CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

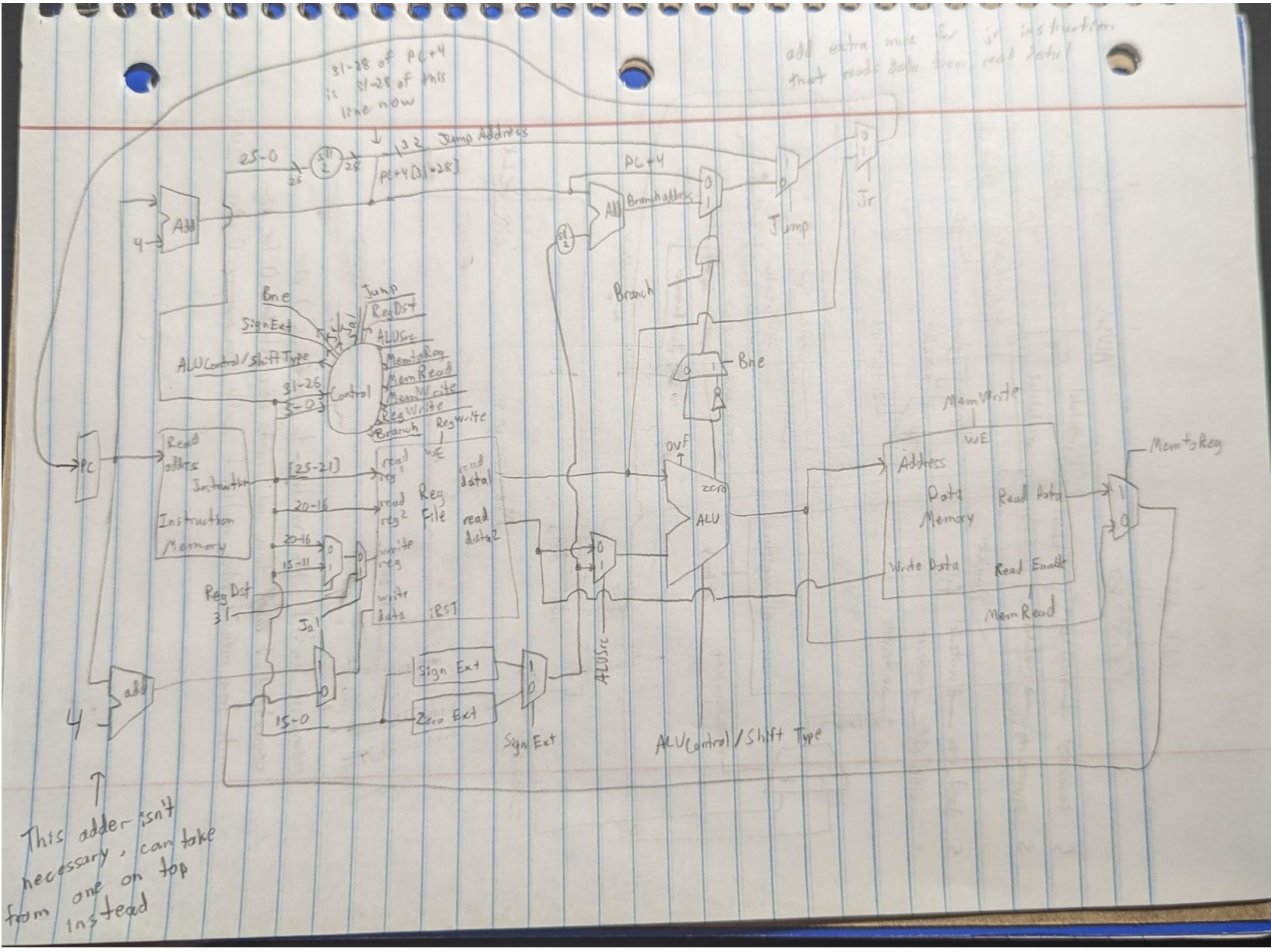
Team Members: _____Connor Hand_____

_____Zach Scurlock_____

Project Teams Group #:_____3_____

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

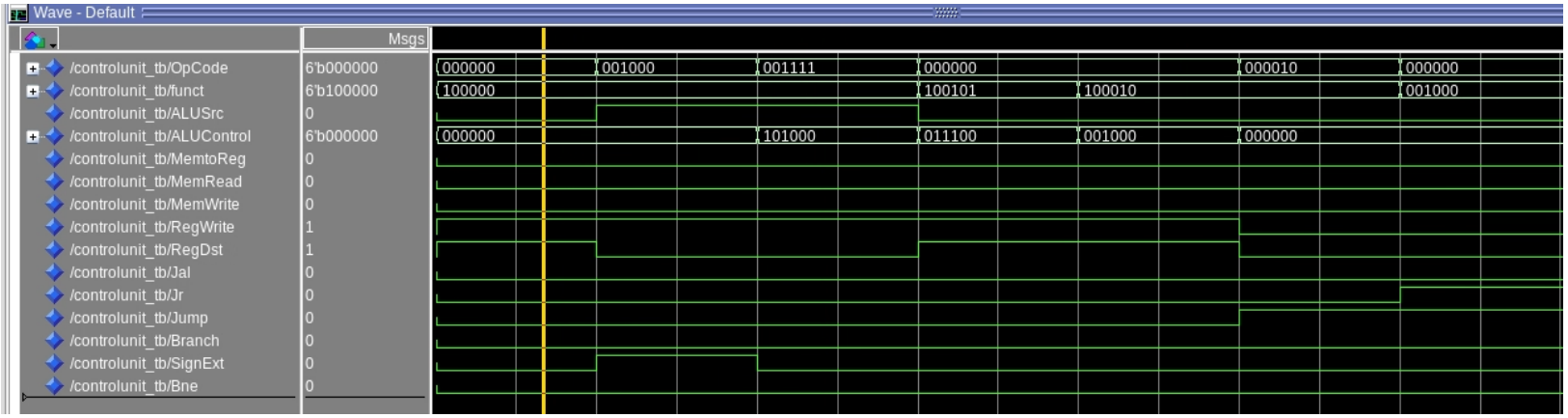
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[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.

	Instruction	Opcode (Binary)	Funct (Binary)	Control Signals												
				ALUSrc	ALUControl/Shift Type (6 bits)	MemtoReg	MemRead	MemWrite	RegWrite	RegDst (0 for rt, 1 for rd)	Jal	Jr	Jump	Branch	SignExt	Bne
3	add	"000000"	"100000"	0	0000/XX	0	0	0	1	1	0	0	0	0	X	0
4	addi	"001000"	"------"	1	0000/XX	0	0	0	1	0	0	0	0	0	1	0
5	addiu	"001001"	"------"	1	0001/XX	0	0	0	1	0	0	0	0	0	1	0
6	addu	"000000"	"100001"	0	0001/XX	0	0	0	1	1	0	0	0	0	X	0
7	and	"000000"	"100100"	0	0100/XX	0	0	0	1	1	0	0	0	0	X	0
8	andi	"001100"	"------"	1	0100/XX	0	0	0	1	0	0	0	0	0	0	0
9	lui	"001111"	"------"	1	1010/XX	0	0	0	1	0	0	0	0	0	0	0
10	lw	"100011"	"------"	1	0000/XX	1	1	0	1	0	0	0	0	0	1	0
11	nor	"000000"	"100111"	0	0101/XX	0	0	0	1	1	0	0	0	0	X	0
12	xor	"000000"	"100110"	0	0110/XX	0	0	0	1	1	0	0	0	0	X	0
13	xori	"001110"	"------"	1	0110/XX	0	0	0	1	0	0	0	0	0	0	0
14	or	"000000"	"100101"	0	0111/XX	0	0	0	1	1	0	0	0	0	X	0
15	ori	"001101"	"------"	1	0111/XX	0	0	0	1	0	0	0	0	0	0	0
16	slt	"000000"	"101010"	0	1000/XX	0	0	0	1	1	0	0	0	0	X	0
17	slti	"001010"	"------"	1	1000/XX	0	0	0	1	0	0	0	0	0	1	0
18	sll	"000000"	"000000"	0	1001/01	0	0	0	1	1	0	0	0	0	X	0
19	srl	"000000"	"000010"	0	1001/10	0	0	0	1	1	0	0	0	0	X	0
20	sra	"000000"	"000011"	0	1001/11	0	0	0	1	1	0	0	0	0	X	0
21	sw	"101011"	"------"	1	0000/XX	0	0	1	0	X	0	0	0	0	1	0
22	sub	"000000"	"100010"	0	0010/XX	0	0	0	1	1	0	0	0	0	X	0
23	subu	"000000"	"100011"	0	0011/XX	0	0	0	1	1	0	0	0	0	X	0
24	beq	"000100"	"------"	0	0010/XX	0	0	0	0	X	0	0	0	1	X	0
25	bne	"000101"	"------"	0	0010/XX	0	0	0	0	X	0	0	0	1	X	1
26	j	"000010"	"------"	X	XXXX/XX	0	0	0	0	X	0	0	1	0	X	0
27	jal	"000011"	"------"	X	XXXX/XX	0	0	0	1	X	1	0	1	0	X	0
28	jr	"000000"	"001000"	X	XXXX/XX	0	0	0	0	X	0	1	1	0	X	0
29	There are definitely more don't cares (X) than the ones marked, all X's will be implemented as 0's in the control unit.															

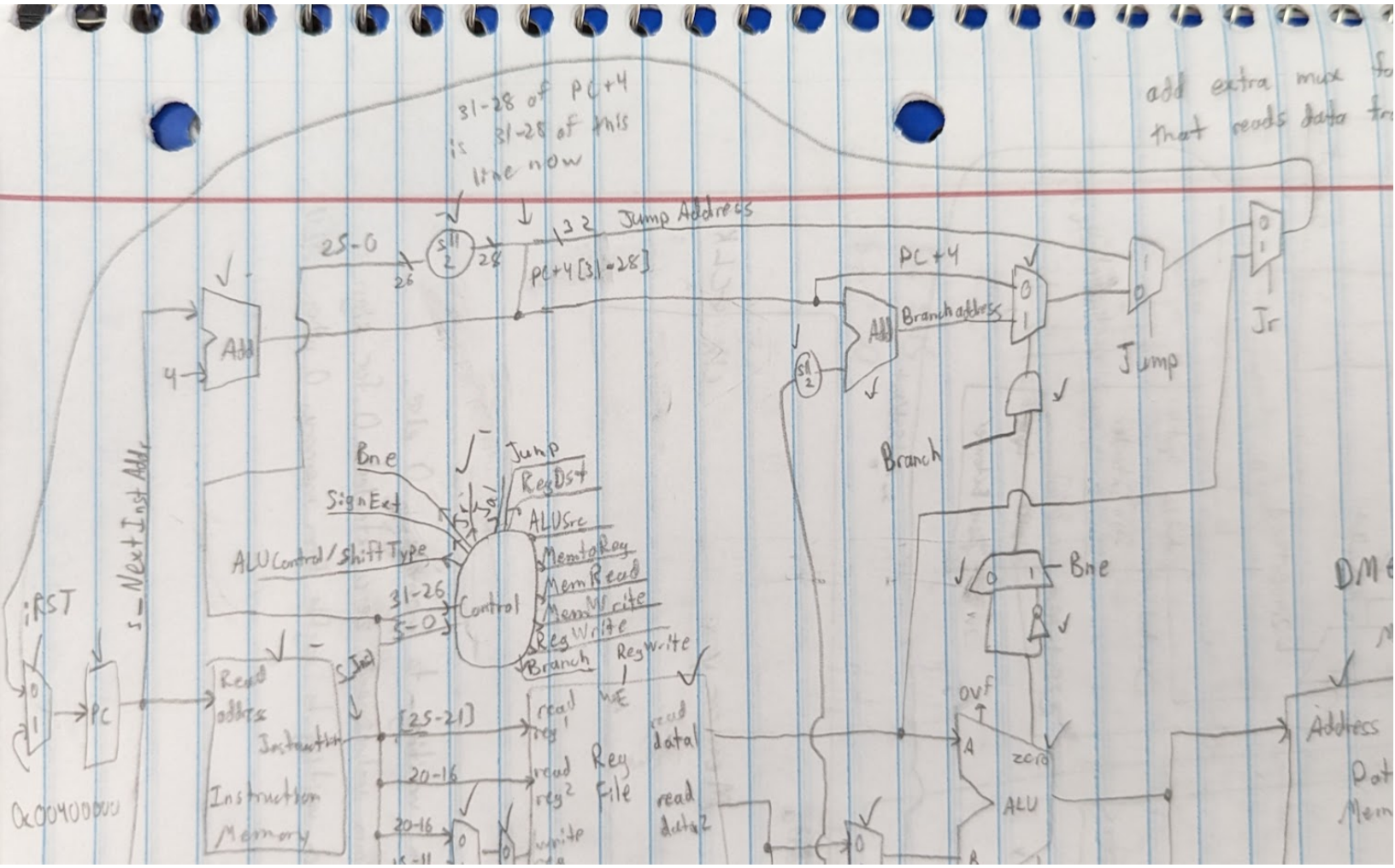
[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually, and show that your output matches the expected control signals from problem 1(a).



[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.

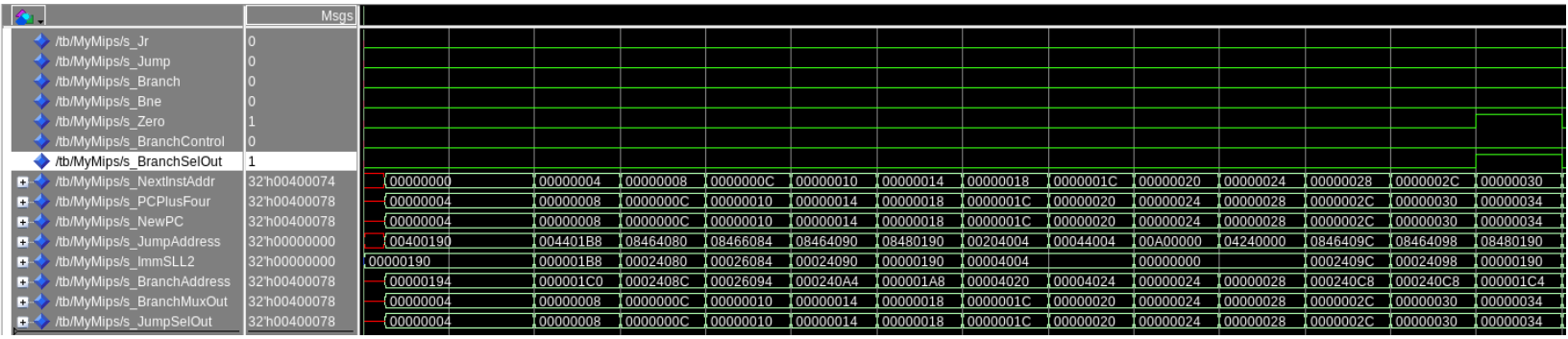
branch - MUXes selects the calculated branch address to be sent to PC
jump - MUXes select the calculated jump address to be sent to PC
jump register - MUXes select the output of read data 1 to be sent to PC

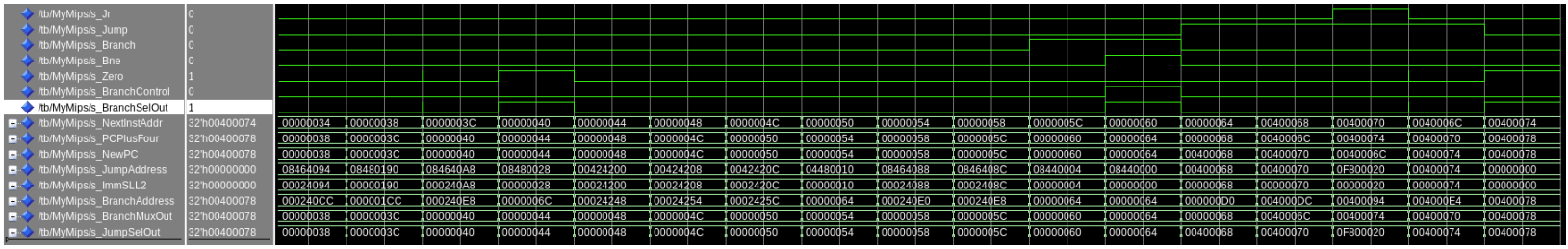
[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?



Additional control signals for the fetch logic include Bne, Branch, Jump, and Jr. Also, iRST is used as a control signal to determine when PC should be resetting to 0x00400000 (which is the start of instruction memory).

[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.





The control flow signals are at the top of the waveform and the signals that hold the possible new values of PC are on the bottom. This is the fetch logic in our Proj1_base_test.s file.

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

srl shifts right and pads with zeroes, sra shifts right but pads with whatever the sign bit was. sla does not exist because the sign bit is only on the left side (the side that shift right needs to pad)

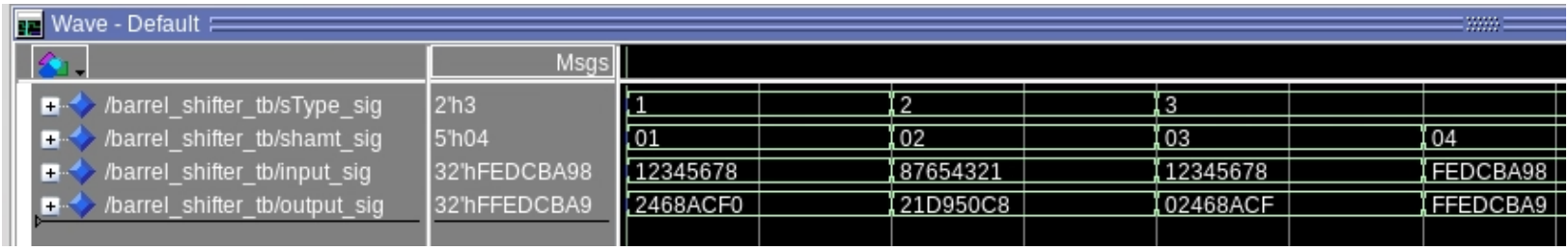
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

Or VHDL code is written in dataflow and using that it detects what the sign bit is when doing an arithmetic shift and pads with either 1 or 0, when doing a logical shift it only pads with 0.

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

Use 4:1 MUXes that have inputs that are no shift, sll, srl, and sra instead of 2:1 MUXes that only have srl and sra as inputs.

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

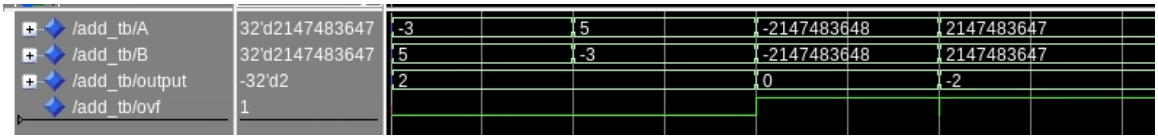


In order, the instructions executed are sll by 1, srl by 2, sra by 3, and sra by 4. Had two tests for sra, one with a positive number and one with a negative number.

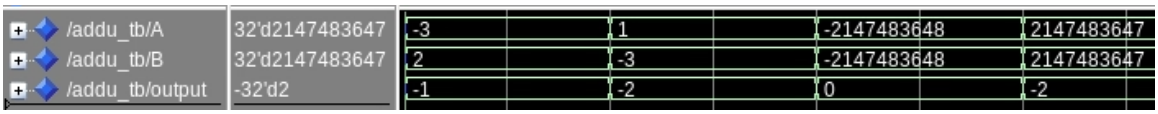
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

All the other functional units in the ALU are add, addu, sub, subu, AND, NOR, XOR, OR, slt, and lui. These were all designed using dataflow. Add, addu, sub, and subu are made in the same way except add and sub have more when statements to determine overflow. AND, NOR, XOR, and OR are all made in the same way just by ANDing, NORing, XORing, or ORing the inputs. Slt was made by setting the output to 1 “when” inputA < inputB, set output to 0 else. Lui was made by taking the input, shifting it left 16 and outputting it.

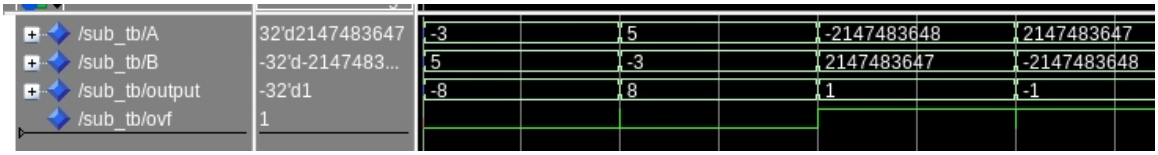
[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



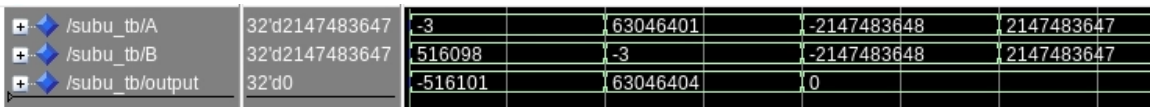
This is the regular add instruction with overflow testing.



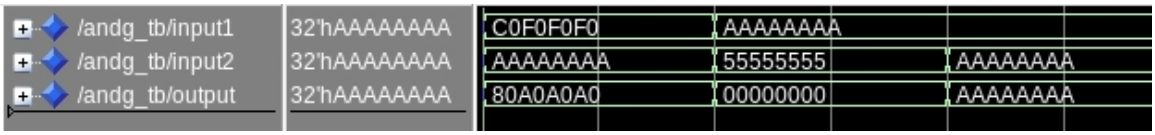
This is the addu instruction.



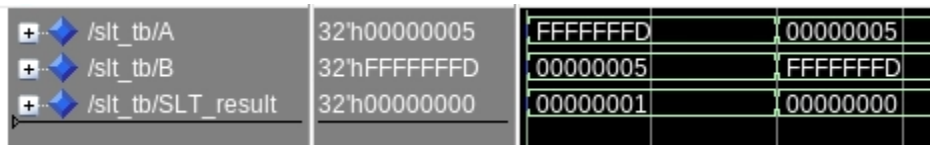
This is the regular sub instruction with overflow testing.



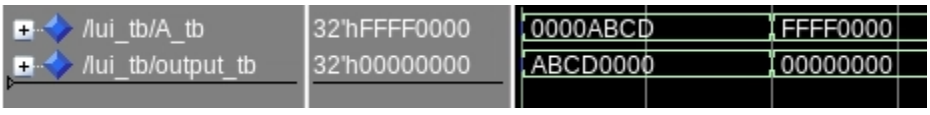
This is the subu instruction.



This is the and instruction.

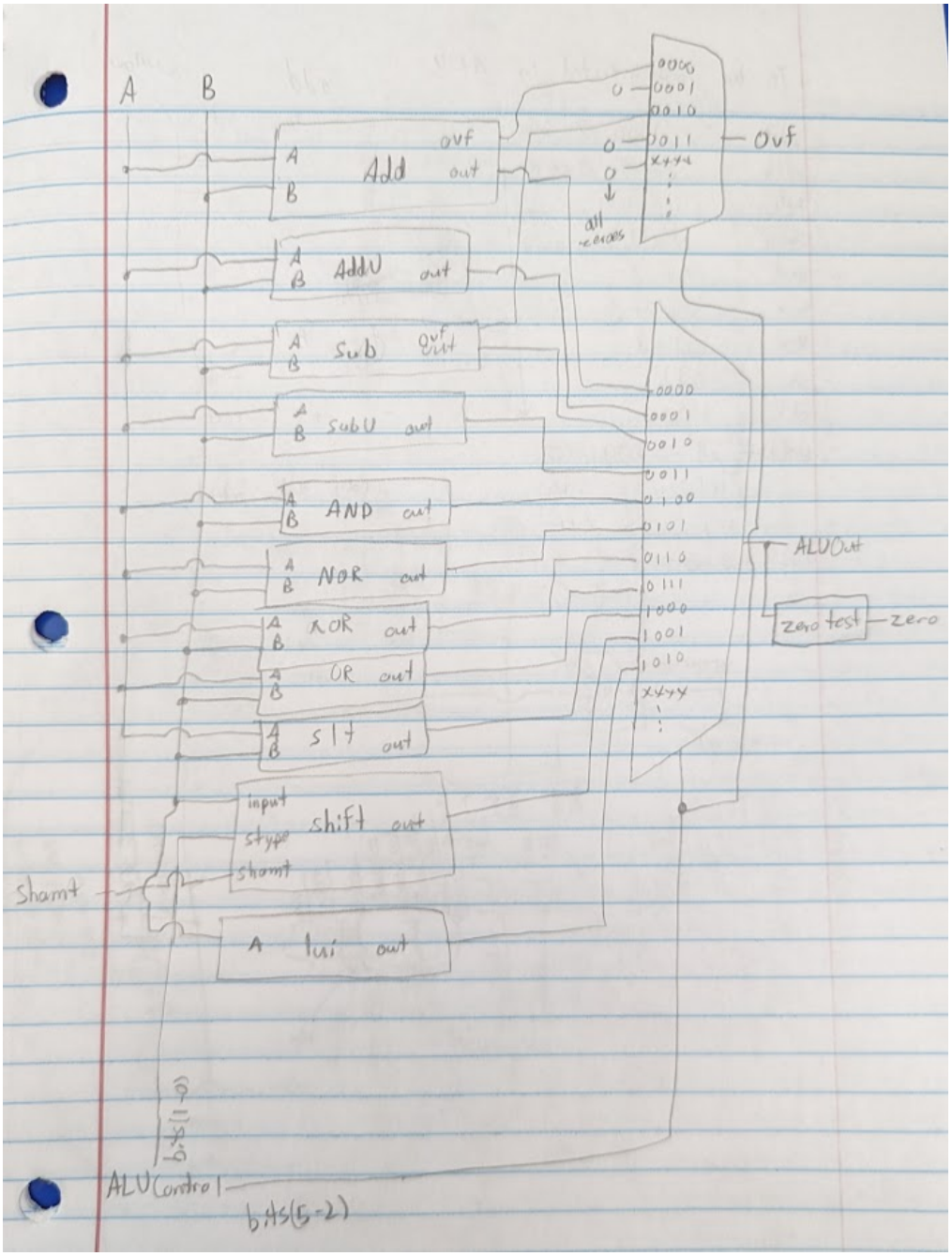


This is the slt instruction.



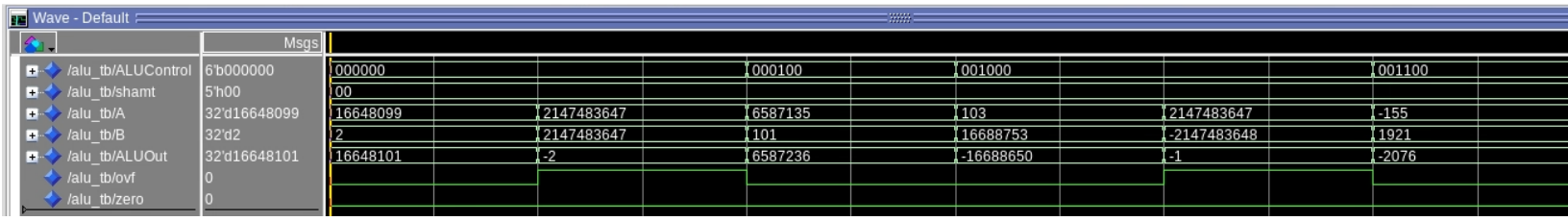
This is the lui instruction.

[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?

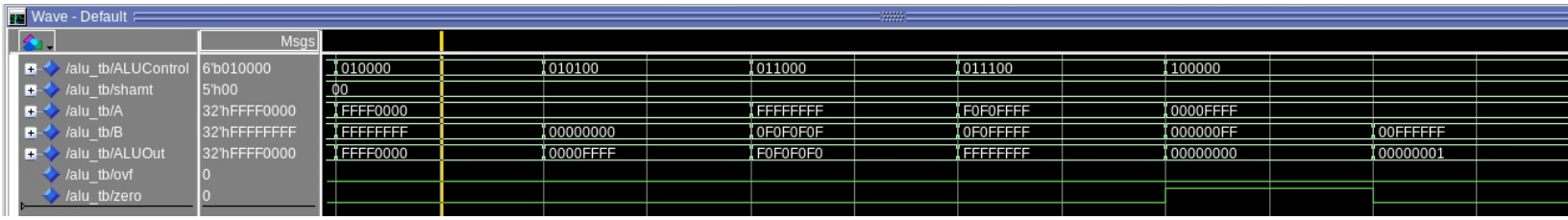


Overflow is calculated by putting the only two possible overflow outputs (add and sub) into a MUX with the same select line as the MUX that selects ALUOut. Zero is calculated by inputting the ALU result into a component, and that component checks if it is 0 and outputs 1 if it is. Slt is implemented as a dataflow component using this line: SLT_result <= x“00000001” when (signed(A) < signed(B)) else x“00000000”;

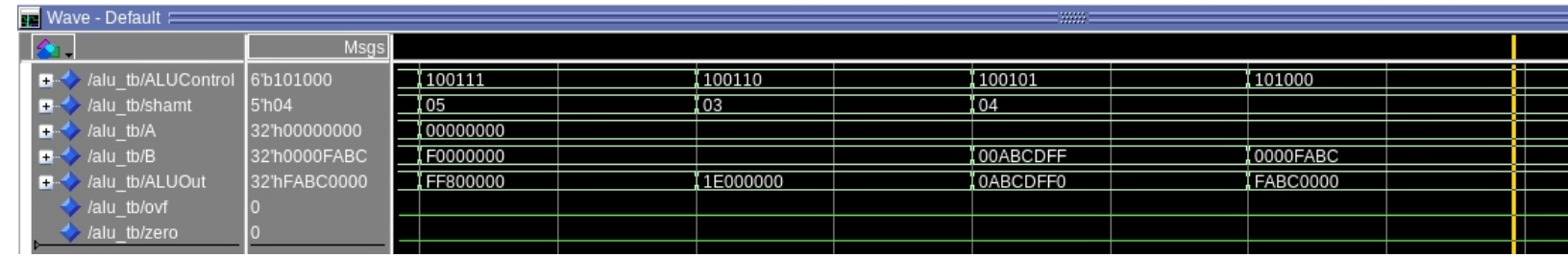
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



add add w/ ovf addu sub sub w/ ovf subu

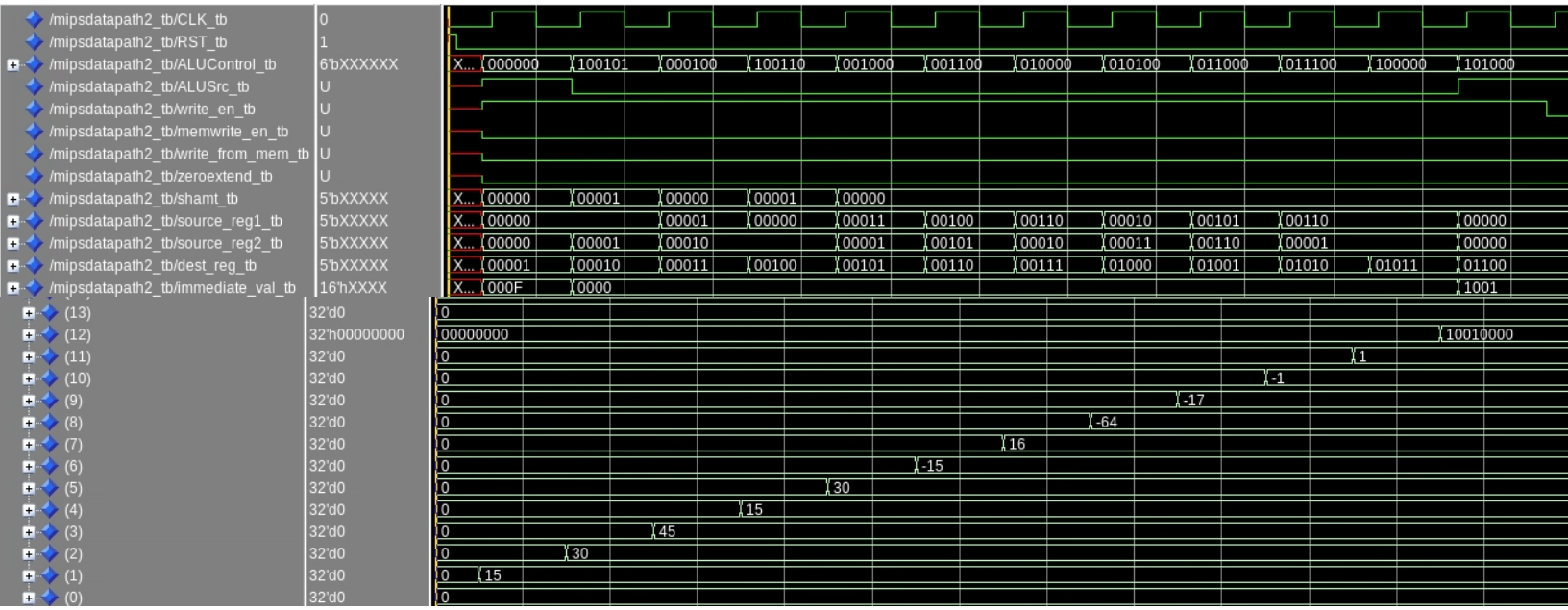


and nor xor or last 2 slt



sra srl sll lui

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

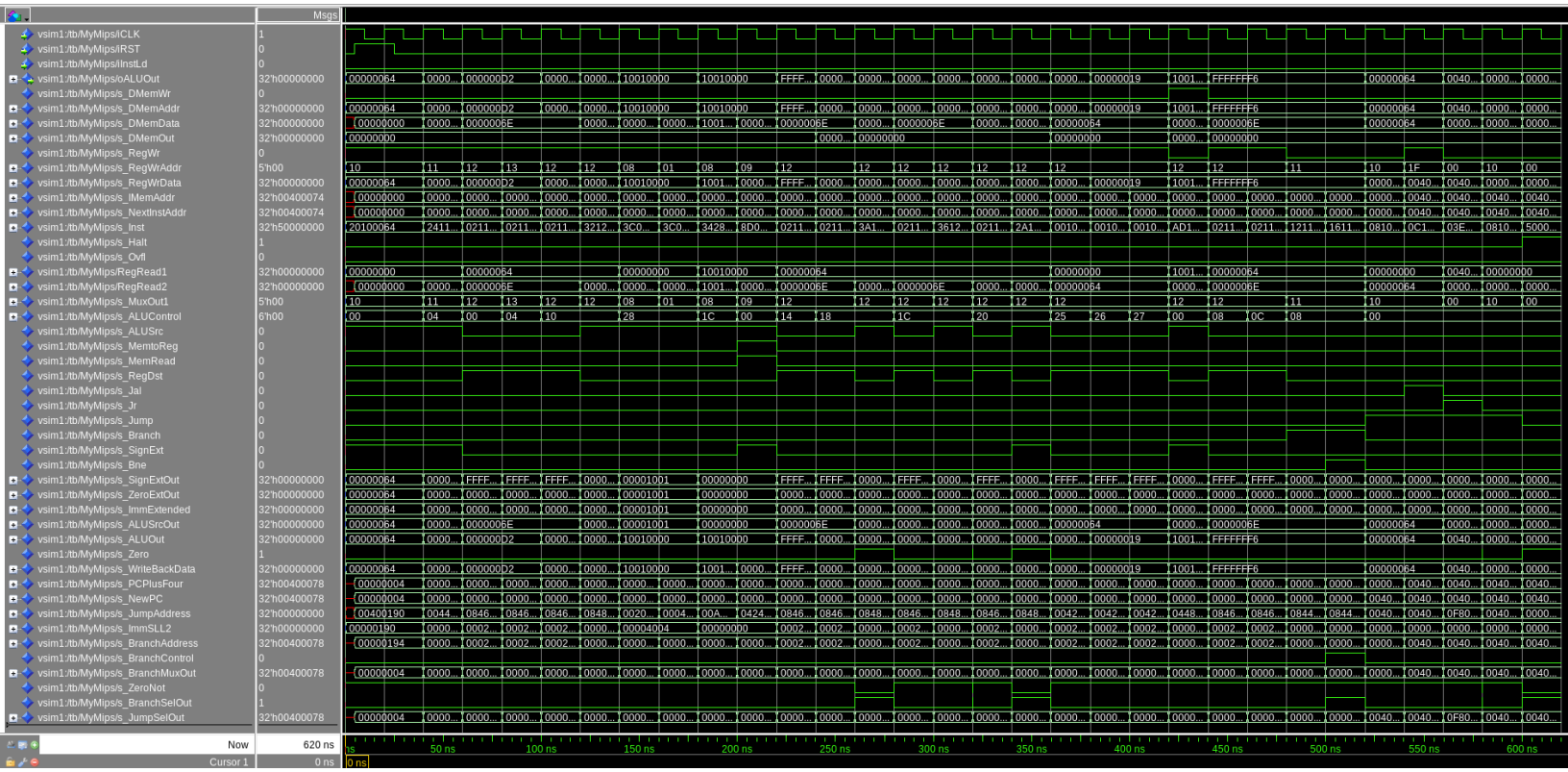


The top image shows control signals and the bottom image is the registers that were manipulated. Our test plan is comprehensive because all cases were tested including the overflow and zero outputs, subtracting 2 negative numbers, trying different values for the specific components etc.

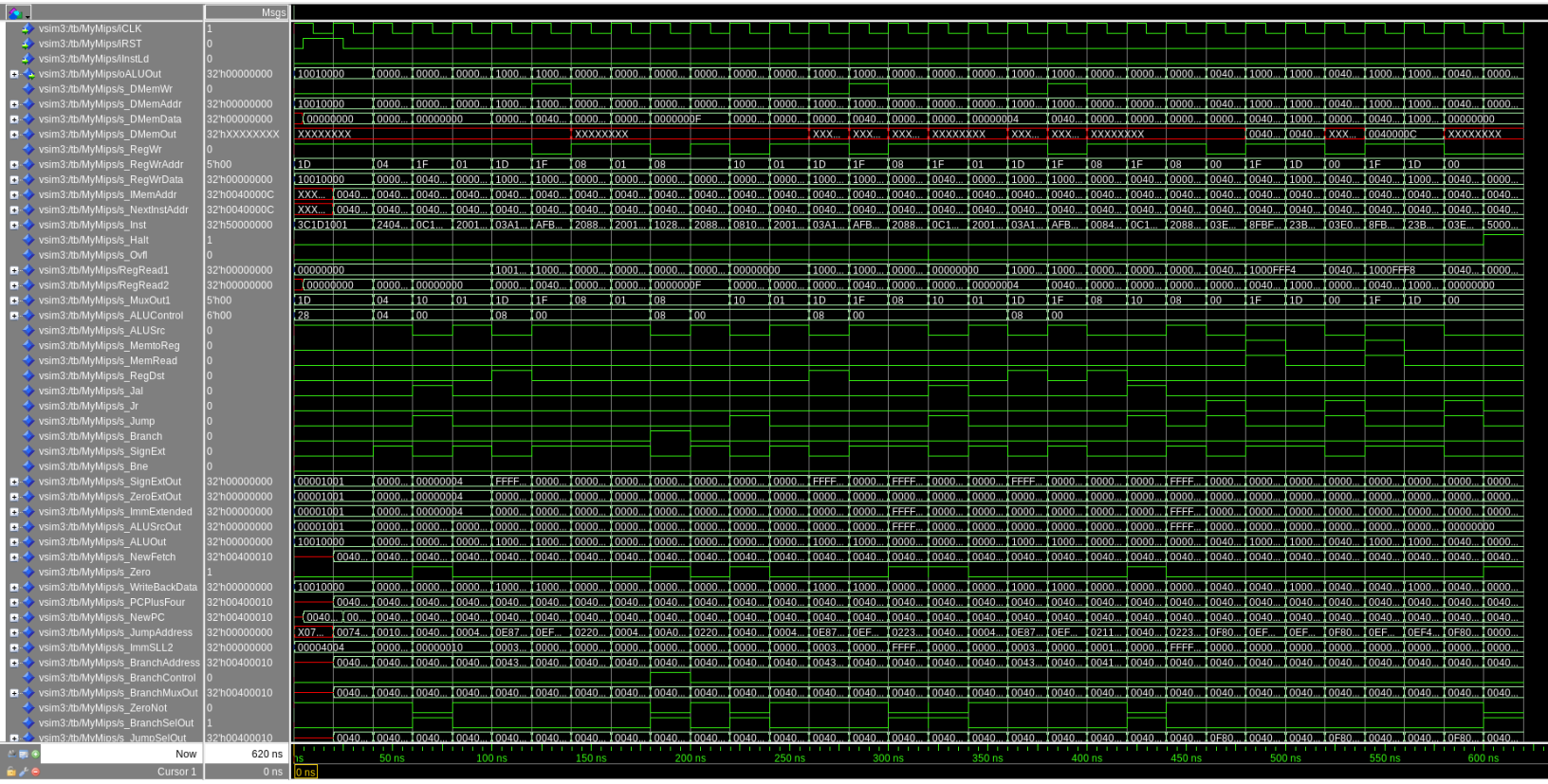
[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

If you trace each of the signals in the waveforms for all of the following tests you will see that everything is working as intended.

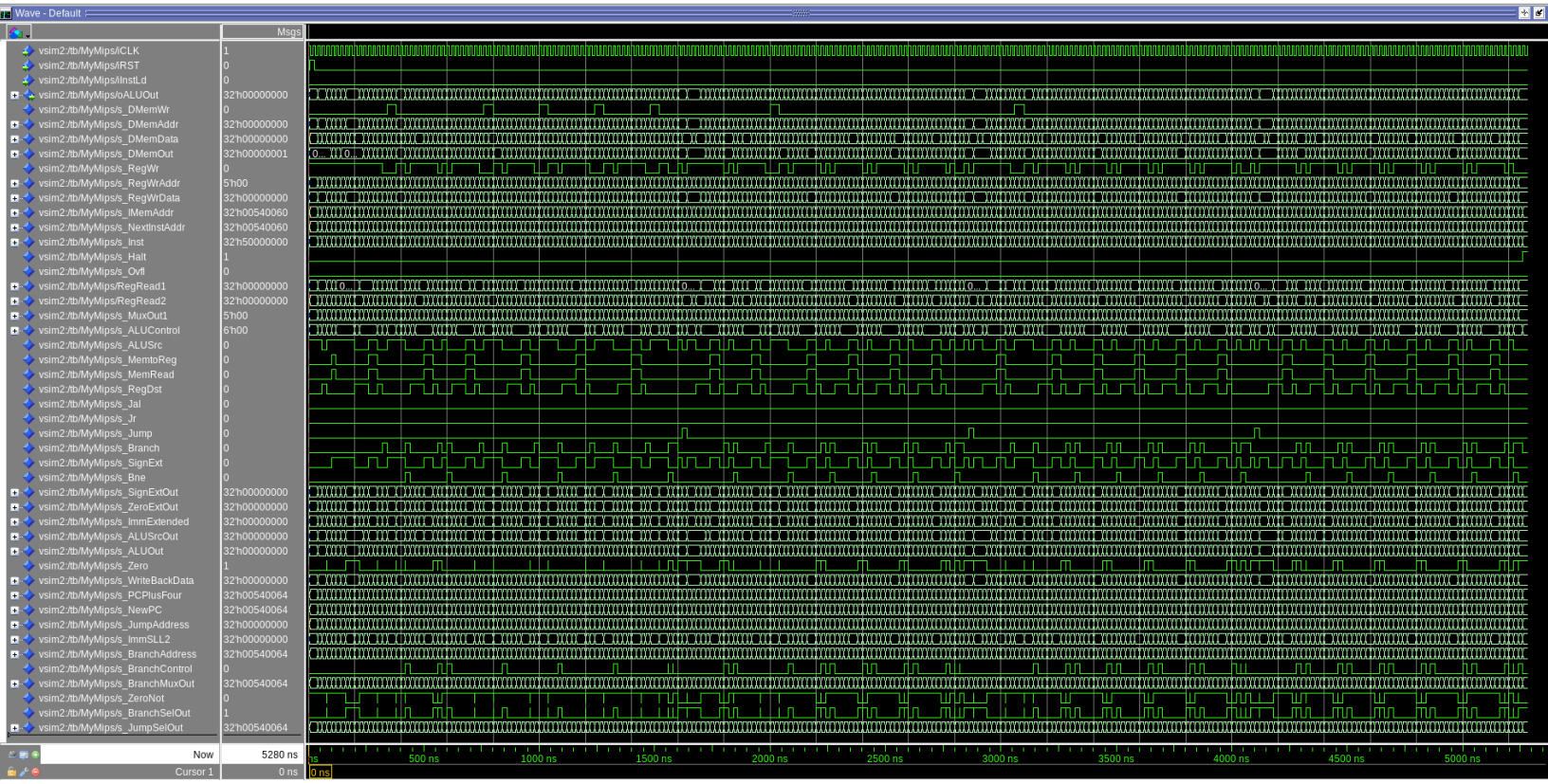
[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.



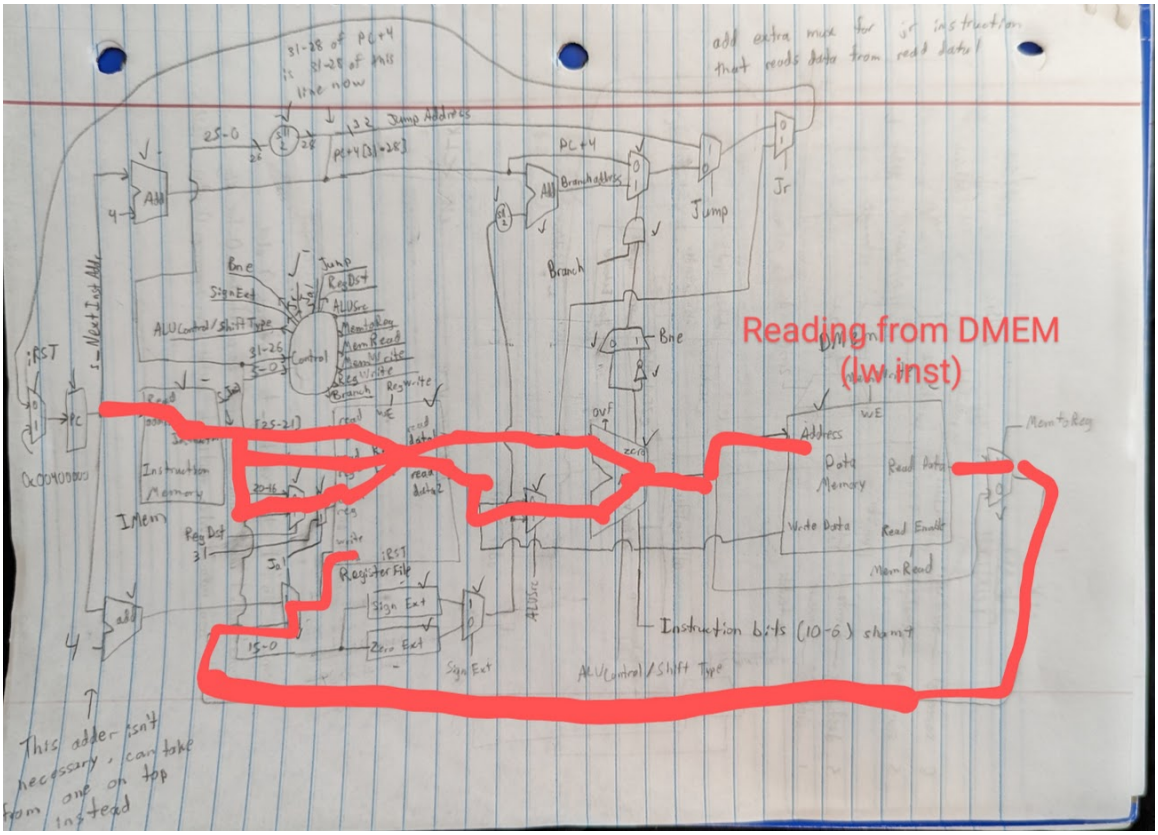
[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.



[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 maximum frequency is 23.38mhz. Data Memory would be the most important thing for us to improve because it is currently the slowest component taking ~13.6ns. Improving any part of the instruction memory, register file, or ALU would also help improve the frequency because it is a single cycle processor.