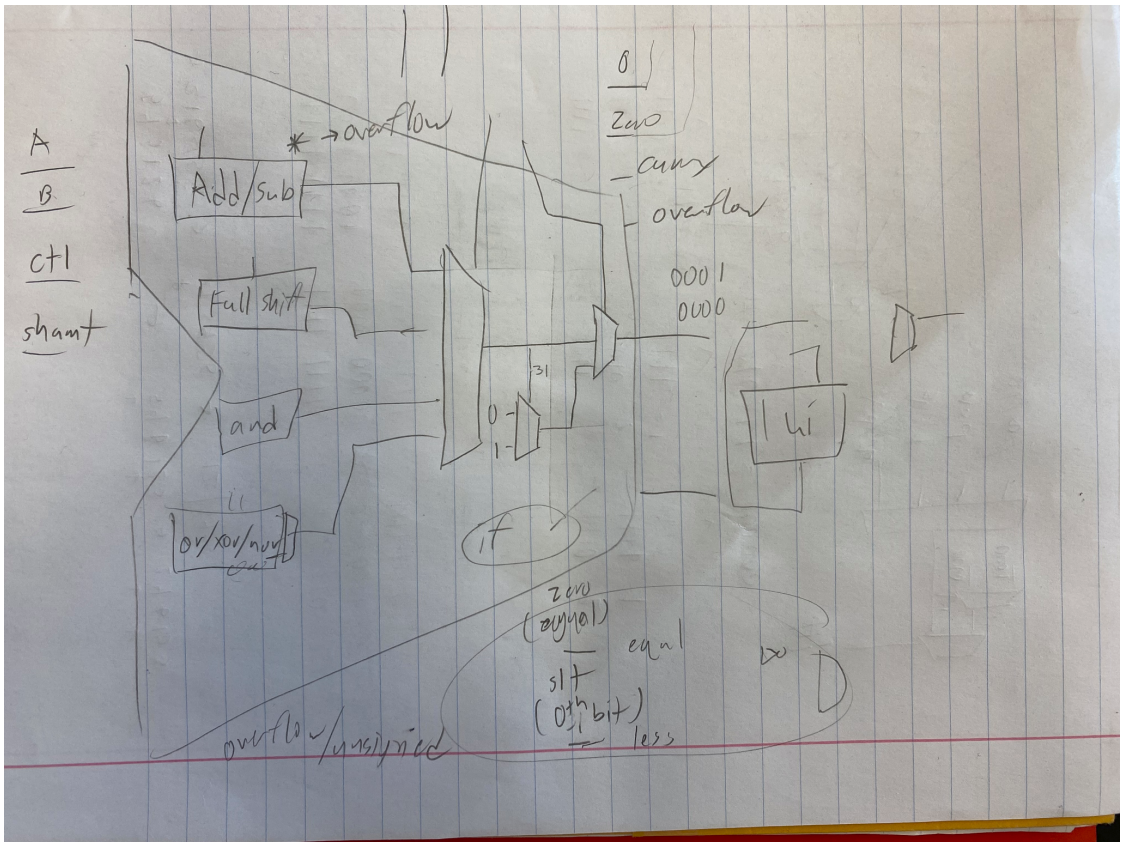
[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

```
-- Test case 1:
s_iS     <= "00";                -- 0 for OR
s_iRS    <= x"0000000F";
s_ALUmux <= x"00000001";
-- s_oF should be x"0000000F"
wait for gCLK_HPER*2;

-- Test case 2:
s_iS     <= "01";                -- 1 for XOR
s_iRS    <= x"0000000F";
s_ALUmux <= x"00000001";
-- s_oF should be x"0000000E"
wait for gCLK_HPER*2;

-- Test case 3:
s_iS     <= "10";                -- 2 for NOR
s_iRS    <= x"0000000F";
s_ALUmux <= x"00000001";
-- s_oF should be x"FFFFFFF0"
wait for gCLK_HPER*2;

-- Test case 4:
s_iS     <= "11";                -- 3 for Nothing
s_iRS    <= x"0000000F";
s_ALUmux <= x"00000001";
-- s_oF should be x"00000000"
wait for gCLK_HPER*2;
```

s_iS is my select bit which toggles between or, xor, and nor. Because there are 4 options in a 4 to 1 mux, the 4th is set to all zeros and should not be reachable.



[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?



The ALU will have 4 inputs:

- Input A: 32 bits (register RS)
- Input B: 32 bits (register RT or Immediate value)
- ALUctl: 6 bits (from the control unit to give it instructions)
- Shamt: 5 bits (bits 6-10 from instruction memory)

The ALU will have 4 outputs:

- Output Data: 32 bits (Value or address, depending on operation)
- Zero: 1 bit (for determining if input A and B are equal)

- Carry: 1 bit
- Overflow: 1 bit (for any possible math errors)

Overflow will be handled by the adder and by a gate to check if we are checking for overflow.

Zero is calculated by comparing the two inputs and returning 1 if they are equal.

Slt uses the sign of a resulting sub process to determine which value to output using the 2 to 1 muxes after the 4 to 1 mux.

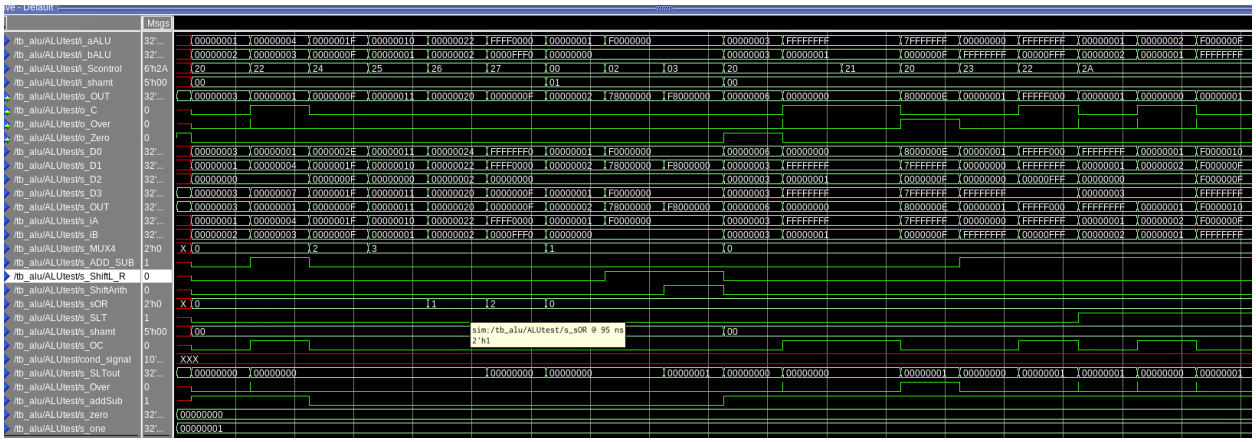
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

Include waveforms

I tested all functions in the ALU by providing the appropriate Opcode/function as the ALU control and then trying to do all required operations on the inputs. I would put expected signals in comments under the test cases so I could check the expected vs actual.

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

Include waveforms



```
-- Test case 1: add
s_i_aALU     <= x"00000001";
s_i_bALU     <= x"00000002";
s_i_Scontrol <= "100000";
s_i_shamt    <= "00000";
-- s_o_OUT should be x"00000003"
wait for gCLK_HPER*2;

-- Test case 2: sub
s_i_aALU     <= x"00000004";
s_i_bALU     <= x"00000003";
s_i_Scontrol <= "100010";
s_i_shamt    <= "00000";
-- s_o_OUT should be x"00000001"
wait for gCLK_HPER*2;

-- Test case 3: and
s_i_aALU     <= x"0000001F";
s_i_bALU     <= x"0000000F";
s_i_Scontrol <= "100100";
s_i_shamt    <= "00000";
-- s_o_OUT should be x"0000000F"
wait for gCLK_HPER*2;

-- Test case 4: or
s_i_aALU     <= x"00000010";
s_i_bALU     <= x"00000001";
s_i_Scontrol <= "100101";
s_i_shamt    <= "00000";
-- s_o_OUT should be x"00000011"
wait for gCLK_HPER*2;
```

For each instruction, we created a test and wrote the expected outcome before attempting the test. Then we looked at the ALU output to make sure the test was functioning as expected. For tests where the zero and carry bit were being tested, those were also listed in the expected outcome section.

The full list of tests can be found in proj/test/ALU. Keep in mind the components were more heavily tested prior to being included in the ALU, and those tests are also available.

```
-- OVERFLOW TESTING

-- Test case 12: addu
s_i_aALU     <= x"FFFFFFFF";
s_i_bALU     <= x"00000001";
s_i_Scontrol <= "100001";
s_i_shamt    <= "00000";
-- s_o_OUT should be x"00000000"
-- s_o_Zero should be 1
-- s_o_C should be 1
-- s_o_Over should be 1
wait for gCLK_HPER*2;

-- Test case 13: add
s_i_aALU     <= x"7FFFFFFF";
s_i_bALU     <= x"0000000F";
s_i_Scontrol <= "100000";
s_i_shamt    <= "00000";
-- s_o_OUT should be x"00000000"
-- s_o_Zero should be 1
-- s_o_C should be 1
-- s_o_Over should be 1
wait for gCLK_HPER*2;
```

[Part 3] 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.

Tested and passed:

All Add
All Addu
All Addi
All Addiu
All And
All Andi
All Beq
All Jal
All Jump
All lui
All nor
All or
All ori
All sub
All subu
All xor
All xori
All slt
All slti

All Jr
All bne
All sw
All blez
All sll
All srl
All sra
All lw

addiseq
Grendel
lab3Seq
Simplebranch
Proj1_base_test
Proj1_bubblesort

Failed tests:

Bgezal_1
Bgtz_4
Bltz_1, Bltz_2, Bltz_3
Bltzal_1, Bltzal_2, Bltzal_3, Bltzal_4

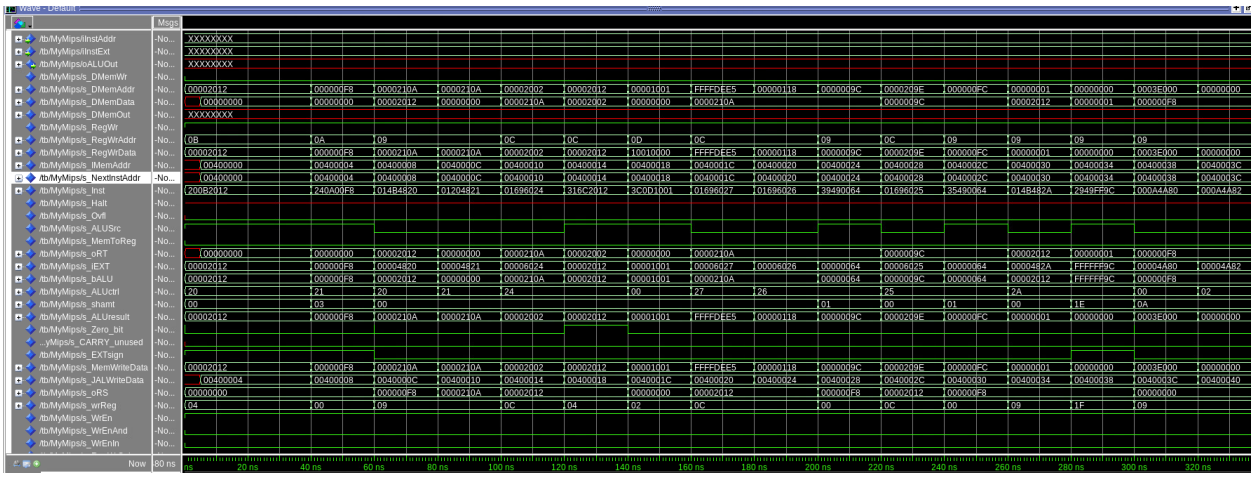
fibonacci
Proj1_cf_tests

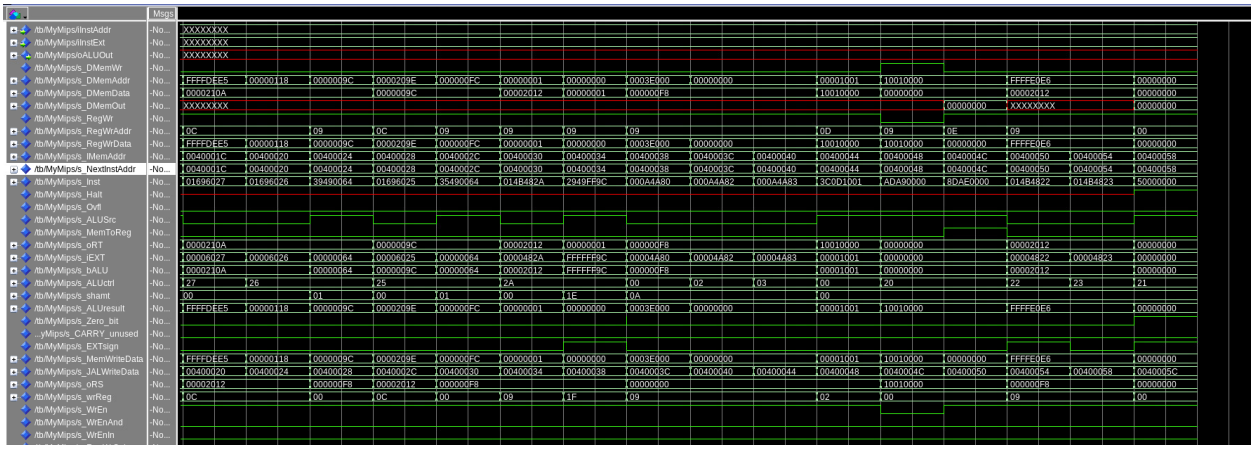
The opcodes are the same for some of the Branch instructions and we could not figure out how to get them running, but we had components designed and logic in place for handling them.

[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 and working. Passed all tests

```
bash-4.2$ ./381_tf.sh test proj/mips/Custom_Tests/Proj1_base_test*
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using $MGC_HOME=/usr/local/mentor/calibre
All VHDL src files compiled successfully
Testing file: proj/mips/Custom_Tests/Proj1_base_test.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 23
Processor Cycles: 23
CPI:   1.0
Results in: output/Proj1_base_test.s
```

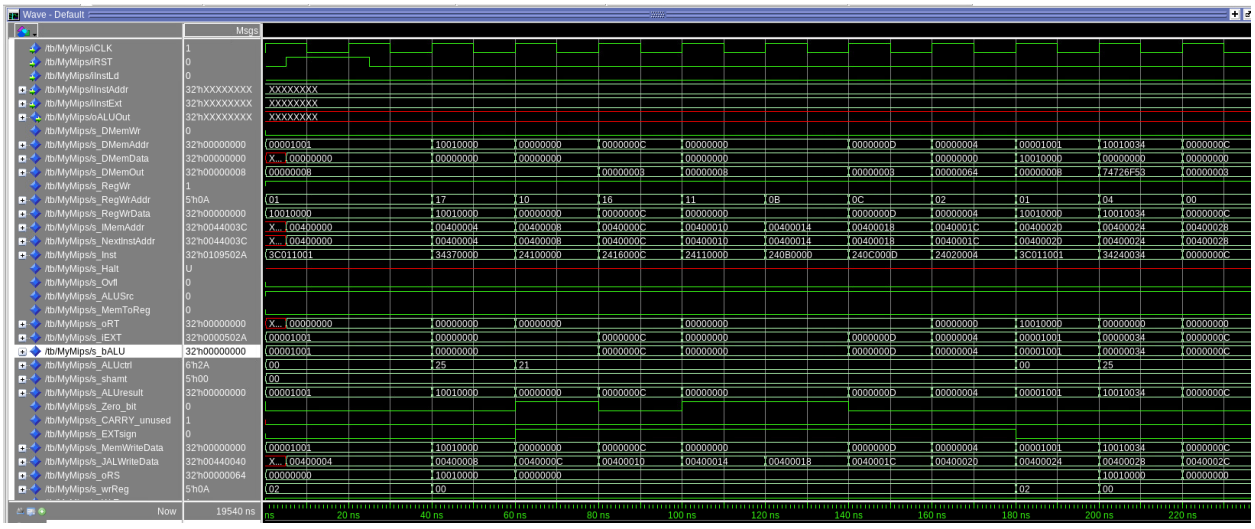




[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

[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.

```
bash-4.2$ ./381_tf.sh test proj/mips/Custom_Tests/Proj1_bubblesort*
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using $MGC_HOME=/usr/local/m
entor/calibre
All VHDL src files compiled successfully
Testing file: proj/mips/Custom_Tests/Proj1_bubblesort.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 976
Processor Cycles: 976
CPI:   1.0
Results in: output/Proj1_bubblesort.s
```



I am only including the first 10 or so instructions for obvious reasons.

[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 schematic from part 1. What components would you focus on to improve the frequency?

```
FMax: 25.53mhz Clk Constraint: 20.00ns Slack: -19.17ns
```

The components I would focus on to improve speed is the ripple carry adders and shifters. There are faster adders that would cut down on the ALU delay which is a big contributor to our current critical path.