CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: _____Ellery Sabado_____

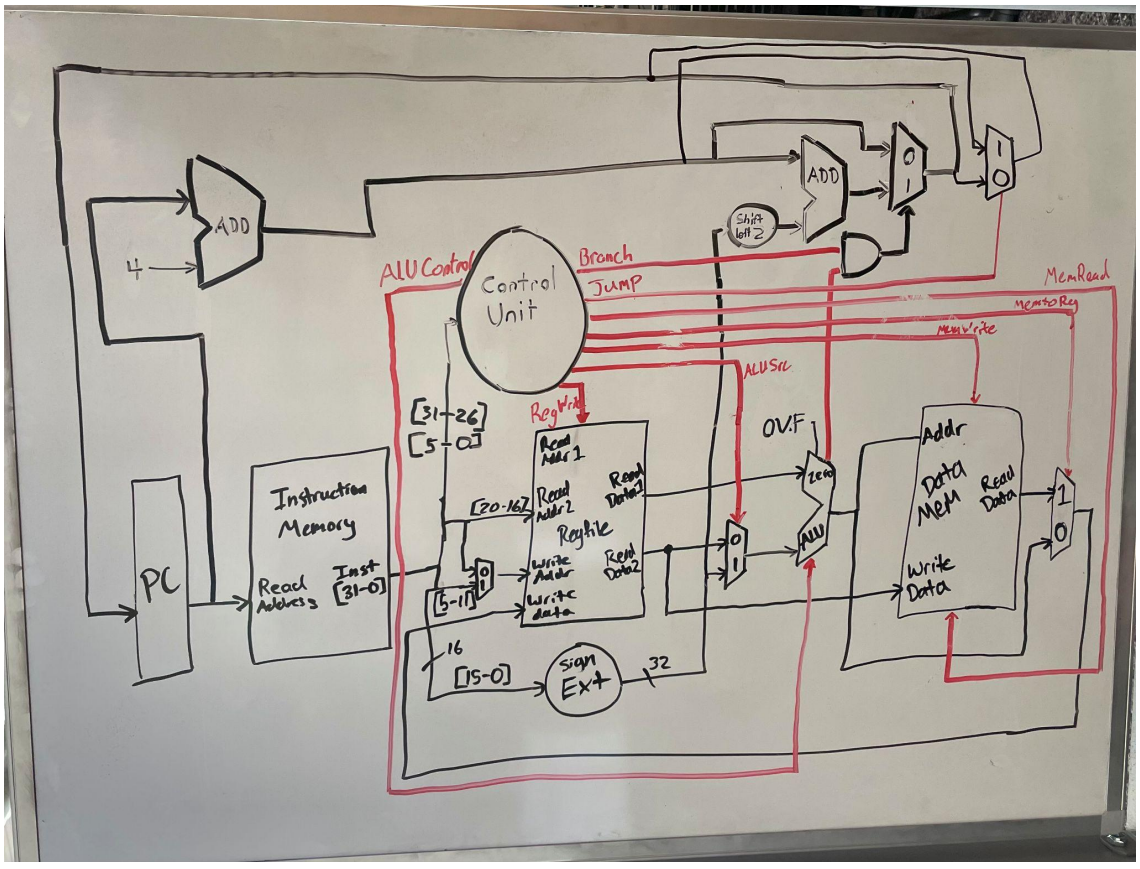
_____ Landon Gulotta _____

_____Andrew Gooding_____

Project Teams Group #:____2___

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

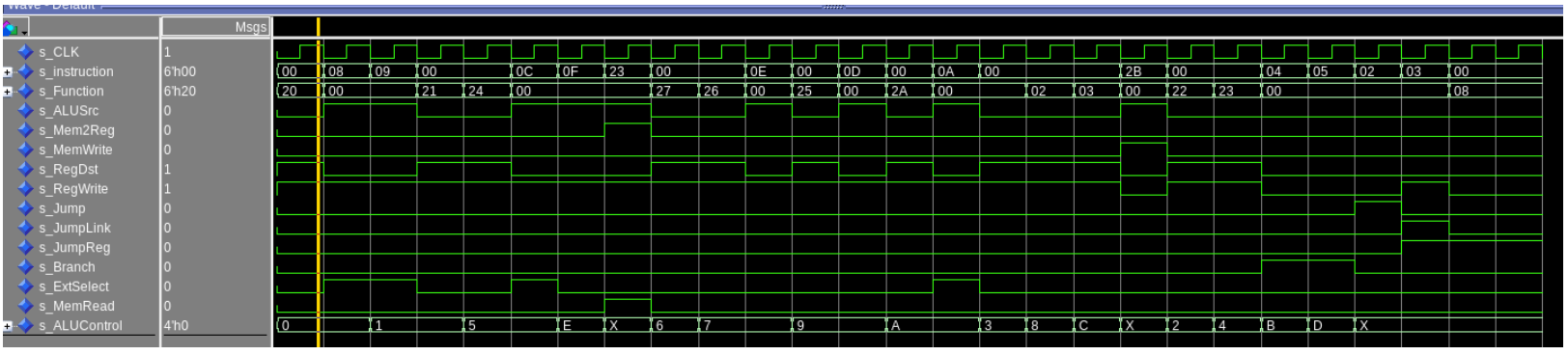
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[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.

				Control Signals												
Instruction	Opcode (Binary)	Funct (Binary)	ALUSrc	ALUControl	MemtoReg	s_DMemWr	s_RegWr	RegDst	Jump	JumpLink	JumpReg	Branch		ExtSelect		MemRead
add	"------"	"100000"	0	"0000"	0	0	1	1	0	0	0	0		0		0
addi	"001000"	"------"	1	"0000"	0	0	1	0	0	0	0	0		1		0
addiu	"001001"	"------"	1	"0001"	0	0	1	0	0	0	0	0		1		0
addu	"------"	"100001"	0	"0001"	0	0	1	1	0	0	0	0		0		0
and	"------"	"100100'	0	"0101"	0	0	1	1	0	0	0	0		0		0
andi	"001100"	"------"	1	"0101"	0	0	1	0	0	0	0	0		0		0
lui	"001111"	"------"	1	"1110"	0	0	1	0	0	0	0	0		1		0
lw	"100011"	"------"	1	-	1	0	1	0	0	0	0	0		0		1
nor	"------"	"100111"	0	"0110"	0	0	1	1	0	0	0	0		0		0
xor	"------"	"100110"	0	"0111"	0	0	1	1	0	0	0	0		0		0
xori	"001110"	"------"	1	"0111"	0	0	1	0	0	0	0	0		0		0
or	"------"	"100101"	0	"1001"	0	0	1	1	0	0	0	0		0		0
ori	"001101"	"------"	1	"1001"	0	0	1	0	0	0	0	0		0		0
slt	"------"	"101010"	0	"1010"	0	0	1	1	0	0	0	0		0		0
slti	"001010"	"------"	1	"1010"	0	0	1	0	0	0	0	0		1		0
sll	"------"	"000000"	0	"0011"	0	0	1	1	0	0	0	0		0		0
srl	"------"	"000010"	0	"1000"	0	0	1	1	0	0	0	0		0		0
sra	"------"	"000011"	0	"1100	0	0	1	1	0	0	0	0		0		0
sw	"101011"	"------"	1	-	0	1	0	0	0	0	0	0		0		0
sub	"------"	"100010"	0	"0010"	0	0	1	1	0	0	0	0		0		0
subu	"------"	"100011"	0	"0100"	0	0	1	1	0	0	0	0		0		0
beq	"000100"	"------"	0	"1011"	0	0	0	0	0	0	0	1		0		0
bne	"000101"	"------"	0	"1101"	0	0	0	0	0	0	0	1		0		0
j	"000010"	"------"	0	-	0	0	0	-	1	0	0	0		0		0
jal	"000011"	"------"	0	-	0	0	1	-	0	1	1	0		0		0
jr	"------"	"001000"	0	-	0	0	0	-	0	0	1	0		0		0

[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).

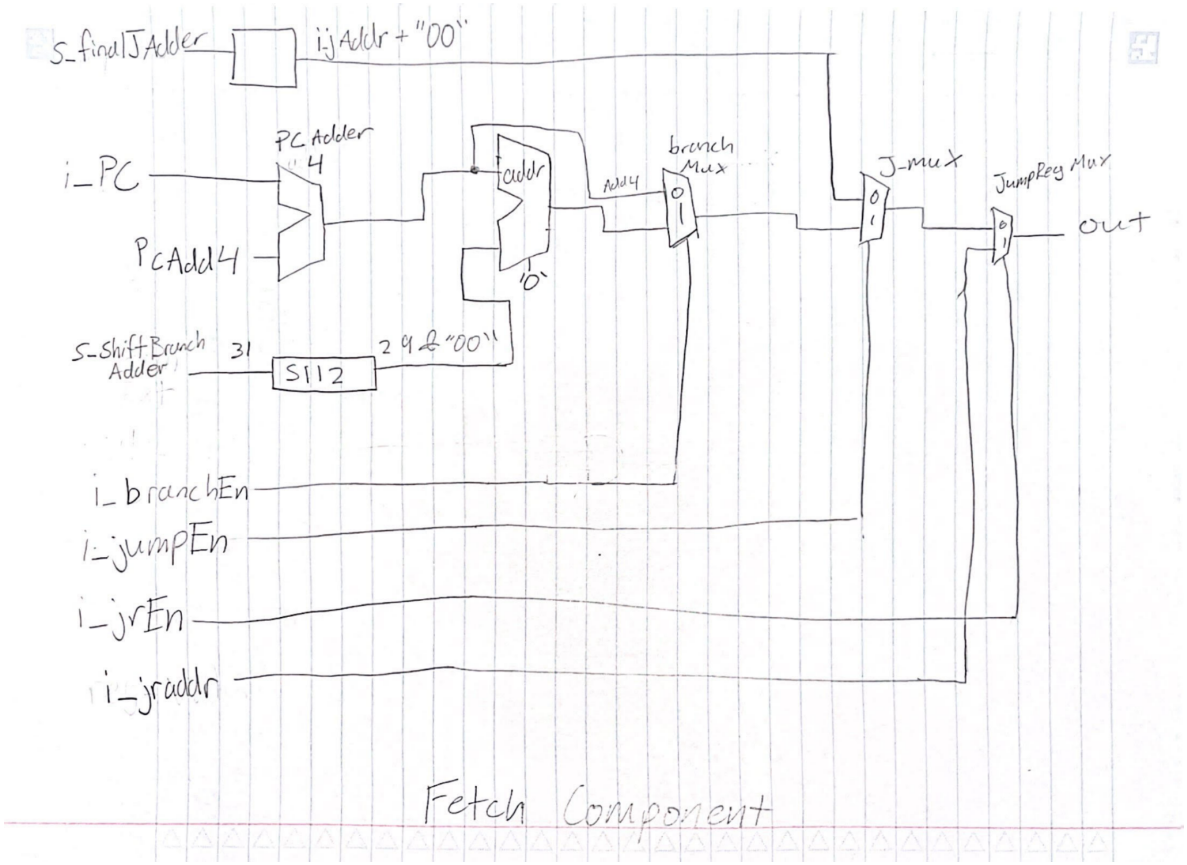


[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

In order to support the fetch logic we must implement the required control flow possibilities. We need more control bits to be able to enable branch, jump, and jump register muxes. All of these new signals will be set by the control component and will set their respective muxes. Branch requires a specific ALU output that is fed through an and gate and inputted into the mux to enable it or not. The jump and jump register are unconditional and can be called by the Control Unit whenever there is an instruction that uses them. The control unit will signal one of the respected muxes in the fetch logic depending on the instruction that is given to it.

For example if we are using a branch equal to instruction then the branch select signal will be the one fired to 1 and send its data into the branch mux and output its value. If we were to send a jump and link instruction, then the jump and jump and link signal will both be selected as 1. This is because the logic says to jump first, then that output is sent to the jump and link mux which is then told by the jump and link signal to output the value in the 1's place. This logic is the same for most of the fetch signals, the muxes are selected from the control component and then their output is sent back to the 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?



The addition control signals that are needed are:

- i_branchEn - to enable the branch mux
- i_jumpEn - to enable the jump mux
- i_jrEn - to enable the jump register mux
- i_jrAddr - the jump address for the jumbo register
- s_shiftBranchAddr - takes the branch address and shifts it left logically by 2 which then becomes an input to the adder.
- s_finalJAdder - if the jump register is enabled it takes this value and shifts it left by 2 bits to get the address for the jump.

All of these enable inputs come from the control unit. Their value will be changed based on what the control unit fires based on the instructions.

[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.

Description:

As we can see in the waveforms, after every instruction pcOut gets incremented by 4 to get to the next instruction. Then when the jump register enable (jrEN) gets a 1 input, he pcOut gets the value of the jump register address (jrAddr) which in this test case is all F's. Also when the branch enable is set to 1, the pcOut should let the branch address add to it. We can see that this is true in our case because the branch address is adding an instruction which makes pcOut increment by 4 bits. I also have all the signals to enable branch, jump, jump register, and jump and link. These will be fired by the control unit to select their respected muxes. With this logic all put together the fetch logic works without errors and can handle any of the fetch instructions.

[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 the bits to the right and will always shift in zero.

sra - Shifts the bits to the right but will shift in the same value of the most significant bit.

MIPS does not have a sla instruction because when you are shifting left you are shifting from the zero element which should always begin as zero.

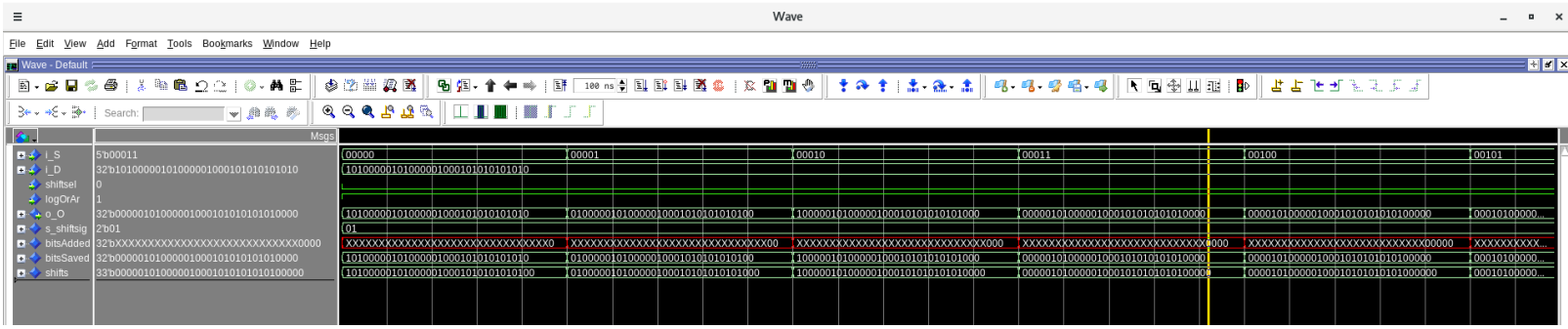
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

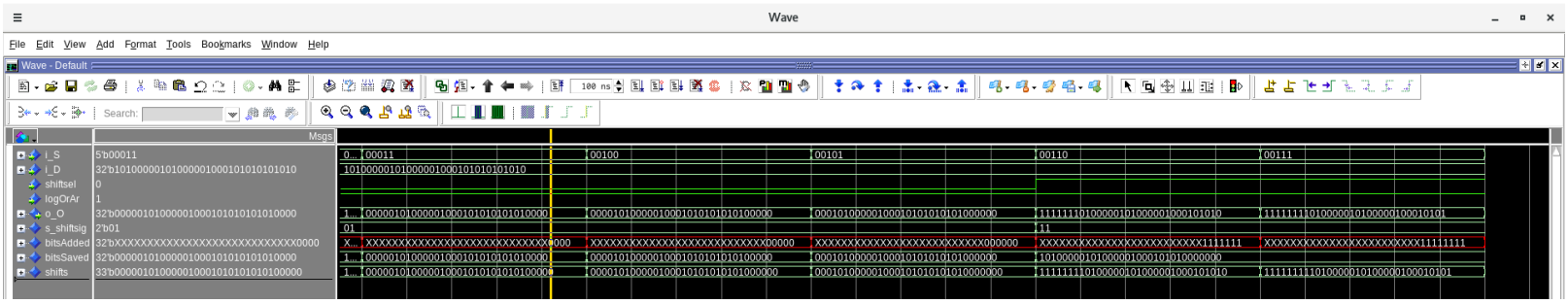
In our VHDL code we implement both arithmetic and logical shifting operations by adding a signal to our barrel shifter that if 0 then shifts logical and if it is one the shift arithmetic.

[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 right barrel shifter can be modified to support shifting left by adding a signal that will determine which way we are shifting.

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





Fifth Test Case: Barrel shifter shifts 4 bit(i_S) to the left(shiftsel) so the output(o_O) shows 4 zeros added at the end and a 1,0,1,0 removed from the front.

Seven Test Case: Barrel shifter shifts 6 bits(i_S) to the right(shiftsel) and adds 6 ones to the front since our logOrAr is 1 and removes 1,0,1,0,1,0

[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.

Design Approach: For the design of our barrel shifter we shift in increments of 1, 2, 4, 8, and 16 bits based on our "i_S" which is our shift amount and our "shiftsel" and "logOrAr" inputs. We create five different signals that will store our shift outputs(o_one, o_two, o_three, o_four, o_five) which are the shifted versions of the input. For each signal we check if i_S is one at the corresponding bit and if our shiftsel and logOrAr is one we use this to decide which way we will shift, how far we will shift and if we are shift logically or arithmetically. Each signal is set the the previous signal after it has been shifted. We then set our o_O to the value of o_five is our last shifted signal.

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

Test Case 1: shift left zero bits

Test Case 2: shifting left 1 bits

Test Case 3: shifting left 2 bits

Test Case 4: shifting left 4 bits

….

Test Case 7: shift right arithmetically 6 bits

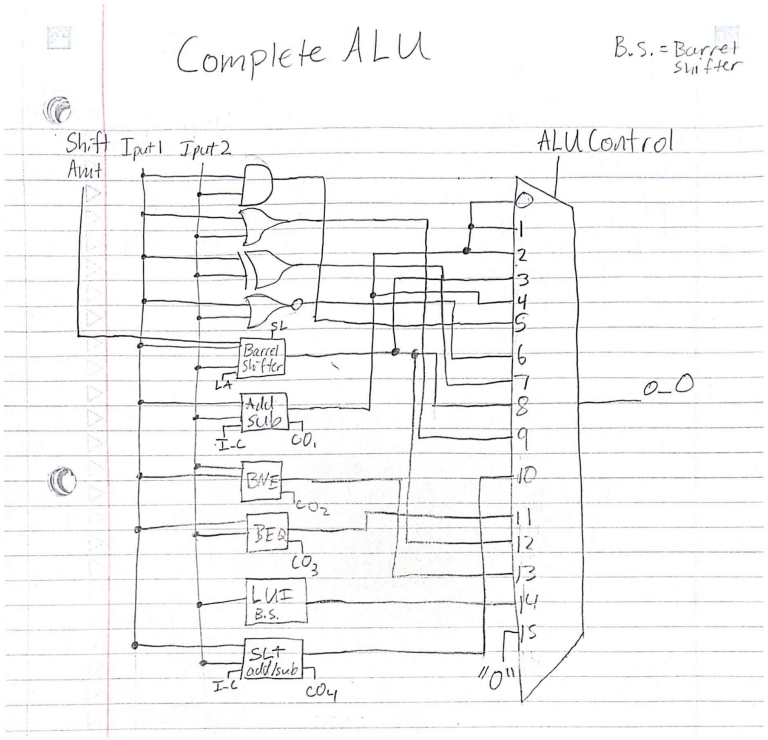
Test Case 8: shift right arithmetically 7 bits

Test Case 9: shift right arithmetically 8 bits

Test Case 10: shift right arithmetically 9 bits

Test Case 11: shift right arithmetically 10 bits

[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: We calculate overflow by creating a signal consisting of our alucontrol, last bit of our first input, last bit of our second input, our aluSig signal, and the last bit of the output to our addsub component.

Zero: To calculate the zero output we create a signal beqAbne which is a concatenation of the first bit of our beq component output, first bit of our bne output and our alu control which will determine with our zero output is 0 or 1.

SLT: For created a signal that would be the output of our slt component and we check if the last bit is 1 or zero which will decide what we are sending into our mux.

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

And: For our AND operation we are just ANDing the two inputs we get for our ALU.

Or: For our OR operation we are just ORing the two inputs we get for our ALU.

Xor: For our XOR operation we are just XORing the two inputs we get for our ALU.

Nor: For our NOR operation we are just NORing the two inputs we get for our ALU.

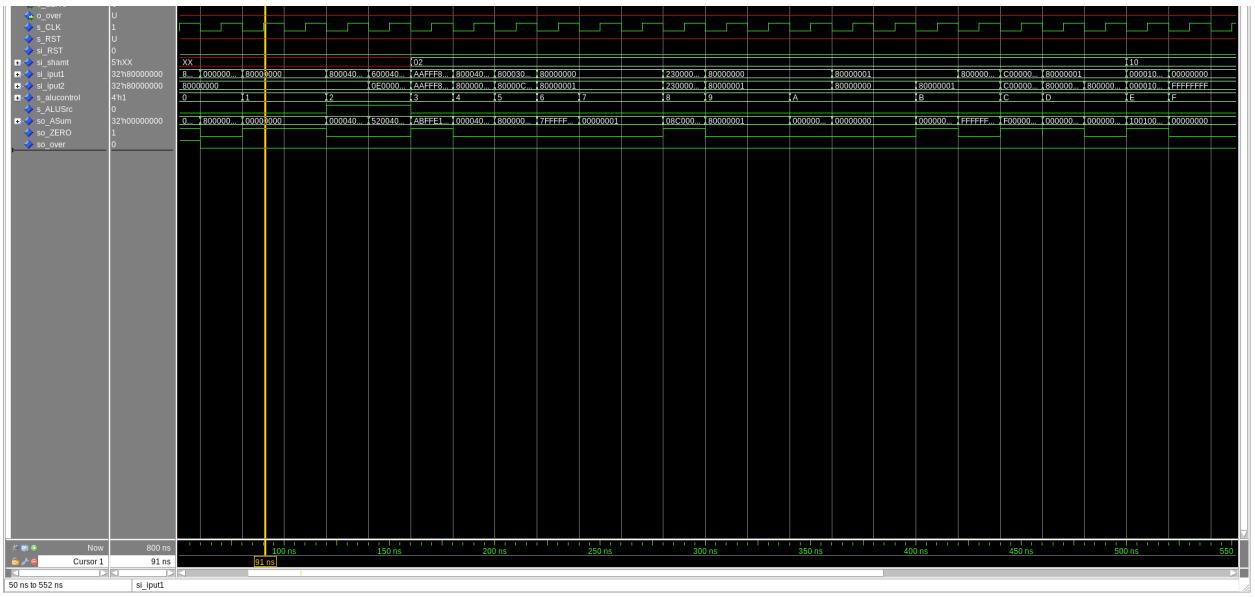
SLT: For our SLT component we subtract the two inputs and if the value is negative we return a zero and if it is positive we return a one.

SLL: We use our barrel shifter to decide if we are going to shift left and add the correct number of zeros depending on how much our shift amount is.

SRL: We use our barrel shifter to decide if we are going to shift right and add the correct number of zeros depending on how much our shift amount is.

SRA: SRL: We use our barrel shifter to decide if we are going to shift right and if it is an arithmetic shift or not then it will add the correct number of zeros or ones depending on how much our shift amount is and the last bit of our second input.

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

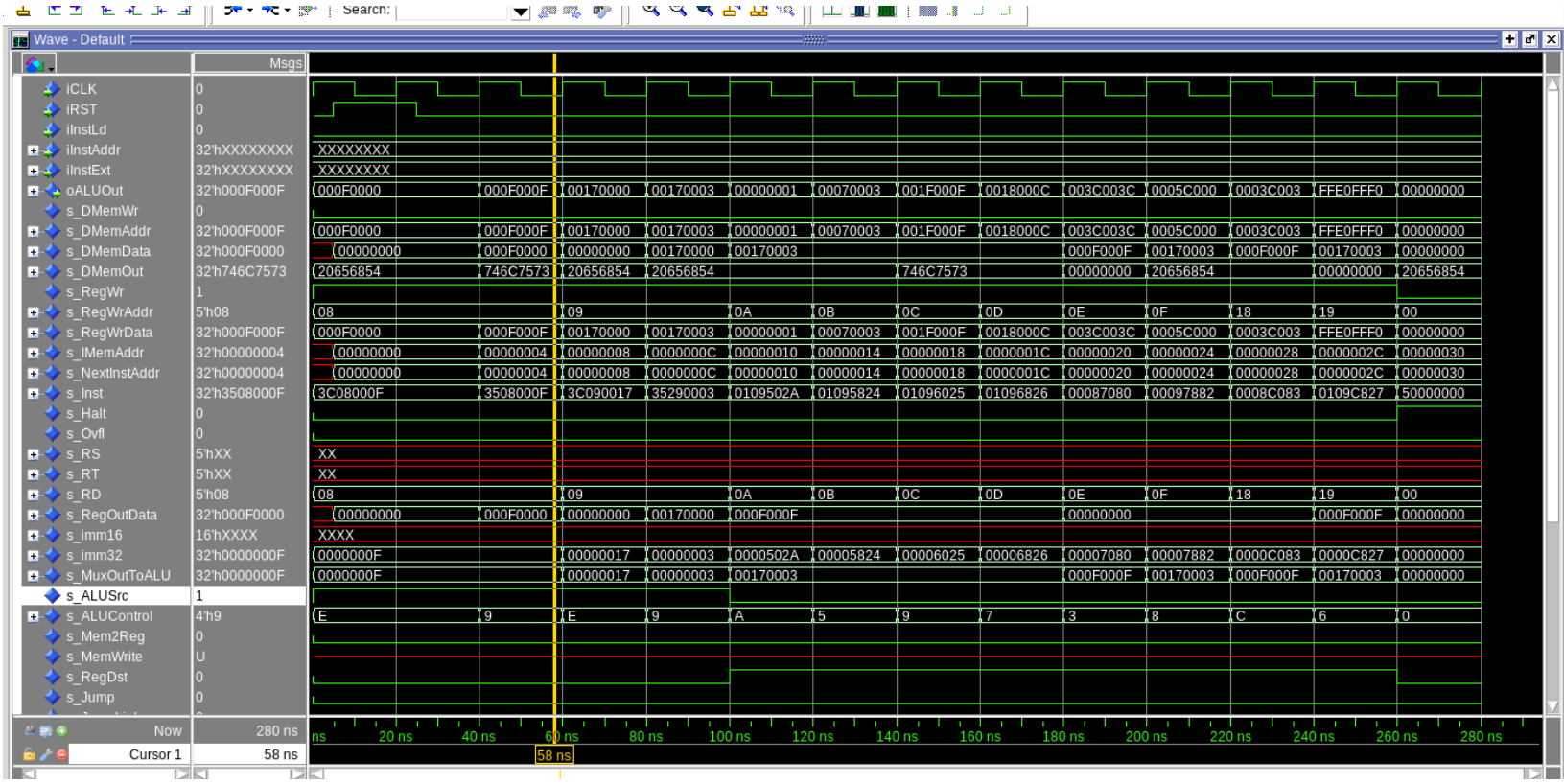


The test plan is comprehensive because it uses every instruction that we need to implement with edges to test when overflow occurs in the add, addi, and sub instructions. We also test the beq and bne heavily to run test cases for our zero output. For our shifter we tested multiple shifts of logic and arithmetic. We also test each one of these parts individually and they all can be out successfully.

[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.

Waveforms and tests below.

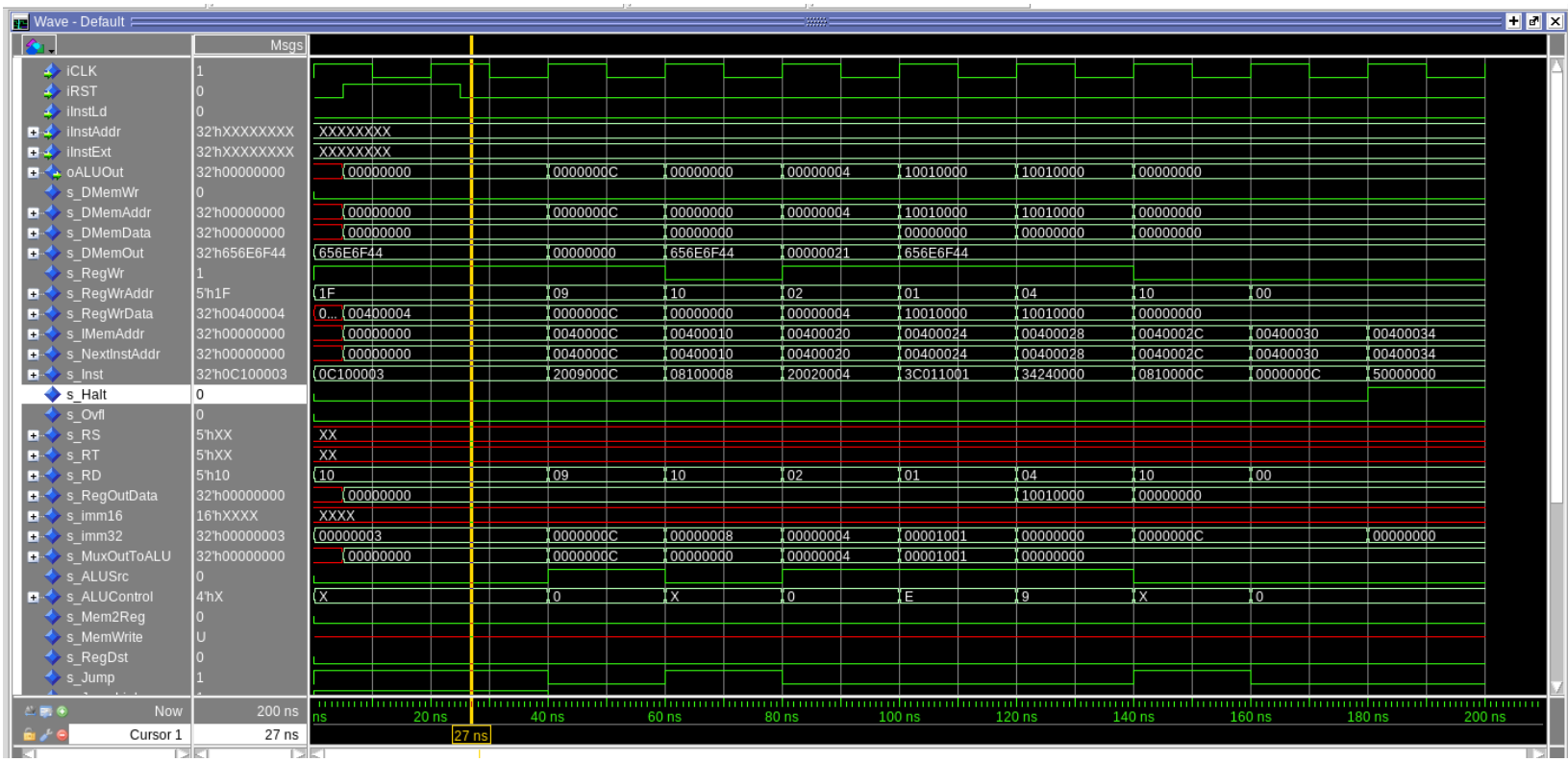
[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially,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

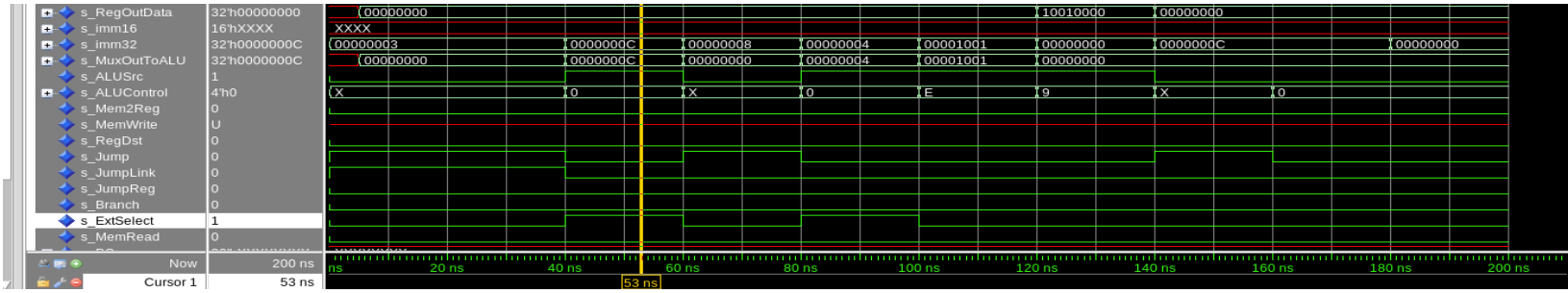


For this base test, we use the instructions: lui, ori, slt, and, or xor, sll, srl, sra, nor, and halt. In the first instruction, lui $t0, 0x000F, we can see that the ALU control is set to "1110", the same as in the spreadsheet. It uses $t0, the 8th reg from the s_RD signal. It stores 0x000F, which we can see in the imm signal. In the s_RegOutData, we can see that it loads the upper 16 bits of the immediate value, "000F0000". In the next cycle the instruction is "ori $t0, $t0, 0x000f". You can see from the ALUControl that ori is being used because the control outputs 9, which is "1001", the same as in the spreadsheet above. These instructions ori with the register used before, so it should output "000F000F", which is shown in s_RegWrData. The Next two instructions are the same instructions with different values. These instructions ori the upper imm value of 0x0017 and "0x0003" register $t1, seen as the nine value in the s_RD. As you can see in the 4th cycle, the output is "00170003". For the next test, we are using other bitwise operations. Each instruction uses the values of $t0, $t1, or some form of an immediate value. In the 5th cycle, we can see the slt instruction working by looking at the ALUOut. It is outputting "1" because $t0 < $t1. The next instruction is the "AND" function, which "ANDs" the two values. We can see that in the ALUOut, the output is 00070003, which is the correct value if you AND both values. The next instruction is the OR, seen in the 7th cycle. ALUOut sees this instruction as the value

"001F000F". The 8th cycle uses XOR instruction, and this works because by looking at the ALUout, it can be seen as the correct output of "0018000C". By looking at the ALUOut for the output of each bitwise operation for sll, srl, sra, and nor, we can see that each has its correct value. For the next cycle, they are: $t6 = $t0 shifted left logical by 2 bits, $t7 = $t1 shifted right logical by 2 bit, $t8 = $t0 shifted right arithmetic by 2 bits, $t9 = NOR of $t0 and $t1. For the last instruction, we are halting the system. It works by looking at the halt signal and outputting "1". These cycles can work correctly if we look at the ALUOut signal.

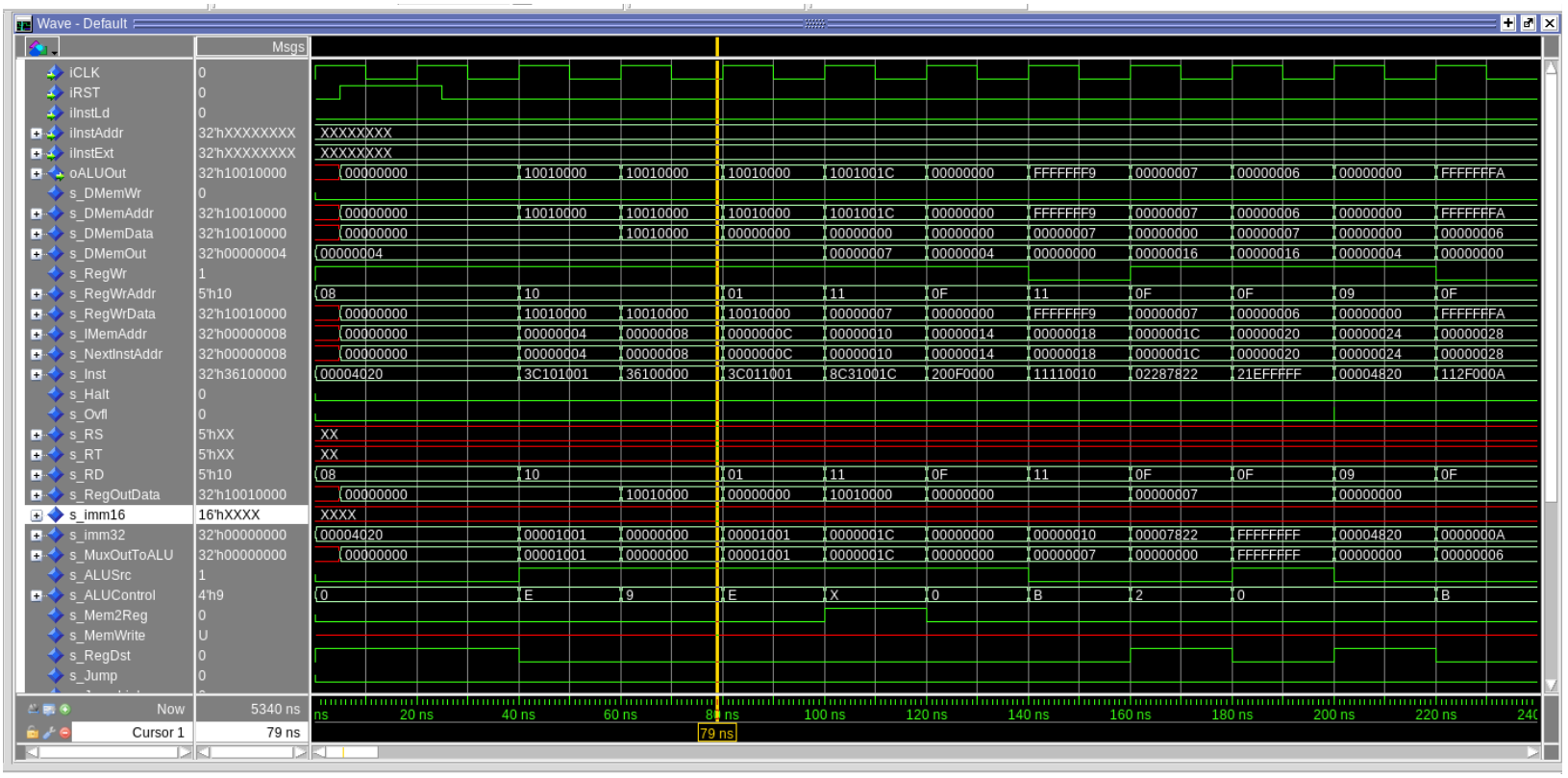
[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

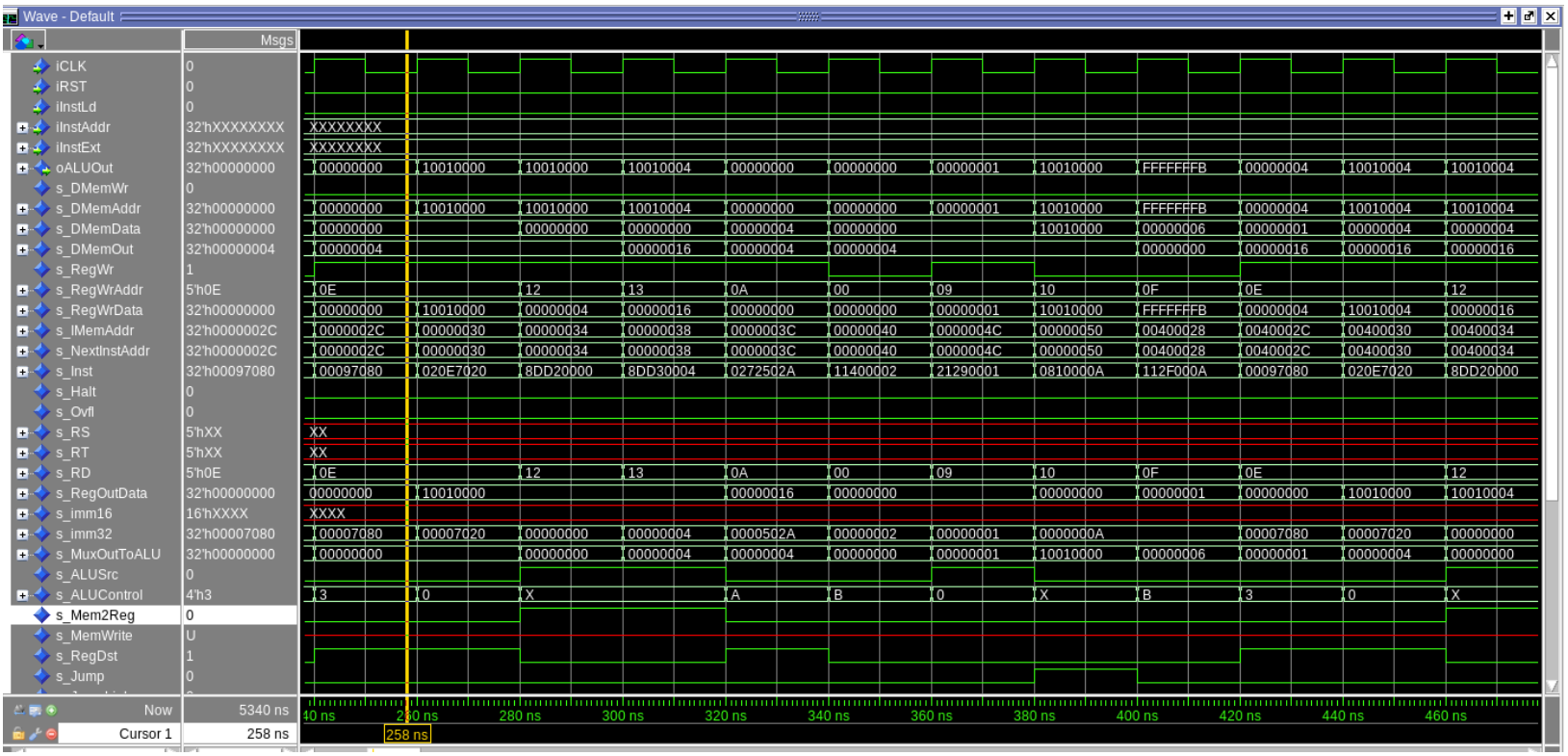




- The cf test was made to test beq, bne, j, jal, and jr instructions respectively. The test starts by jumping and linking to the first loop. In the waveform you can see that the RegWrAddr or the register write address is jumping from 0x0 to 0x40000C to get to the jump address. Then we are using an addi instruction to load 12 into register $t1, which can be seen from the oALUout signal. It then jumps to another loop which we can see when the IMemAddr changes to 0x4000010 to 0x4000020. We then make the new jump register to be 0 and the instructions continue. Finally register $t1 is compared to register $t2 with a bne instruction and if they are equal then the program finishes up and halts. In the second screenshot you can see how the control unit fires signals to each of the respected instructions. As you can see it starts off with jump and link and continues from there. This test tested our bne, beq, jump, jump and link, and jump register instructions to which they all passed the cases and got the correct values.

[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.





Based on the link provided, BubbleSort is a simple sorting algorithm that iterates through an array multiple times, comparing adjacent elements and swapping them if they are in the wrong order. This process continues until no more swaps are needed, indicating that the array is fully sorted. In the bubble sort test, we have 7 numbers in a list: 4, 22, 18, 1, 9, 10, 11.

We coded it where it sorts an array stored in memory by comparing adjacent elements and swapping them if they are out of order. The outer loop, controlled by registers $t0 and $s1, iterates through the list. The inner loop, controlled by registers $t1, $t6, $s0, $s2, and $s3, performs the actual comparisons and swaps. When two elements need to be swapped, the sw instructions rearrange their positions. The process repeats until the entire list is sorted, with sorted elements "bubbling up" to their correct positions.

For the first main steps before the two loops we have the instructions of, add $t0, $zero, $zero → lui $s0, 0x1001 → ori $s0, $s0, 0x0000 → lw $s1, size → addi $t7, $zero, 0. These can be seen in the first 5 cycles of the waveforms. The ADD instruction should add 0 into the $t0, the second instruction loads the upper immediate of 0x1001, ori OR register $s0 with an immediate value of 0x0000, lw loads into register $s1 with the value of size 7, and the addi puts 0 into register $t7. As we can see in the ALUOut signal, the outputs are correct based on the first five instructions.

The next part of the code and waveform is the outer loop of the bubble sort. The next instruction is Beq, which determines if the $t0 and $s1 are equal and then moves to the exit. We see that the ALUOut is equal to 0, meaning it is not equal and does not jump to the exit. The next three instructions are sub, addi, and add. We can see that these instructions work. For example, the sub should be 7-0, and the ALUOut outputs a value of 00000007. The next two outputs should be 7-1 and 0, and the ALUOut is 00000006 and 00000000 in this order. After following every cycle, you can see that the bubble sort works by looking at the ALUout output for every cycle and matching it up with the instructions from the bubble sort. You can find the correctness of each instruction following these instructions:

```
 outer_loop: beq     $t0, $s1, exit
        sub   $t7, $s1,     $t0
        addi  $t7, $t7,     -1
        add   $t1, $zero,  $zero
inner_loop: beq     $t1, $t7,     continue
        sll   $t6, $t1,     2
        add   $t6, $s0,     $t6
        lw    $s2, 0($t6)
        lw    $s3, 4($t6)
        slt   $t2, $s3,     $s2
        beq   $t2,  $zero,  sorted
        sw    $s3,  0($t6)
        sw    $s2,  4($t6)
sorted: addi  $t1,  $t1,  1
        j     inner_loop
continue: addi $t0,  $t0,  1
        j   outer_loop
exit: halt
```

[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 that our processor can run at is 25.48mhz.

There are 2 major components that could be improved to get a better frequency.
CompleteALU - The complete ALU took around 6ns to get the output which was the most of any of the components besides the memory components. Improving this would improve the maximum frequency that our processor can run at. This component used a lot of time to traverse into each component that it uses. First it starts at the ALU then goes into the adder/subtractor, next it goes into the adder then or gates and finally gives an output.

RegisterFile - The register file took the second longest of the components that we created to get through. It took about 5.5ns to get the output which could be improved. This component took a while because it had to traverse through a bunch of muxes to get the output, which was expected. To improve this we could design a way for the register file to not use as many multiplexores to get the correct output.

By focusing on improving these 2 components we can improve the maximum frequency of our processor. Out of all the components that we created for this processor, these would be the 2 to improve on to make our processor run faster.