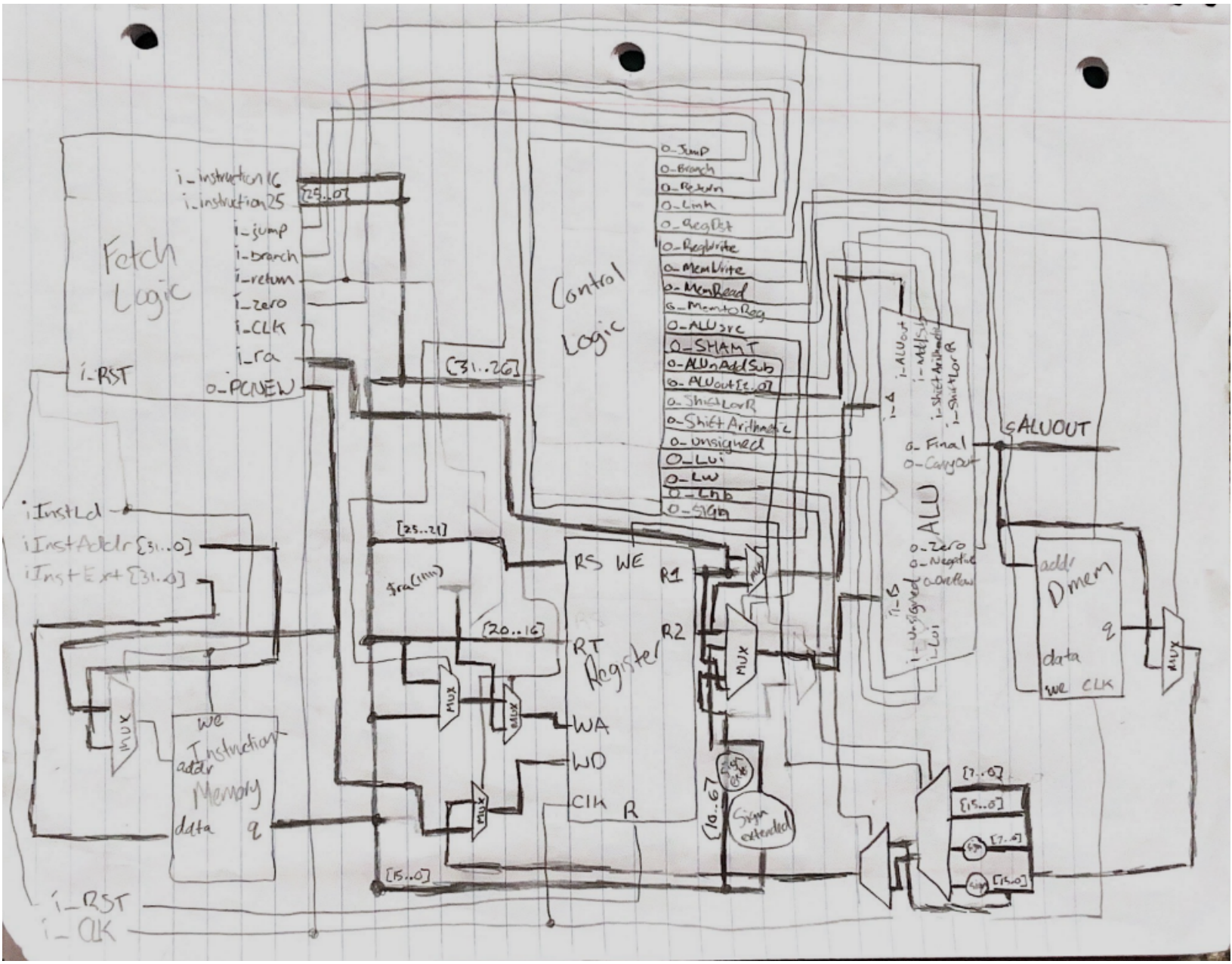
CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: Cameron Gilbertson
Mason Kotlarz

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.

Instruction	Opcode	Funct	Register and Memory operations						Load Byte and Load Word			PC fetch logic						ALU Operations						
			RegDst	RegWrite	ALUsrc	MemWrite	MemRead	MemToReg	HoB	lw	sign	Branch	Branchne	Return	Link	Jump	Halt	ALUnaddsub	ALUout	Shift Left/Right	Shift Arithmetic	SHAMT	Unsigned	Lui
add	000000	100000	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	010	0	0	0	1	0
addi	001000		0 [I Type]	1	1 [I Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	010	0	0	0	1	0
addiu	001001		0 [I Type]	1	1 [I Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	010	0	0	0	0 [overflow and]	0
addu	000000	100001	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	010	0	0	0	0 [overflow and]	0
and	000000	100100	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	000	0	0	0	0	0
andi	001100		0 [I Type]	1	1 [I Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	000	0	0	0	0	0
lui	001111		0 [I Type]	1	1	0	0	0 [doesnt read]	0	0	0	0	0	0	0	0	0	0	010	0	0	0	0	1
lw	100011		0 [I Type]	1	1 [I Type]	0	1 [Load Operation]	1	0	0	0	0	0	0	0	0	0	..	010	0	0	0	0	0
nor	000000	100111	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	101	0	0	0	0	0
xor	000000	100110	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0
xori	001110		0 [I Type]	1	1 [I Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0
or	000000	100101	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	000	0	0	0	0	0
ori	001101		0 [I Type]	1	1 [I Type]	0	0	0	0	0	0	0	0	0	0	0	0	0	000	0	0	0	0	0
slt	000000	101010	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	1 [carryIn]	111	0	0	0	0	0
slti	001010		0 [I Type]	1	1 [I Type]	0	0	0	0	0	0	0	0	0	0	0	0	1 [carryIn]	111	0	0	0	0	0
sll	000000	000000	1 [R Type]	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	110	1 [1 == left]	0	1	0	0
srl	000000	000010	1 [R Type]	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	110	0 [0 == right]	0	1	0	0
sra	000000	000011	1 [R Type]	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	110	0 [0 == right]	1	1	0	0
sw	101011		x	0	1 [I Type]	1 [Store Operation]	0	x	0	0	0	0	0	0	0	0	0	0	010	0	0	0	0	0
sub	000000	100010	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	1 [carryIn]	011	0	0	0	1	0
subu	000000	100011	1 [R Type]	1	0 [R Type]	0	0	0	0	0	0	0	0	0	0	0	0	1 [carryIn]	011	0	0	0	0 [overflow and]	0
beq	000100		x	0	0 [J Type]	0	0	x	0	0	0	1	0	0	0	0	0	1 [carryIn]	011	0	0	0	0	0
bne	000101		x	0	0 [J Type]	0	0	x	0	0	0	0	1	0	0	0	0	1 [carryIn]	011	0	0	0	0	0
j	000010		x	0	x [J Type]	0	0	x	0	0	0	0	0	0	0	1	0	x	x	x	x	0	x	x
jal	000011		x	1	0	0	0	x	0	0	0	0	0	0	1	1	0	x	x	x	x	0	x	x
jr	000000	001000	x	0	x [J Type]	0	0	x	0	0	0	0	0	1	0	1	0	x	x	x	x	0	x	x
lb	100000		0 [I Type]	1	1 [I Type]	0	1 [Load Operation]	1	0	1	1	0	0	0	0	0	0	0	010	0	0	0	0	0
lh	100001		0 [I Type]	1	1 [I Type]	0	1 [Load Operation]	1	1	1	1	0	0	0	0	0	0	0	010	0	0	0	0	0
lbu	100100		0 [I Type]	1	1 [I Type]	0	1 [Load Operation]	1	0	1	0	0	0	0	0	0	0	0	010	0	0	0	0	0
lhu	100101		0 [I Type]	1	1 [I Type]	0	1 [Load Operation]	1	1	1	0	0	0	0	0	0	0	0	010	0	0	0	0	0
sllv	000000	000100	1 [R Type]	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	110	1 [1 == left]	0	1	0	0
srlv	000000	000110	1 [R Type]	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	110	0 [0 == right]	0	1	0	0
srav	000000	000111	1 [R Type]	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	110	0 [0 == right]	1	1	0	0
halt	010100		0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

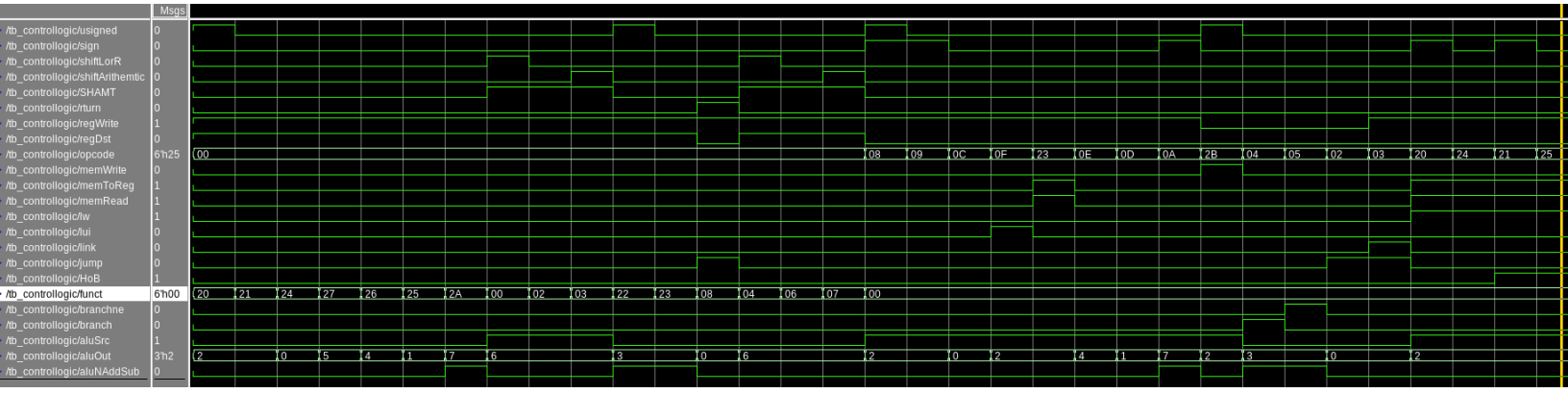
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF
Load Byte and Load Word			PC fetch logic						ALU Operations								Working					
HoB	lw	sign	Branch	Branchne	Return	Link	Jump	Halt	ALUnaddsub	ALUout	Shift Left/Right	Shift Arithmetic	SHAMT	Unsigned	Lui							
0	0	0	0	0	0	0	0	0	0	010	0	0	0	1	0	add						
0	0	0	0	0	0	0	0	0	0	010	0	0	0	1	0	add						
0	0	0	0	0	0	0	0	0	0	010	0	0	0	0 [overflow and]	0	add						
0	0	0	0	0	0	0	0	0	0	010	0	0	0	0 [overflow and]	0	add						
0	0	0	0	0	0	0	0	0	0	000	0	0	0	0	0	and						
0	0	0	0	0	0	0	0	0	0	000	0	0	0	0	0	and				ALUout		o_Final
0	0	0	0	0	0	0	0	0	0	010	0	0	0	0	1	add			c2	c1	c0	Operation
0	0	0	0	0	0	0	0	0	..	010	0	0	0	0	0	add			0	0	0	A AND B
0	0	0	0	0	0	0	0	0	0	101	0	0	0	0	0	nor			0	0	1	A OR B
0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	xor			0	1	0	A + B
0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	xor			0	1	1	A - B
0	0	0	0	0	0	0	0	0	0	000	0	0	0	0	0	or			1	0	0	A XOR B
0	0	0	0	0	0	0	0	0	0	000	0	0	0	0	0	or			1	0	1	A NOR B
0	0	0	0	0	0	0	0	0	1 [carryIn]	111	0	0	0	0	0	slt			1	1	0	Shift
0	0	0	0	0	0	0	0	0	1 [carryIn]	111	0	0	0	0	0	slt			1	1	1	SLT
0	0	0	0	0	0	0	0	0	0	110	1 [1 == left]	0	1	0	0	sll						
0	0	0	0	0	0	0	0	0	0	110	0 [0 == right]	0	1	0	0	srl						
0	0	0	0	0	0	0	0	0	0	110	0 [0 == right]	1	1	0	0	sra						
0	0	0	0	0	0	0	0	0	0	010	0	0	0	0	0	add						
0	0	0	0	0	0	0	0	0	1 [carryIn]	011	0	0	0	1	0	sub						
0	0	0	0	0	0	0	0	0	1 [carryIn]	011	0	0	0	0 [overflow and]	0	sub						
0	0	0	1	0	0	0	0	0	1 [carryIn]	011	0	0	0	0	0	sub						
0	0	0	0	1	0	0	0	0	1 [carryIn]	011	0	0	0	0	0	sub						
0	0	0	0	0	0	0	1	0	x	x	x	x	0	x	x	x						
0	0	0	0	0	0	1	1	0	x	x	x	x	0	x	x	x						
0	0	0	0	0	1	0	1	0	x	x	x	x	0	x	x	x						
0	1	1	0	0	0	0	0	0	0	010	0	0	0	0	0	add						
1	1	1	0	0	0	0	0	0	0	010	0	0	0	0	0	add						
0	1	0	0	0	0	0	0	0	0	010	0	0	0	0	0	add						
1	1	0	0	0	0	0	0	0	0	010	0	0	0	0	0	add						
0	0	0	0	0	0	0	0	0	0	110	1 [1 == left]	0	1	0	0	sll						
0	0	0	0	0	0	0	0	0	0	110	0 [0 == right]	0	1	0	0	srl						
0	0	0	0	0	0	0	0	0	0	110	0 [0 == right]	1	1	0	0	sra						
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	x						

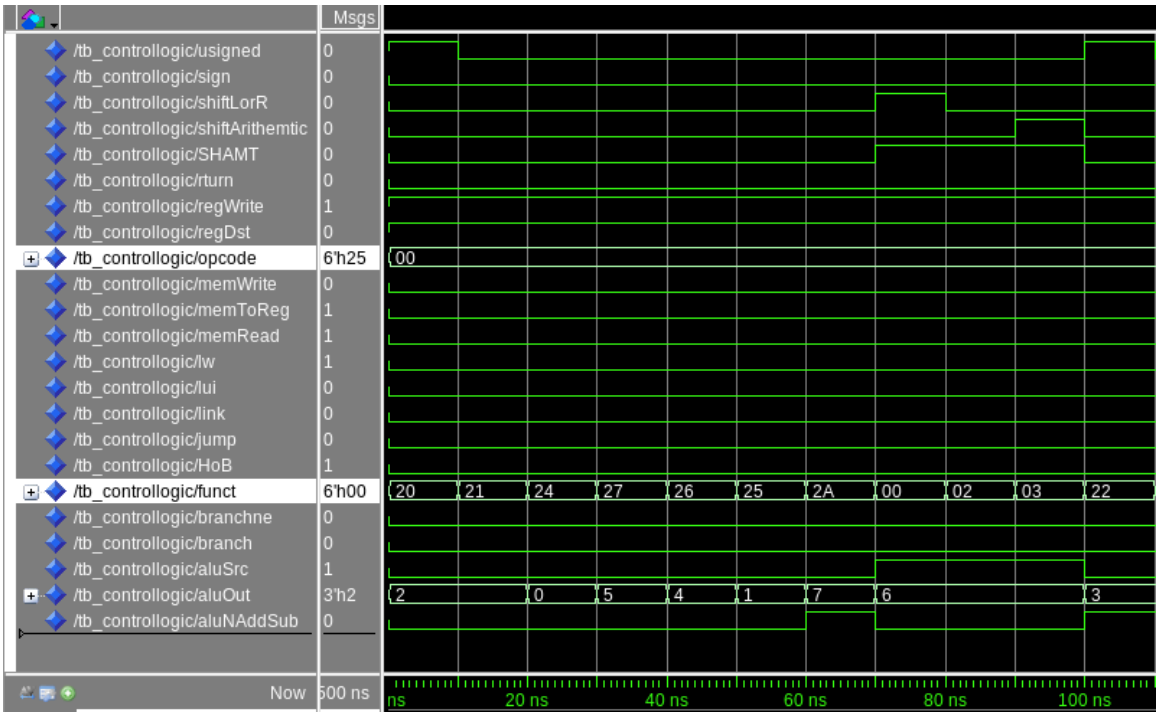
See attached google spreadsheet.

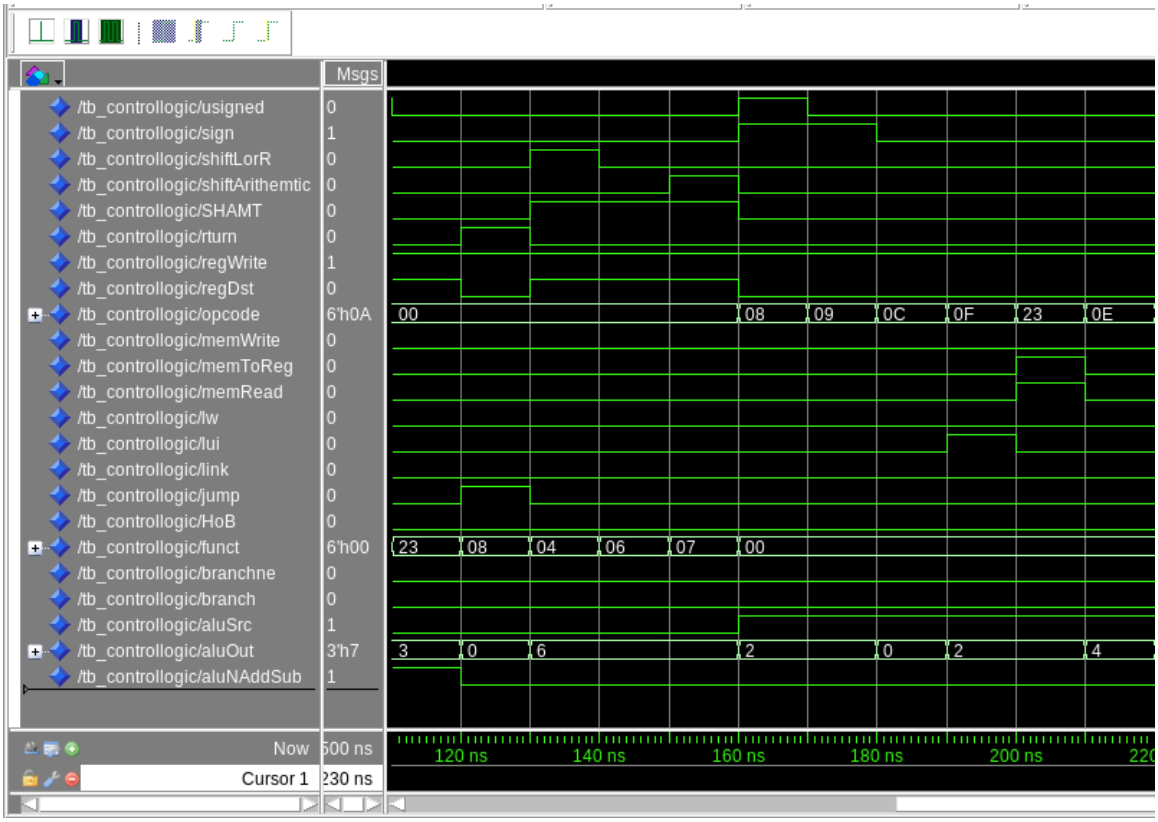
[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually and show that your output matches the expected control signals from problem 1(a).

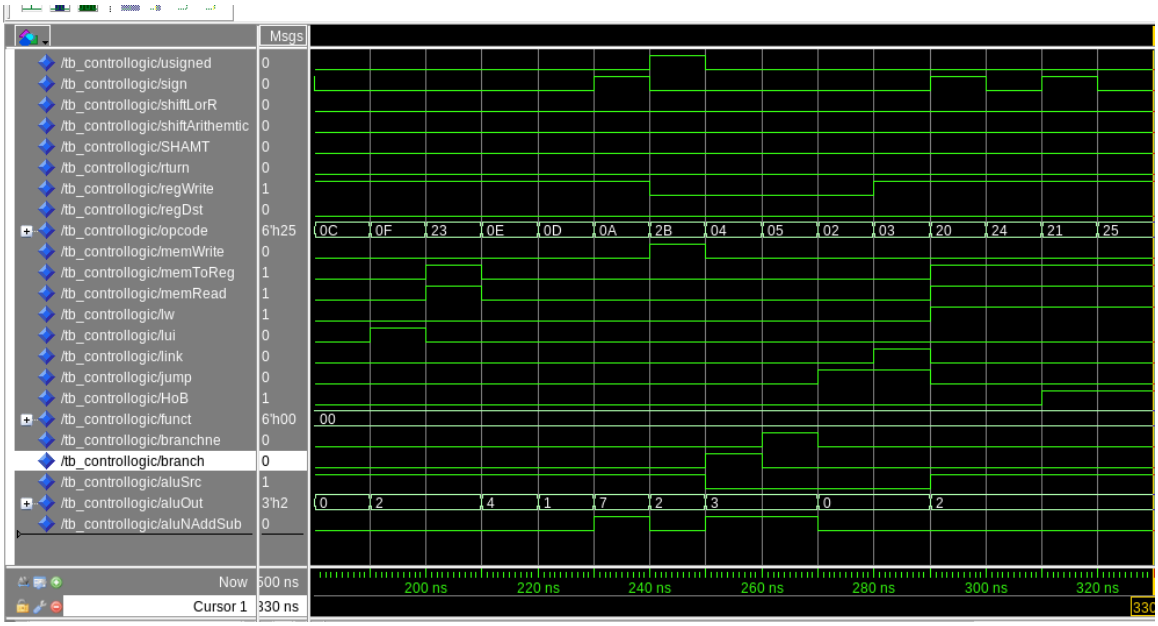
The first screenshot is of the whole waveform testbench overall. while the other 3 are closer up screenshots moving to the left of the waveform. As you can see we simulated every R type instruction first, then every I type instruction. You can also see this in our code. This waveform is correct and accurate because it reflects the Control Logic SpreedSheet.

(p.s. I recommend you use the ns as a benchmark for helping you navigate between reading the screenshots… I can read the waveform, however, if you can't then let me know I will submit the screenshot differently)





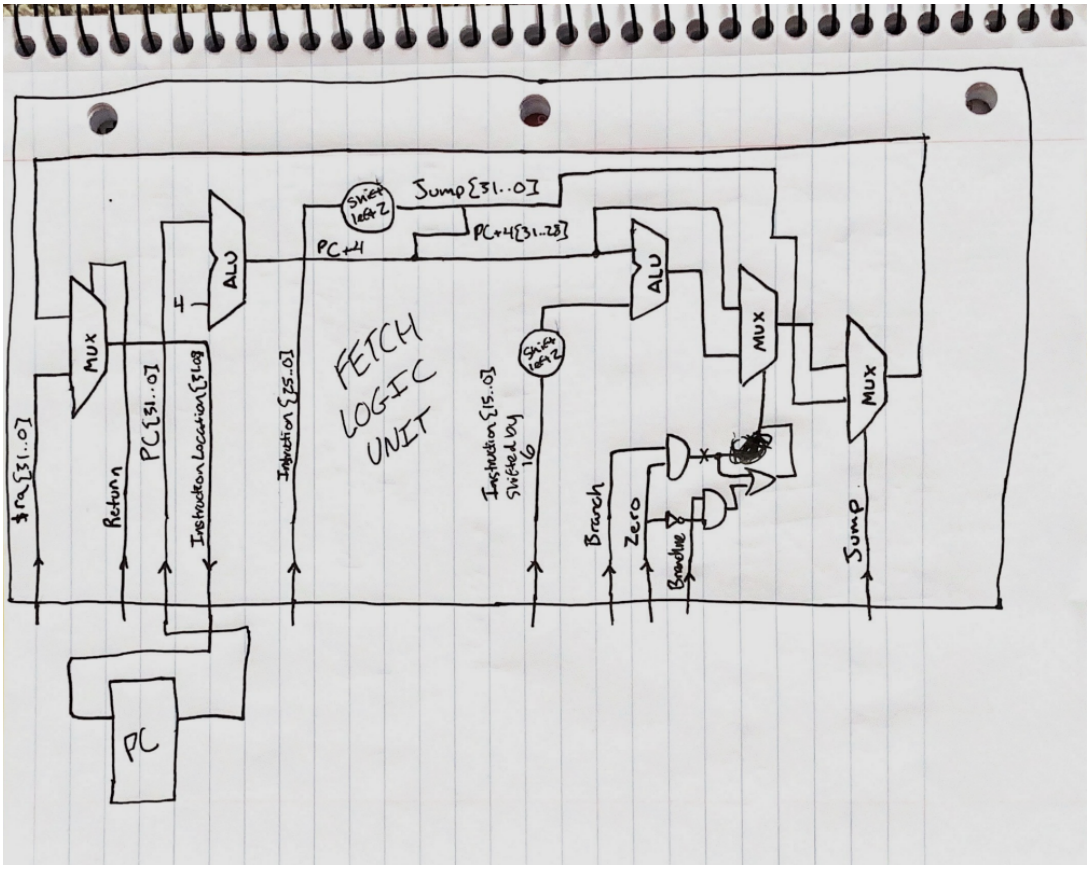

Msgs
/tb_controllogic/usigned
/tb_controllogic/sign
/tb_controllogic/shiftLorR
/tb_controllogic/shiftArithemtic
/tb_controllogic/SHAMT
/tb_controllogic/rturn
/tb_controllogic/regWrite
/tb_controllogic/regDst
/tb_controllogic/opcode
/tb_controllogic/memWrite
/tb_controllogic/memToReg
/tb_controllogic/memRead
/tb_controllogic/lw
/tb_controllogic/lui
/tb_controllogic/link
/tb_controllogic/jump
/tb_controllogic/HoB
/tb_controllogic/funct
/tb_controllogic/branchne
/tb_controllogic/branch
/tb_controllogic/aluSrc
/tb_controllogic/aluOut
/tb_controllogic/aluNAddSub
0
1
0
0
0
0
1
0
6'h0A
0
0
0
0
0
0
0
6'h00
0
0
1
3'h7
1
00
08
09
0C
0F
23
0E
23
08
04
06
07
00
3
0
6
2
0
2
4
Now
Cursor 1
120 ns
140 ns
160 ns
180 ns
200 ns


Msgs
/tb_controllogic/usigned
/tb_controllogic/sign
/tb_controllogic/shiftLorR
/tb_controllogic/shiftArithemtic
/tb_controllogic/SHAMT
/tb_controllogic/rturn
/tb_controllogic/regWrite
/tb_controllogic/regDst
/tb_controllogic/opcode
/tb_controllogic/memWrite
/tb_controllogic/memToReg
/tb_controllogic/memRead
/tb_controllogic/lw
/tb_controllogic/lui
/tb_controllogic/link
/tb_controllogic/jump
/tb_controllogic/HoB
/tb_controllogic/funct
/tb_controllogic/branchne
/tb_controllogic/branch
/tb_controllogic/aluSrc
/tb_controllogic/aluOut
/tb_controllogic/aluNAddSub
0
0
0
0
0
0
1
0
6'h25
0
1
1
1
0
0
0
1
6'h00
0
0
1
3'h2
0
0C
0F
23
0E
0D
0A
2B
04
05
02
03
20
24
21
25
00
0
2
4
1
7
2
3
0
2
Now
Cursor 1
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns
320 ns

[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 unit must be able to do all the functionality of the program counter and other logic related to the program counter. There are plenty of instructions that need to be implemented using control signals. Using the fetch logic unit the processor gains (j, jal, jr, beq, and bne). All of these functions are implemented in the fetch logic and controlled by the control logic.

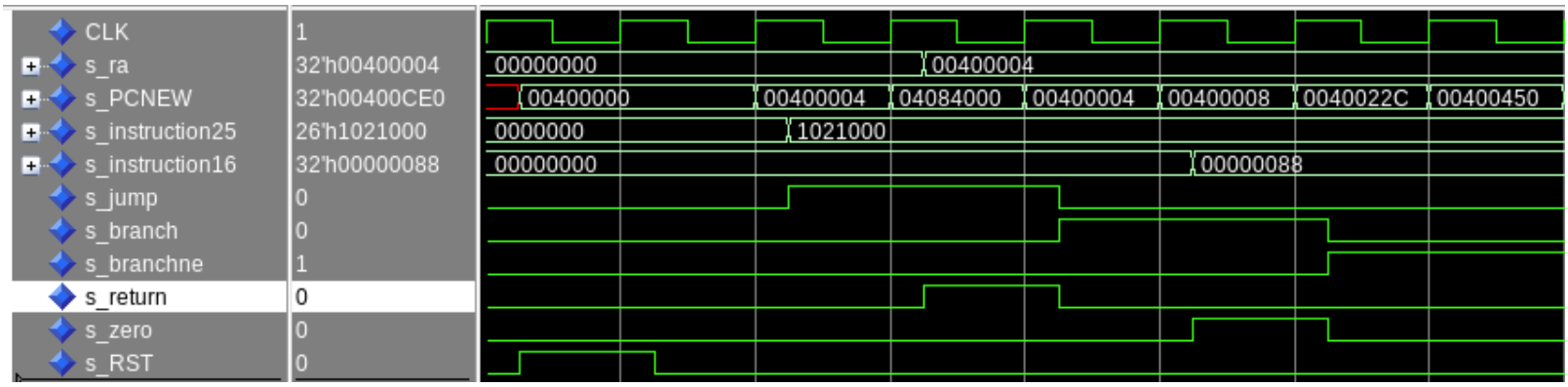
[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?



All the control signals we will need included (s_Return for jr, s_Branch for beq, s_Branchne for bne, s_Zero for beq and bne, and s_Jump for j and jal). Using these control signals and some digital logic the fetch logic works and includes the program counter.

[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.



These are the waveforms for our fetch logic unit. In this test benched we test every instruction the fetch would be responsible for. The instructions we tested here included (j, jal, jr, beq, bne, and general PC+4).

In the first test, we made sure that the PC reset to 0x00400000 which is important because that is the starting address for the instructions in MARS. Our expectations matched our waveforms for this part.

In the second test, we tested the general PC+4 case which is the most common use for the fetch logic unit. Here our waveforms increased from 0x00400000 => 0x00400004, which is what we expect.

In the third test, we tested the jump instruction, which is a J-Type instruction and therefore uses the 25..0 bits of the instruction. So in our fetch logic, that is the input i_instruction25 which takes in the 25..0 bits of the instruction from the instruction memory. So here our output matched our expected.

In the fourth test, we tested jump return instruction which sets the PC to a value from a register. In our fetch logic unit, the input for this is s_ra we chose this name because commonly, jr is used with $ra, which is the return register. Here our waveforms matched our expectations.

In the fifth and sixth tests, we tested the beq instruction, which is going to set PC to an immediate value when the two registers are equal. So the inputs used for the fetch logic are the i_branch and i_zero, so these two must be 1 for the branch to occur. In the fifth test, the i_zero is not 1 so the branch does not occur. But in the sixth case, i_zero is 1, so the branch occurs.

In the seventh and final test, we tested the bne instruction, which is going to set PC to an immediate value when the two registers are not equal. Overall this instruction works the same as beq but inverted. Our output matched our expected for this case as well.

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

Shift Right Logical (srl) puts zeros into the empty spaces once the shift has happened. Shift Right Arithmetic (sra) puts the sign bit into those empty spaces. So either 0 or 1 depending on whether or not the most significant bit is 0 or 1. The reason that there is not a (sla) is we should never put 1's into the least significant bits because even if the number is signed, adding 1's in this location would not preserve the sign of the number.

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

Our implementation uses 32 layers of 32 2:1 muxes. This is for the right barrel shifter. We use a special decoder that converts by SHAMT value. Then these SHAMT values are put into the select lines of the MUXs. In order to have the functionality of shift right arithmetic, we used an AND gate with the signed input and the most significant bit of the incoming data.

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

In order to support left shifting operations we are going to need another input control line in order to distinguish left and right operations. So we are also going to need a larger mux. We are going to use 32 layers of 32 4:1 muxes instead of 2:1. We will have the fourth input which we do not use attached to ground.

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

In this testbench, we tested all the different shift operations that our processor would need to complete. There were about 10 test cases in total with each testing a different shifting instruction.
So the first two shifts were normal srl. Those performed as expected with the correct shift outputs. The next two tests were sra. The first sra was with a 1 in the most significant bit. This caused the newly shifted bits to be 1 as expected. The second sra was with a 0 in the most significant bit. This caused the newly shifted bits to be 0 as expected. The next two

were more normal srl which both worked as expected. Then there were the last 4 test cases were all sll operations. The output matched our expected output.

[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.

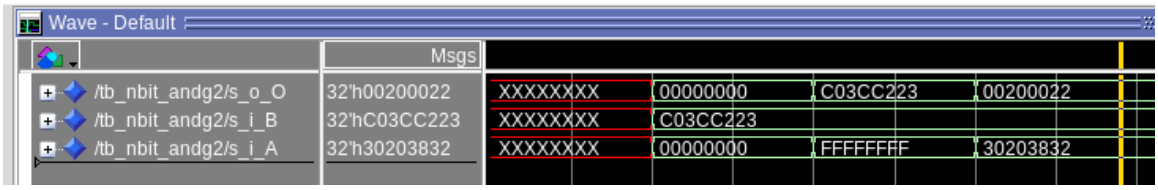
I first went through every single one of our instructions then classified them each as a certain type of operation the ALU needs to perform, so if you look at my Control Logic spreadsheet you can look in the z column and see how I classified every instruction… So for example an add instruction is simply an ADD (010) type, while a branch instruction is a SUB (011) type A Load and Store instruction is an ADD (010) type… The Jump instructions are not ALU type instructions you would think they would be add type however the addition is performed in the Fetch Logic. This is my Logic for the 8 to 1 mux.

c2	c1	c0	Operation
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A + B
0	1	1	A - B
1	0	0	A XOR B
1	0	1	A NOR B
1	1	0	Shift
1	1	1	SLT

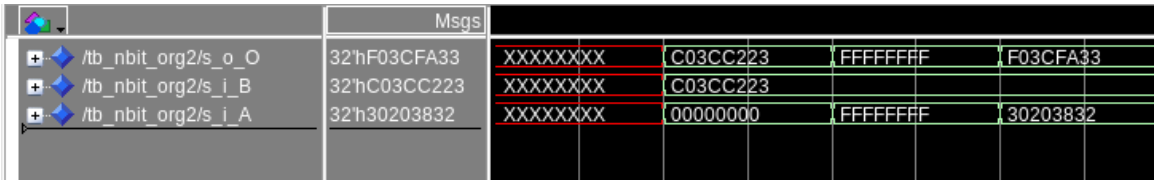
Then I needed to create the inputs for the 8 to 1 mux; this was rather simple for the addition, subtraction, and, or, xor, nor operations. The barrel shifter was a tiny bit more complex, it simply plugin the correct signals and the A and B values correctly… One of the other design decisions I made was the Negative checker for the SLT type… the input of the negative checker is the output of the adder/subtractor, then it outputs either 0x1 or 0x0 depending on whether the input is negative or not. The last design decision I made was the Lui operation. I could've verywell added the functionality for lb lh lbu lhu, and in hindsight I would change it. But to expand the LUI part, we shift the output of the 8 to 1 mux (because a LUI is an add operation) then select whether we want the shifted 16 bit value from the 2 to 1 mux.

For any additional functional units you design, go through the design process for each component [Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

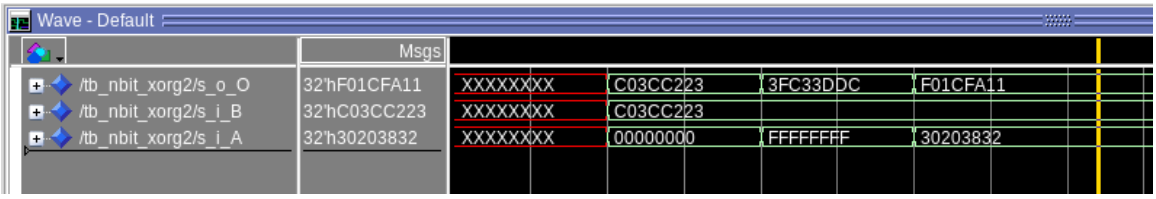
Below is my Waveform testing my Nbit_Andg2



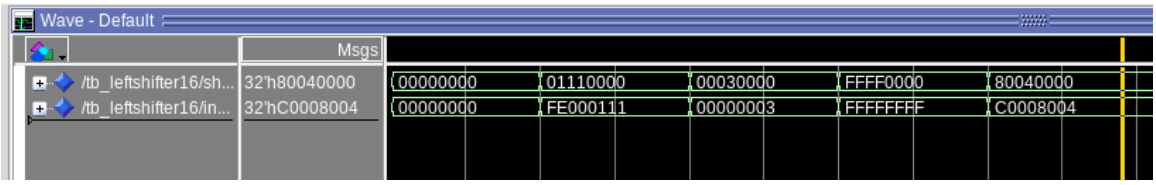
Below is my Waveform testing my Nbit_Org2



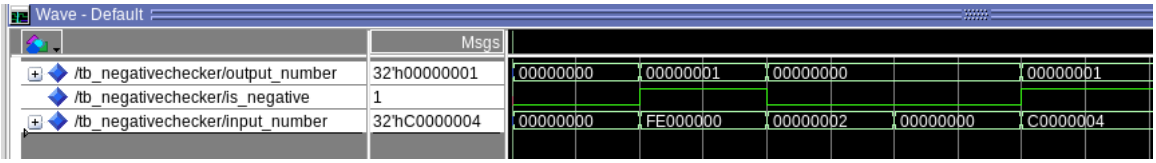
Below is my Waveform testing my Nbit_XORg2



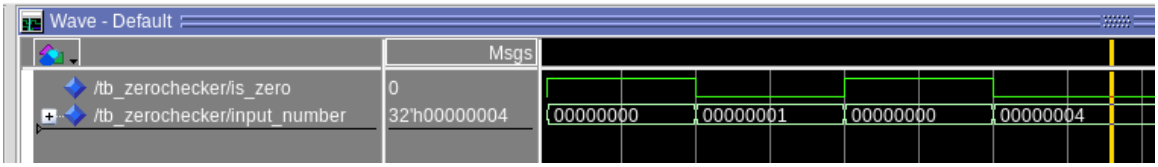
Below is my Waveform testing my Nbit_leftshift16



Below is my Waveform testing my Negativecheker

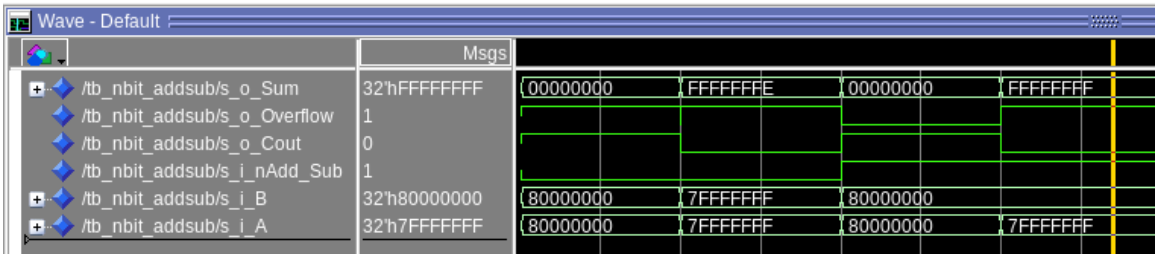


Below is my Waveform testing my ZeroChecker



Below is my Waveform testing my Nbit_FullAdder

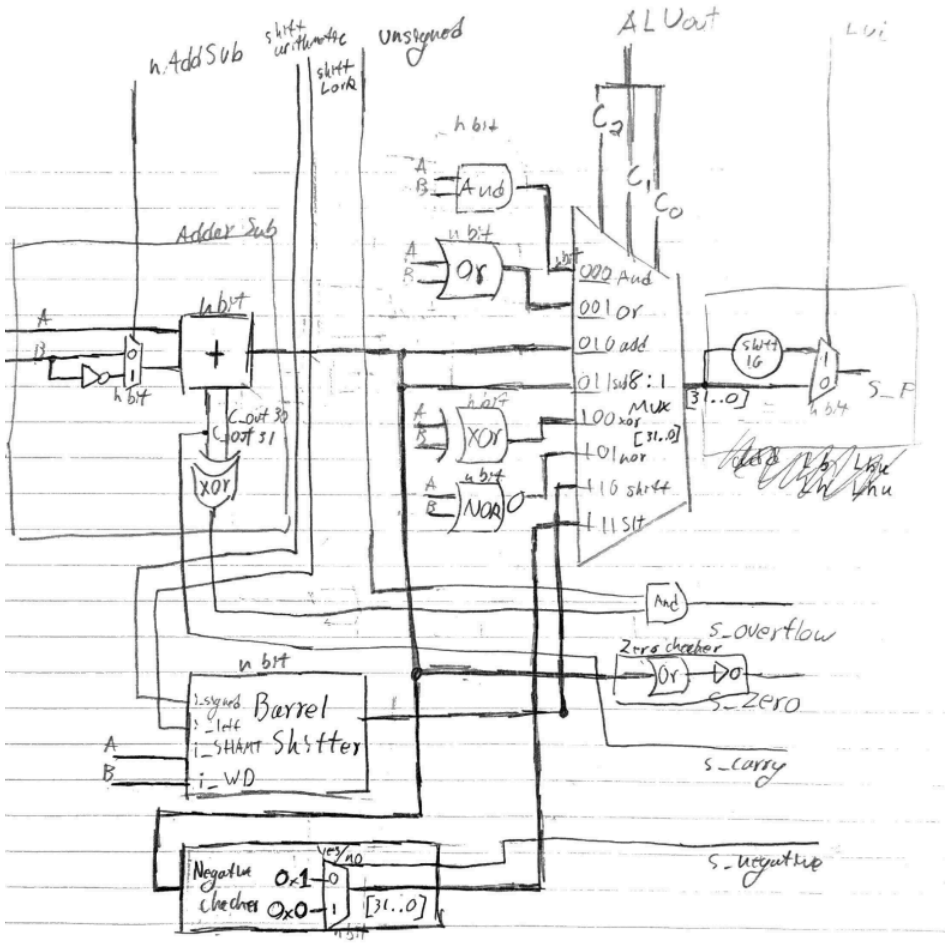
Below is my Waveform testing my Nbit_AdderSubtractor



[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?

Zero is calculated by taking the subtraction operation of the whole Subtraction operation and then performing an OR operation on every bit… if any bit is true 1 then the s_zero will end up being 0... however if any bit is 0 then the output of the s_zero will be 1.

For Slt we first perform a subtraction operation, and then we perform a negative check… essentially if the first bit of the output is equal to 1 then it is negative however if it is 0 then it is positive. If the output is negative then that means the slt operation a < b is true therefore we need to set the output of ALU to 0x00000001… However, if the output of the negative checker is false then we set the output of the ALU to 0x00000000.



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

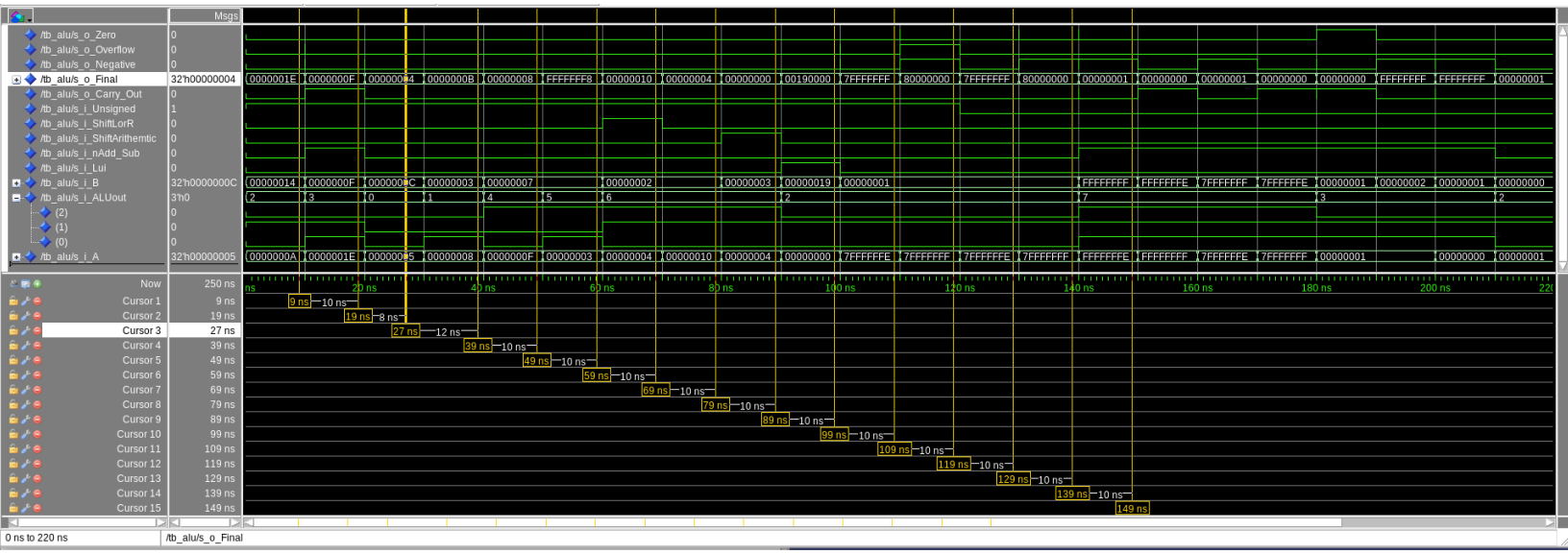
This is a screenshot of my overall ALU waveform. Here we have several tests, one for each type of instruction carried out by the ALU (o_Final), and then testing the outputs of each of the different signal lines for the s_zero, s_overflow, and s_carryout…

The first test is the ADD operation
The second test is the SUB operation
The 3rd test is AND
4th test is OR
5th is XOR
6th is NOR
7th is SLL (shift)
8th is SRL (shift)
9th is SRA (shift)
10th is LUI (add)
11th is ADDU (add)
12th is ADDU (add)
13th is ADDU (add) (unsigned input)
14th is ADDU (add) (unsigned input)
15th is SLT (sub)
16th is SLT (sub)
17th is SLT (sub) (unsigned input)
18th is SLT (sub) (unsigned input)
the rest are zero output checking (sub)

c2	c1	c0	Operation
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A + B
0	1	1	A - B
1	0	0	A XOR B
1	0	1	A NOR B
1	1	0	Shift
1	1	1	SLT

I will walk you through the first ALU ADD test for an example:

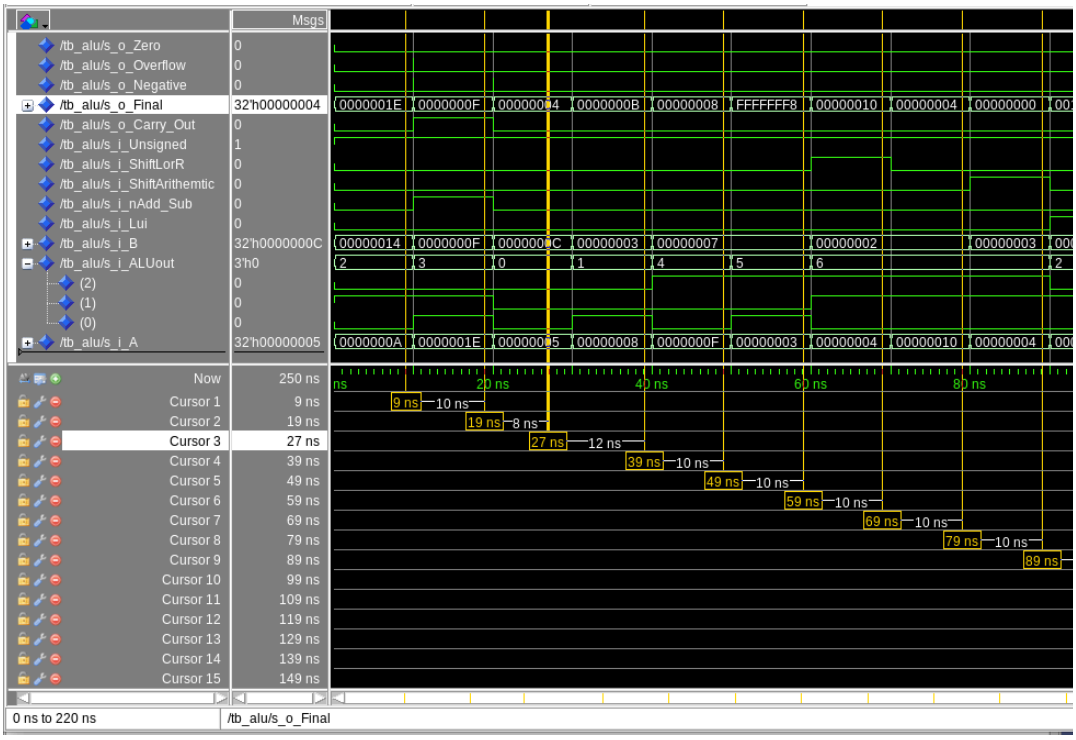
As you can see we have the s_i_ALUout set to 010 meaning the ADD output of the 8_1 mux will be selected. And we have the s_i_A and s_i_B set to different numbers A and 14 respectively, then we add the 2 numbers together and get the value 1E… this translated to base 10 (decimal) is 10 + 20 = 30 this is correct and expected… s_i_Unsigned is set to 1 to correctly do the and gate for the overflow checking if the value of the Unsigned is 0 then we don't care about overflow. In this specific case we care about overflow and the s_o_overflow value is 0 because no overflow occurred.

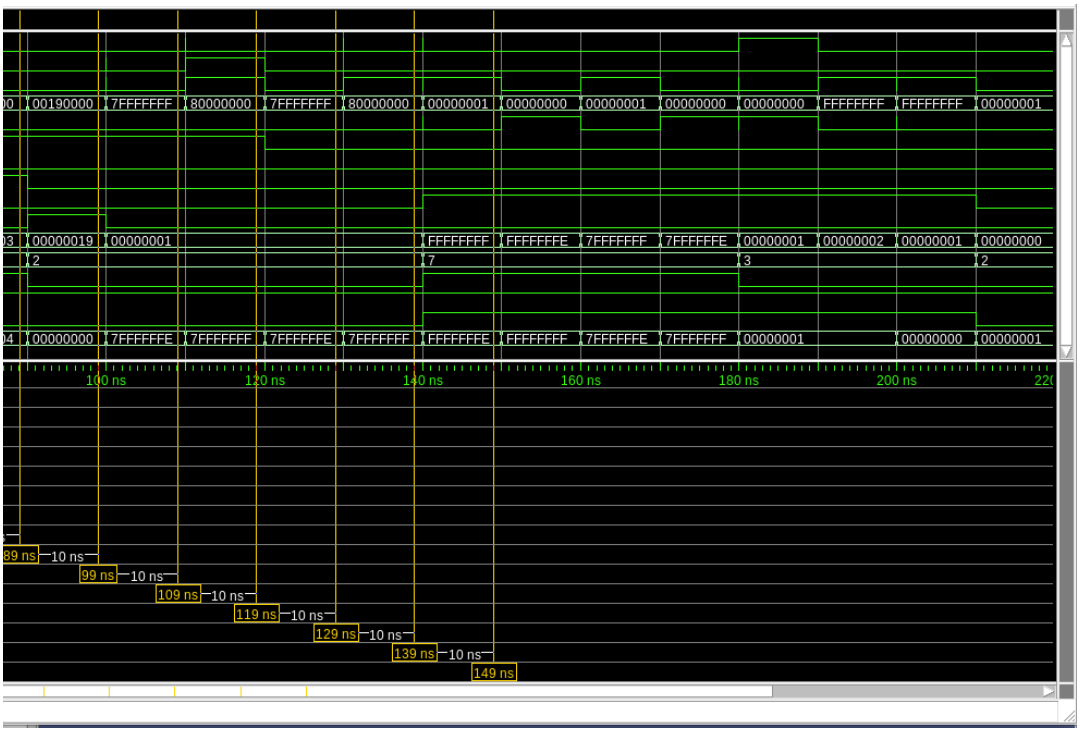


[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

Addition: Tests addition operation with positive numbers.
Subtraction: Tests subtraction operation with positive numbers.
Bitwise AND: Tests bitwise AND operation.
Bitwise OR: Tests bitwise OR operation.
Bitwise XOR: Tests bitwise XOR operation.
NOR: Tests NOR operation.
Shift Left Logical: Tests logical left shift operation.
Shift Right Logical: Tests logical right shift operation.
Shift Right Arithmetic: Tests arithmetic right shift operation.
Load Upper Immediate (LUI): Tests LUI operation.
11-14. Addition with overflow: Tests addition operation with both signed and unsigned inputs to cover overflow scenarios.
15-18. SLT (Set Less Than): Tests SLT operation with both signed and unsigned inputs.
19-22. Zero output: Tests scenarios where the output should be zero.

This comprehensive testbench covers various scenarios including positive and negative numbers, overflow conditions, zero output, and different ALU operations. We are able to see all of the different types of output for the overflow and signed numbers in the last few cases.





i_A and i_B represent the inputs to the ALU.
i_ALUout represents the ALU operation selection line.
o_Final represents the output of the ALU.
o_Carry_Out, o_Zero, o_Negative, and o_Overflow represent the status flags of the ALU.

[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.

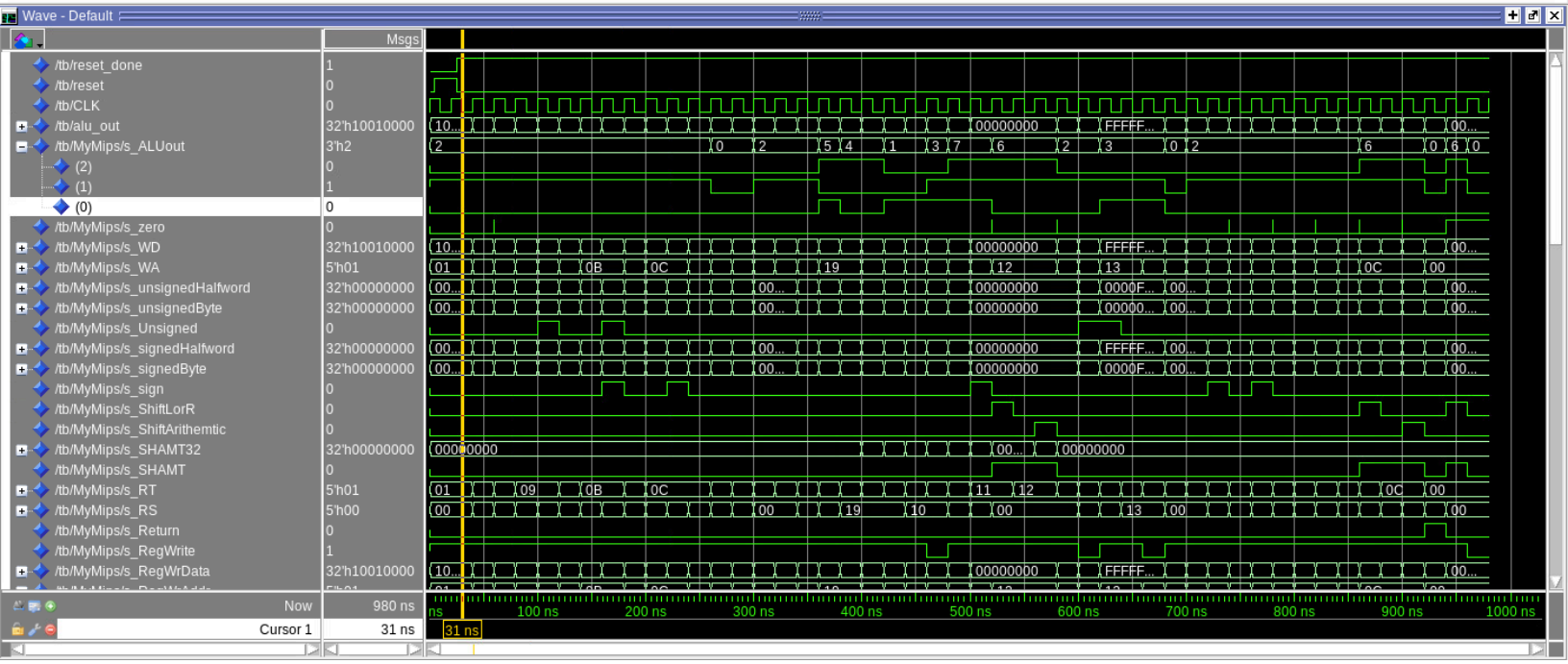
I explained the correctness of each test below.

[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.

See Attached Files to see the Mips Test Code

```
bash-4.4$ ./381_tf.sh test proj/mips/Proj1_base_test.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using $MGC_HOME=/usr/local/m
entor/calibre
All VHDL src files compiled successfully
Testing file: proj/mips/Proj1_base_test.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 48
Processor Cycles: 48
CPI:   1.0
Results in: output/Proj1_base_test.s
-----------------------------------------------
bash-4.4$
```

Screenshot of working waveform is below (I know it looks like alot however the first 5 lines are really the only one you need to pay attention to)



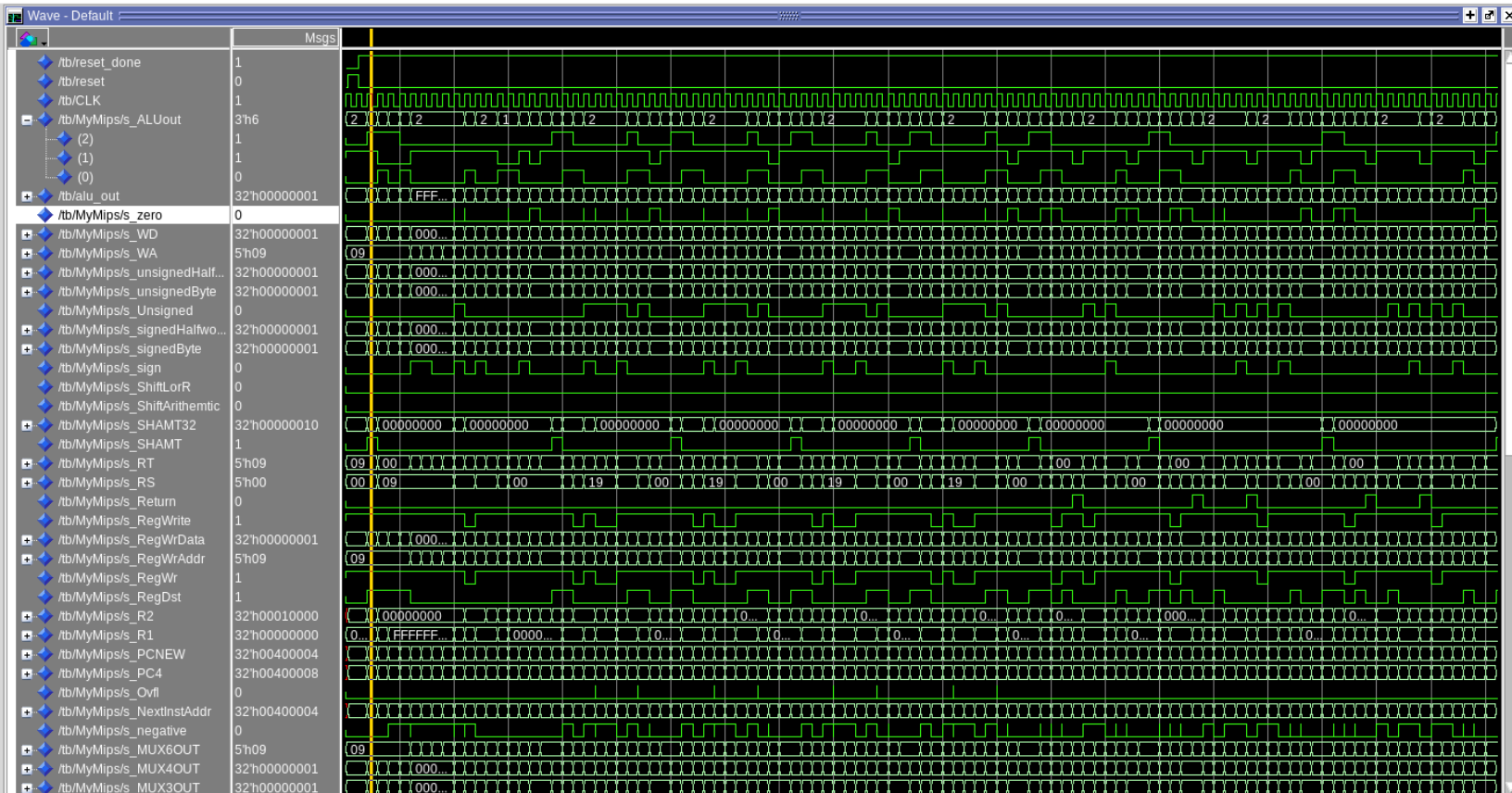
I know that it is working because of the MyMips/s_ALUout code and the alu_out lines. The first 8 instructions in my Mips code are add type instructions, and in my process 010 is equivalent to a add instruction, where the ALU only selects the Add line from the 8 to 1 mux in my ALU. This coupled with the alu_out line suggest that for all of the instructions the ALU is working properly… Furthermore because the values in the ALU make sense related to the test program the waveform is correct.

[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.

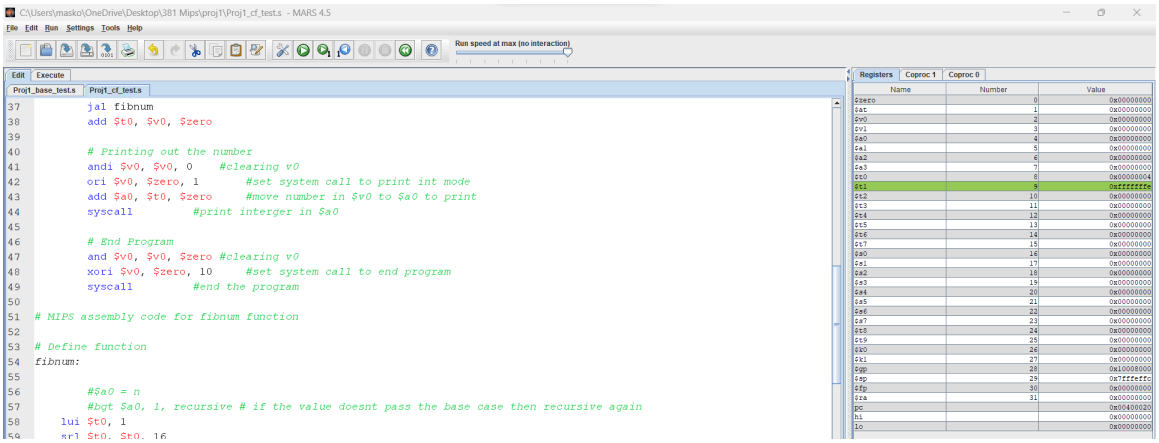
See Attached Files to see the Mips Test Code

```
bash-4.4$ ./381_tf.sh test proj/mips/Proj1_cf_test.s
Using LAB Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Proj1_cf_test.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 220
Processor Cycles: 221
CPI:   1.0
Results in: output/Proj1_cf_test.s
-----------------------------------------------
bash-4.4$
```

Screenshot of working waveform is below (I know it looks like alot however the first 5 lines are really the only one you need to pay attention to)



This is the waveform of my cf Mips code I wrote… I believe that it is working because of the MyMips/s_ALUout code and the alu_out lines. For example we can look at the code and the first couple instructions are Lui srl nor and xor, and if we look at the s_ALUout line we see that the proper numbers are set (lui is an add type which is 010)… and also running the code in mars we can see that the numbers on the right as we step through the program match the waveform alu_out line.

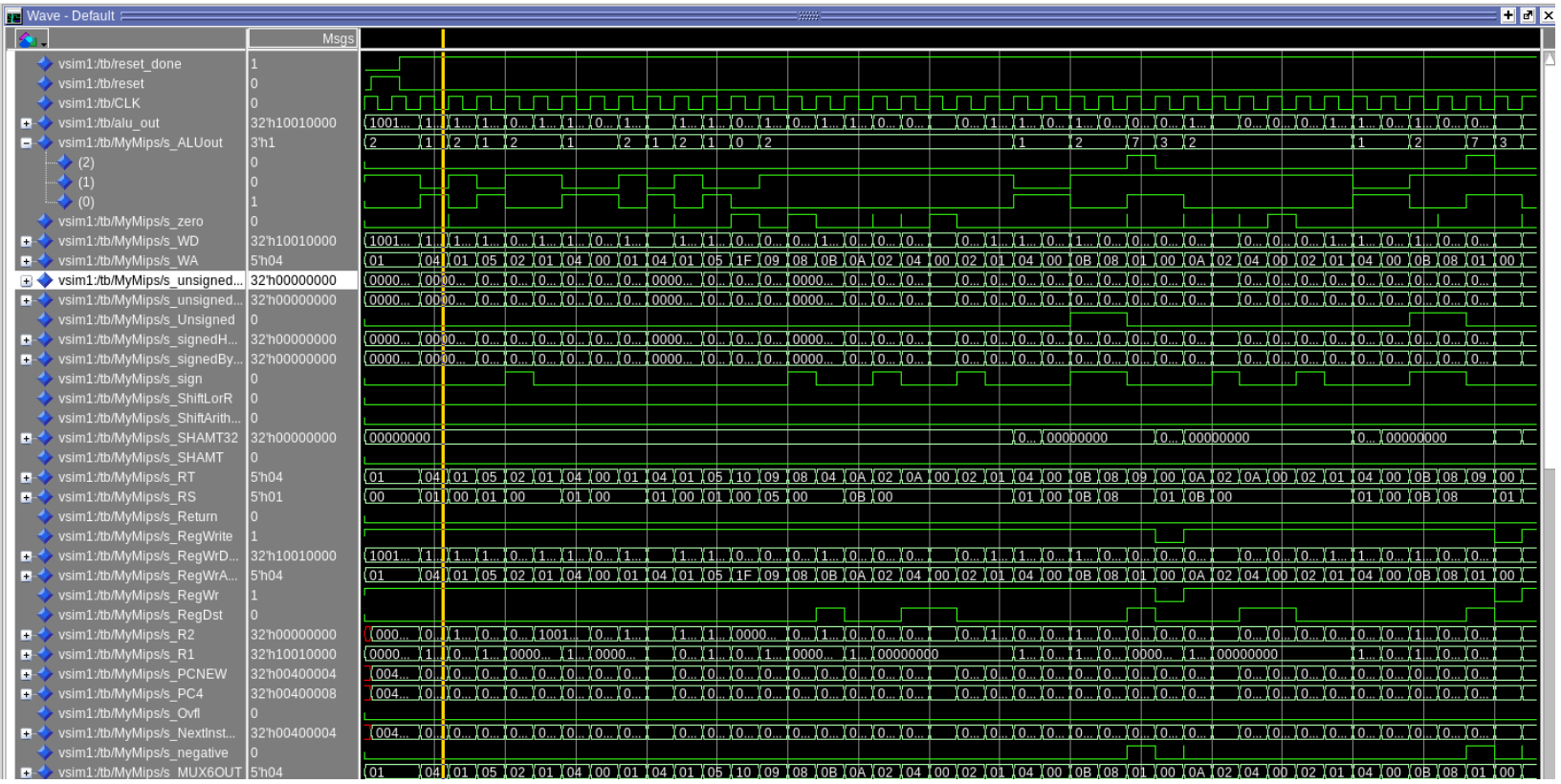


[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

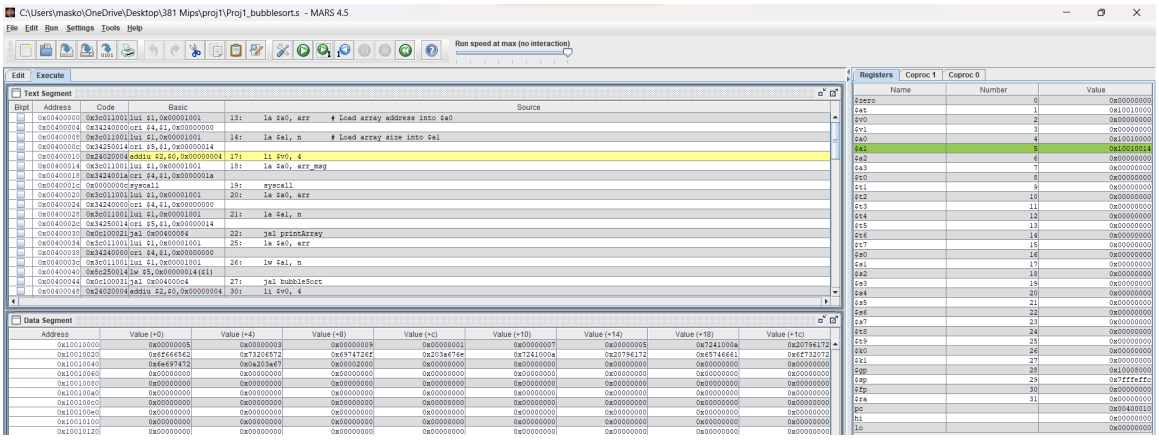
See Attached Files to see the Mips Test Code

```
bash-4.4$ ./381_tf.sh test proj/mips/Proj1_bubblesort.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using $MGC_HOME=/usr/local/m
entor/calibre
All VHDL src files compiled successfully
Testing file: proj/mips/Proj1_bubblesort.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 383
Processor Cycles: 384
CPI:    1.0
Results in: output/Proj1_bubblesort.s
-----------------------------------------------
bash-4.4$
```

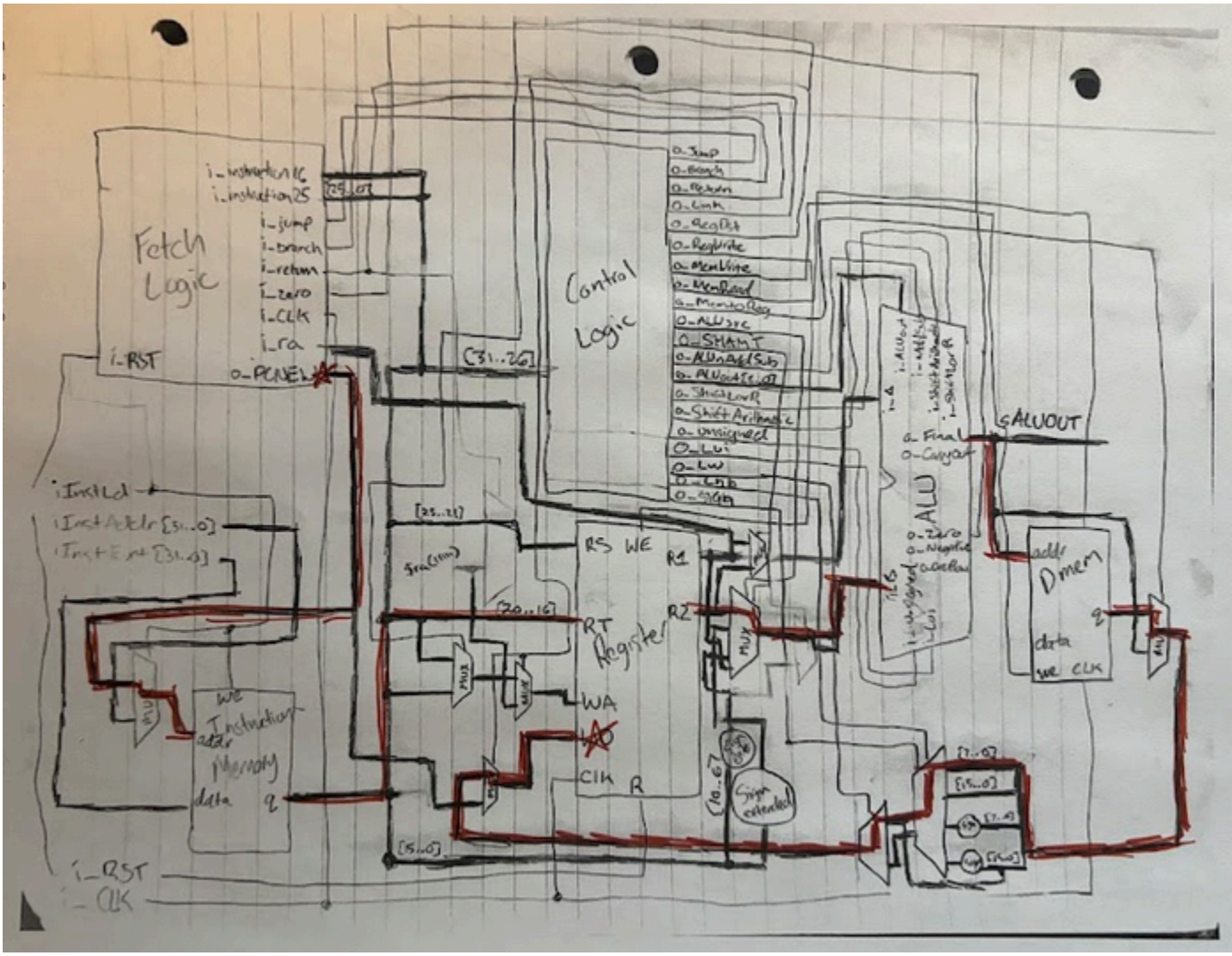
Screenshot of working waveform is below (I know it looks like alot however the first 5 lines are really the only one you need to pay attention to)



This is the waveform for the bubble sort mips code. This is correct again because of the MyMips/s_ALUout code and the alu_out lines. For example we can look at the code and the first couple instructions are la la li la and if we look at the s_ALUout line we see that the proper numbers are set… and also running the code in mars we can see that the numbers on the right as we step through the program match the waveform alu_out line. for example this mars code is showing the 4th instruction and this matches the alu_out in $at



[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?



The maximum frequency is 25.11mhz The critical path is from the FetchLogic to writing into the Register. I would work on changing the Instruction memory because it is in most of the instructions and in our synthesis this was the longest single part of datapath. In this datapath the instruction spends the most amount of time in the Instruction memory. However If I were to pick a different one to improve I would choose the ALU because the second longest part of the datapath was the shift operation and muxes inside of the ALU. Overall in addition the Muxes and the Dmem are also rather slow… Register operations are much faster than I had expected.

```
#
# CprE 381 toolflow Timing dump
#

FMax: 25.11mhz Clk Constraint: 20.00ns Slack: -19.82ns

The path is given below

 ====================================================================
 From Node    : FetchLogic:FETCH|PC:PC4|dffg:\G_NBit_REG:3:DFF|s_Q
 To Node      : Reg32File:REG|RegFile:\G_NBit_REG:3:REG|dffg:\G_NBit_REG:0:DFF|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
      0.000      0.000           launch edge time
      2.775      2.775  R        clock network delay
      3.007      0.232     uTco  FetchLogic:FETCH|PC:PC4|dffg:\G_NBit_REG:3:DFF|s_Q
      3.007      0.000 FF  CELL  FETCH|PC4|\G_NBit_REG:3:DFF|s_Q|q
      3.390      0.383 FF    IC  s_IMemAddr[3]~0|datad
      3.515      0.125 FF  CELL  s_IMemAddr[3]~0|combout
      5.824      2.309 FF    IC  IMem|ram~33683|dataa
      6.248      0.424 FF  CELL  IMem|ram~33683|combout
      6.519      0.271 FF    IC  IMem|ram~33684|datab
      6.944      0.425 FF  CELL  IMem|ram~33684|combout
      8.350      1.406 FF    IC  IMem|ram~33692|dataa
      8.774      0.424 FF  CELL  IMem|ram~33692|combout
     10.979      2.205 FF    IC  IMem|ram~33693|datac
     11.260      0.281 FF  CELL  IMem|ram~33693|combout
     11.489      0.229 FF    IC  IMem|ram~33704|datad
     11.639      0.150 FR  CELL  IMem|ram~33704|combout
     11.842      0.203 RR    IC  IMem|ram~33832|datac
     12.129      0.287 RR  CELL  IMem|ram~33832|combout
     13.184      1.055 RR    IC  IMem|ram~34003|datad
     13.339      0.155 RR  CELL  IMem|ram~34003|combout
```

```
    11.260      0.281 FF  CELL  IMem|ram~33693|combout
    11.489      0.229 FF    IC  IMem|ram~33704|datad
    11.639      0.150 FR  CELL  IMem|ram~33704|combout
    11.842      0.203 RR    IC  IMem|ram~33832|datac
    12.129      0.287 RR  CELL  IMem|ram~33832|combout
    13.184      1.055 RR    IC  IMem|ram~34003|datad
    13.339      0.155 RR  CELL  IMem|ram~34003|combout
    13.572      0.233 RR    IC  IMem|ram~34174|datab
    14.006      0.434 RF  CELL  IMem|ram~34174|combout
    15.987      1.981 FF    IC  REG|mux32_1_1|Mux13~7|datad
    16.137      0.150 FR  CELL  REG|mux32_1_1|Mux13~7|combout
    16.339      0.202 RR    IC  REG|mux32_1_1|Mux13~8|datad
    16.494      0.155 RR  CELL  REG|mux32_1_1|Mux13~8|combout
    21.079      4.585 RR    IC  REG|mux32_1_1|Mux13~9|dataa
    21.437      0.358 RR  CELL  REG|mux32_1_1|Mux13~9|combout
    21.639      0.202 RR    IC  REG|mux32_1_1|Mux13~19|datac
    21.926      0.287 RR  CELL  REG|mux32_1_1|Mux13~19|combout
    22.153      0.227 RR    IC  \MUX10_32:18:MUX10|o_O~0|datad
    22.308      0.155 RR  CELL  \MUX10_32:18:MUX10|o_O~0|combout
    22.688      0.380 RR    IC  A|Shifter|\MUX2_32:18:MUX2|o_O~2|datad
    22.843      0.155 RR  CELL  A|Shifter|\MUX2_32:18:MUX2|o_O~2|combout
    23.057      0.214 RR    IC  A|Shifter|\MUX2_32:17:MUX2|o_O~3|datad
    23.212      0.155 RR  CELL  A|Shifter|\MUX2_32:17:MUX2|o_O~3|combout
    23.961      0.749 RR    IC  A|Shifter|\MUX5_32:5:MUX5|o_O~1|datac
    24.248      0.287 RR  CELL  A|Shifter|\MUX5_32:5:MUX5|o_O~1|combout
    24.966      0.718 RR    IC  A|Shifter|\MUX5_32:5:MUX5|o_O~2|datad
    25.121      0.155 RR  CELL  A|Shifter|\MUX5_32:5:MUX5|o_O~2|combout
    25.330      0.209 RR    IC  A|Shifter|\MUX16_32:5:MUX16|o_O~0|datac
    25.617      0.287 RR  CELL  A|Shifter|\MUX16_32:5:MUX16|o_O~0|combout
    25.819      0.202 RR    IC  A|Shifter|\MUX16_32:5:MUX16|o_O~2|datac
    26.106      0.287 RR  CELL  A|Shifter|\MUX16_32:5:MUX16|o_O~2|combout
    26.309      0.203 RR    IC  A|muxxx|Mux6|\G_NBit_MUX:5:MUXI|o_O~1|datad
    26.464      0.155 RR  CELL  A|muxxx|Mux6|\G_NBit_MUX:5:MUXI|o_O~1|combout
    26.669      0.205 RR    IC  A|muxxx|Mux6|\G_NBit_MUX:5:MUXI|o_O~2|datad
    26.808      0.139 RF  CELL  A|muxxx|Mux6|\G_NBit_MUX:5:MUXI|o_O~2|combout
    27.045      0.237 FF    IC  A|luiselection|\G_NBit_MUX:5:MUXI|o_O~0|datad
    27.170      0.125 FF  CELL  A|luiselection|\G_NBit_MUX:5:MUXI|o_O~0|combout
    29.778      2.608 FF    IC  DMem|ram~33280|dataa
    30.202      0.424 FF  CELL  DMem|ram~33280|combout
    31.206      1.004 FF    IC  DMem|ram~33281|datad
    31.356      0.150 FR  CELL  DMem|ram~33281|combout
    33.029      1.673 RR    IC  DMem|ram~33284|datad
    33.184      0.155 RR  CELL  DMem|ram~33284|combout
    33.387      0.203 RR    IC  DMem|ram~33287|datad
    33.542      0.155 RR  CELL  DMem|ram~33287|combout
    35.297      1.755 RR    IC  DMem|ram~33319|datad
    35.452      0.155 RR  CELL  DMem|ram~33319|combout
    39.770      4.318 RR    IC  DMem|ram~33320|datad
    39.925      0.155 RR  CELL  DMem|ram~33320|combout
    40.131      0.206 RR    IC  DMem|ram~33321|datad
    40.286      0.155 RR  CELL  DMem|ram~33321|combout
    40.491      0.205 RR    IC  DMem|ram~33492|datad
    40.646      0.155 RR  CELL  DMem|ram~33492|combout
    40.849      0.203 RR    IC  \MUX5_32:0:MUX5|o_O~0|datad
    41.004      0.155 RR  CELL  \MUX5_32:0:MUX5|o_O~0|combout
    41.206      0.202 RR    IC  \MUX5_32:0:MUX5|o_O~1|datac
    41.493      0.287 RR  CELL  \MUX5_32:0:MUX5|o_O~1|combout
    42.808      1.315 RR    IC  REG|\G_NBit_REG:3:REG|\G_NBit_REG:0:DFF|s_Q|asdata
    43.214      0.406 RR  CELL  Reg32File:REG|RegFile:\G_NBit_REG:3:REG|dffg:\G_NBit_REG:0:DFF|s_Q
Data Required Path:
Total (ns)  Incr (ns)     Type  Element
==========  ========= ==  ====  ===================================
```