CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

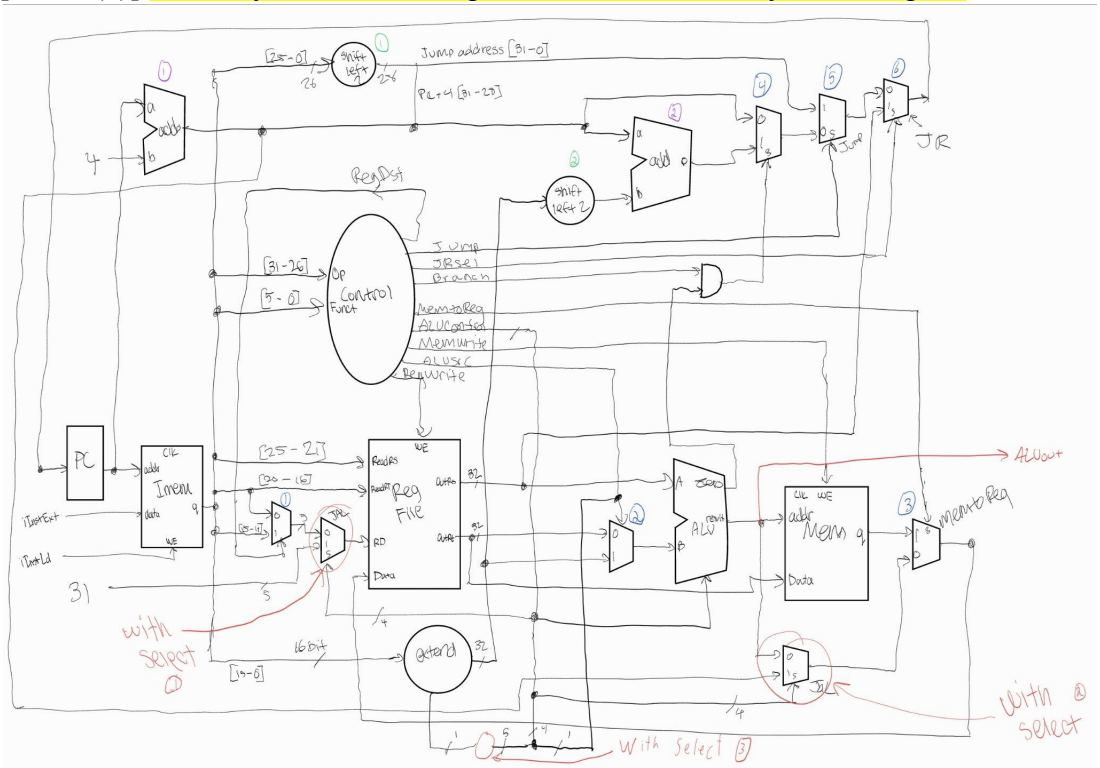
Team Members: Nathan Reff

Logan Roe

Project Teams Group #: 5-7

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

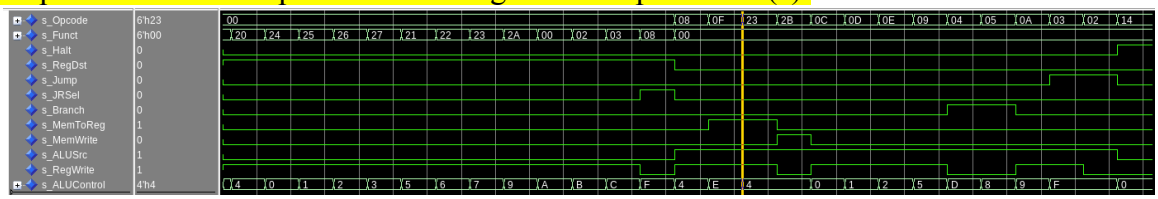
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[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.

	s	Opcode (Binary)	Funct (Binary)	Control Signals									
				Halt	RegDst	Jump	JRSel	Branch	MemtoReg	ALUControl	s_DMemWr	ALUSrc	s_RegWr
3	halt	"10100"	"------"	1	0	0	0	0	0	"0000"	0	0	0
4	and	"000000"	"100100"	0	1	0	0	0	0	"0000"	0	0	1
5	andi	"001100"	"------"	0	0	0	0	0	0	"0000"	0	1	1
6	or	"000000"	"100101"	0	1	0	0	0	0	"0001"	0	0	1
7	ori	"001101"	"------"	0	0	0	0	0	0	"0001"	0	1	1
8	xor	"000000"	"100110"	0	1	0	0	0	0	"0010"	0	0	1
9	xori	"001110"	"------"	0	0	0	0	0	0	"0010"	0	1	1
10	nor	"000000"	"100111"	0	1	0	0	0	0	"0011"	0	0	1
11	add	"000000"	"100000"	0	1	0	0	0	0	"0100"	0	0	1
12	addi	"001000"	"------"	0	0	0	0	0	0	"0100"	0	1	1
13	lw	"100011"	"------"	0	0	0	0	0	1	"0100"	0	1	1
14	sw	"101011"	"------"	0	0	0	0	0	0	"0100"	1	1	0
15	addu	"000000"	"100001"	0	1	0	0	0	0	"0101"	0	0	1
16	addiu	"001001"	"------"	0	0	0	0	0	0	"0101"	0	1	1
17	sub	"000000"	"100010"	0	1	0	0	0	0	"0110"	0	0	1
18	beq	"000100"	"------"	0	0	0	0	1	0	"1101"	0	1	0
19	subu	"000000"	"100011"	0	1	0	0	0	0	"0111"	0	0	1
20	bne	"000101"	"------"	0	0	0	0	1	0	"1000"	0	1	0
21	slt	"000000"	"101010"	0	1	0	0	0	0	"1001"	0	0	1
22	slti	"001010"	"------"	0	0	0	0	0	0	"1001"	0	1	1
23	sll	"000000"	"000000"	0	1	0	0	0	0	"1010"	0	0	1
24	srl	"000000"	"000010"	0	1	0	0	0	0	"1011"	0	0	1
25	sra	"000000"	"000011"	0	1	0	0	0	0	"1100"	0	0	1
26	lui	"001111"	"------"	0	0	0	0	0	1	"1110"	0	1	1
27	jal	"000011"	"------"	0	X	1	0	0	0	X - 1111	0	1	1
28	jr	"000000"	"001000"	0	1	X	1	0	0	X - 1111 - don't c	0	0	0
29	j	"000010"	"------"	0	X	1	0	0	0	X - 1111	0	1	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).

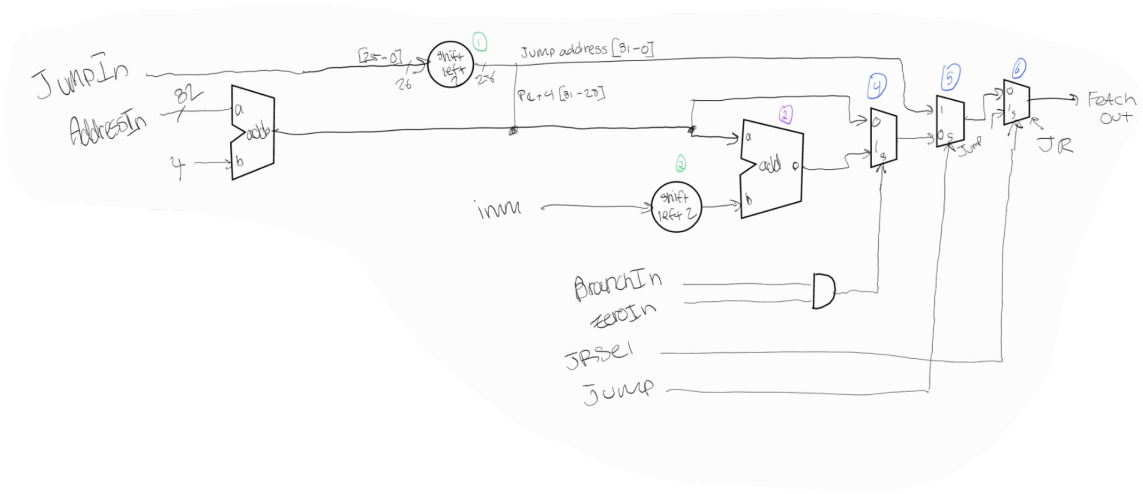


the waveform goes through all the scenarios shown in the spreadsheet. The first test case is opcode: 00 = “000000” and function code 24 = “100100” for the actions “and, andi”. When this is the case, all the outputs halt, regDst … ALUControl are equal to the expected outputs in the excel sheet. The waveform matches the excel sheet in all instances, therefore this module works properly.

[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.

There are four different possibilities for changing the address. This can be done via a simple PC + 4 which is just incrementing to the next instruction, a jump, which uses a 26-bit value to jump to a specified location, a jump register, which uses a 32-bit value to jump to a specific location, and a branch which uses a 16-bit value to jump to a specific label based upon the outcome of the beq/bne.

[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 required would be JumpIn, AddressIn, and FetchOut. The other signals shown here are already assigned to signals as they are used elsewhere.

[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.

We did not create a separate fetch logic block, we just implemented this straight into the top level processor. Due to the processor's performance, we can tell that it works as intended in all situations, but we do not have any waveforms to resemble this due to it not being its own block.

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

Srl is shift right logical which just shifts the bits by a defined shift amount to the right. It does this by using zero extension at all times. Sra is shift right arithmetic which shifts the bits by a defined shift amount to the right, but uses sign extension instead of zero extension. MIPS does not have an sla instruction because that would potentially be extending ones into the least significant bits which serves no real purpose.

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

When shifting logical, the extra bits become zero (zero-extended) while shifting arithmetic, the extra bits become the same value as the last bit (sign extended). For both we used the line:

```
o_Out(N-1-s_ShamtInt downto 0) <= i_In(N-1 downto s_ShamtInt);
```

setting the new output shifted value to the original input downto the shift amount.
The next line:

```
o_Out(N-1 downto N-s_ShamtInt) <= (others => x);
```

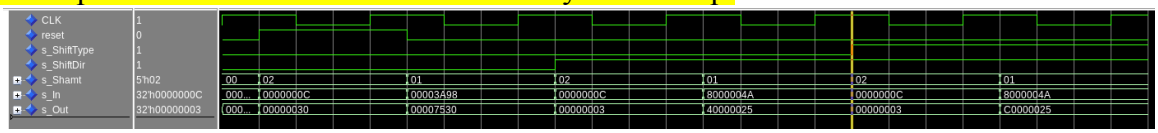
where x is either '0' for logical and i_In(N-1) for arithmetic

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.
In order to get the right barrel shifter to also support left shifting operations, an input that holds the value of the shift direction (0 representing left, 1 right) needs to be added. With that, an if statement can be added to check if it is 0 or 1. If it is 0, left shift, then perform the following lines:

```
o_Out(N-1 downto s_ShamtInt) <= i_In(N-(1 + s_ShamtInt) downto 0);
o_Out((s_ShamtInt - 1) downto 0) <= (others => '0');
```

This does the shift and then assigns 0's to the rest of the values in the least significant bits. If the shift direction is 1, right, then it will simply perform the actions described in the previous question.

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

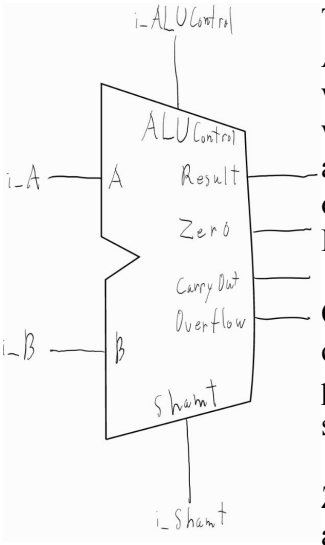


In our code we had two inputs: shiftType and shiftDir; when shiftType is 0, then logical and when 1, then arithmetic. when shiftDir is 0, then left and when shiftDir is 1, right.
test case 1: sll 2 (0000000C) expected output: 00000030
test case 4: srl 1 (8000004C) expected output: 40000025
test case 6: sra 1 (8000004a) expected output: C0000025

[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.
In order to determine how we would design the ALU, we first thought about all of the functionality that it has to have. From there, we drew a rough sketch of what it *could* look like and, through some discussion and redrawing, we came to a final rough sketch that showed all of the functionality and the design we wanted. We concluded by calculating each potential output every time and then using a with select to determine which of the calculated values would be the correct value for this instance by using the ALUOp value.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.
Due to how we implemented the ALU, such as using a with select instead of a larger mux at the end, we do not have any other blocks to show waveforms for that are not already created. However, in relation to the with select at the end, all it does is look at the ALU Control code and select the wanted output based upon this. [Part 2 (c.v)] shows the whole ALU's waveforms which can be looked at to show the functionality of the ALU as a whole works.

[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?



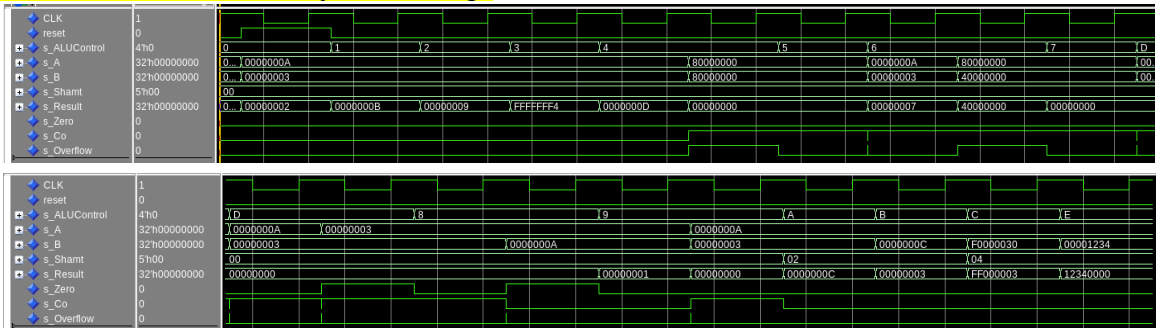
The adder block, pictured on the left, has inputs of ALUControl, A, B, and Shamt. ALUControl is a 4-bit code which denotes which instruction is being performed. A and B are 32-bit inputs which are used for operations. Shamt is the shift amount taken as a 5-bit input, if a shift operation were to take place. The outputs of this are Result, Zero, CarryOut, and Overflow where Result is 32-bit, Zero, CarryOut, and Overflow are 1-bit.

Overflow is calculated by xoring the last carry in and the last carry out values. This will only ever be affected when performing one of the following operations: add, addi, sub, lw, sw, beq, bne, slt.

Zero is calculated only when performing a branch instruction and will only be one when the branch instruction is true.

Slt is implemented by subtracting B from A and then checking the result's most significant bit. If the most significant bit is a 1, then a 1 is outputted, otherwise 0.

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



the first test case tests the and functionality. the alu control input for and is 0, and the two inputs are A and 3, anding A with 3 results in 2, which shows to work properly in the testbench. test case 5 shows the ALUControl input as 4 which corresponds to the adding and functionality; the two inputs are both "80000000". Adding these will result in an overflow and the result will be "00000000." this is shown to be true in the testbench. when the ALUControl input is 9, then the functionality occurring is slt and slti. the first test case for this functionality is 3 < A; the expected output is 1, and the next case is A < 3; the expected output is 0. The testbench proves this to work properly. Along with these test cases the waveforms show the ALU to be properly working for all functionality.

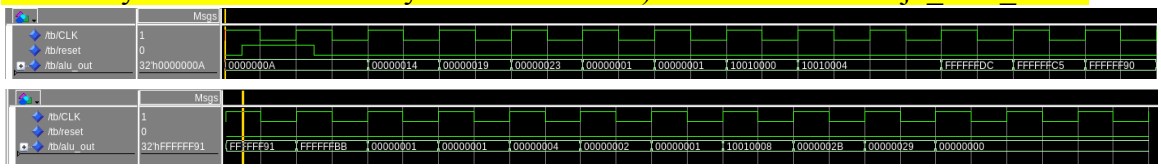
[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

All references made to waveforms correspond to the previous waveforms in [part 2 (c.v)]
The test plan tests every single instruction that it must be able to perform at least once. Using the base result value, it can be seen that the instructions perform as intended. For overflow, multiple instructions are used to show that overflow turns on, when expected,

during signed operations but not during unsigned under the same circumstances. As for carry out, this can be seen to be working in all of the normal operations when expected. Zero is being tested during the beq and bne instructions to show that it turns on when the beq or bne instructions are true and turns off otherwise.

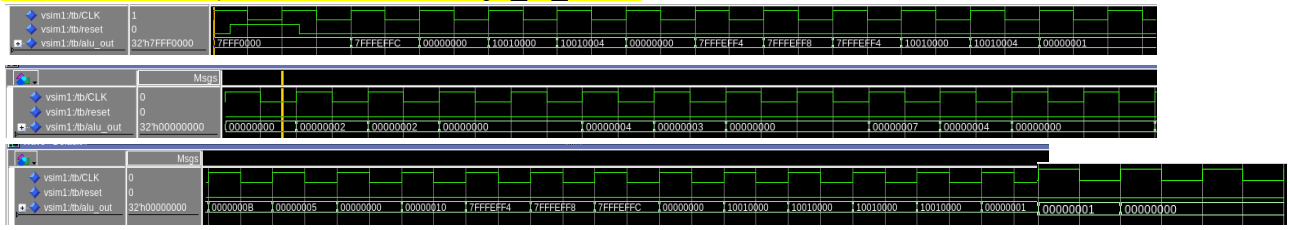
[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.
All of the sections below this, in relation to part 3, we show the waveforms, with explanations, to prove that the processor works as intended in multiple different scenarios.

[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.



With this test, we used at least one example of every instruction that our processor is supposed to support. Further, we also did rewriting of registers/memory, reading from registers/memory, and general writing to registers/memory. With all of this in place, it shows that our processor is capable of performing all of the base level required instructions. For example, in the first instruction, an addi is being tested with $s0 being set to 10, which, as can be seen in the waveform, works as intended. This same idea can be applied to the rest of the instructions as well which shows that all instructions work as intended.

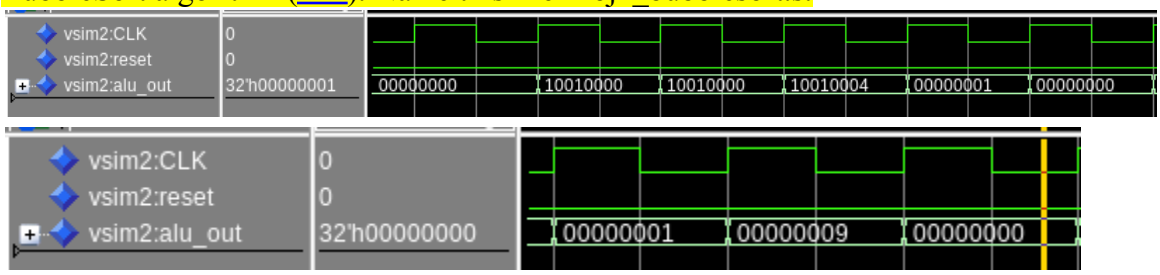
[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.



In this test, we simply used a recursive function that just increments to a certain predefined number (5 in this case) and, as incrementing, it will add the current count value to the previously calculated value. To start, it is initializing some memory and then jumping to the main method. As can be seen in the second instruction, the ALU output is 0x7FFFEFFC which is expected as this is what $sp should be initialized to. Looking at

each individual instruction and the output corresponding to it, it can be seen that the processor is fully tested with control flow logic and works as intended in all cases.

[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.



In the top waveform, a sample of the loop is being shown, which is this code:

```
loop:
        sll $t7, $s1, 2                         #multiply $s1 by 2 and put it in t7
        add $t7, $s7, $t7                       #add the address of numbers to t7

        lw $t0, 0($t7)                          #load numbers[j]
        lw $t1, 4($t7)                          #load numbers[j+1]

        slt $t2, $t0, $t1                       #if t0 < t1
        bne $t2, $zero, increment

        sw $t1, 0($t7)                          #swap
        sw $t0, 4($t7)
```

In the waveform, sll is being run, which should result in 0 due to $s1 being initialized to 0 and this being the first loop, and add should result in the address of numbers (base address of 0x10010000). The first lw should result in loading the first value in numbers and the second should result in loading the second value in numbers. The set less than should result in 1 since $t0 is less than $t1 which then causes bne to branch to increment.

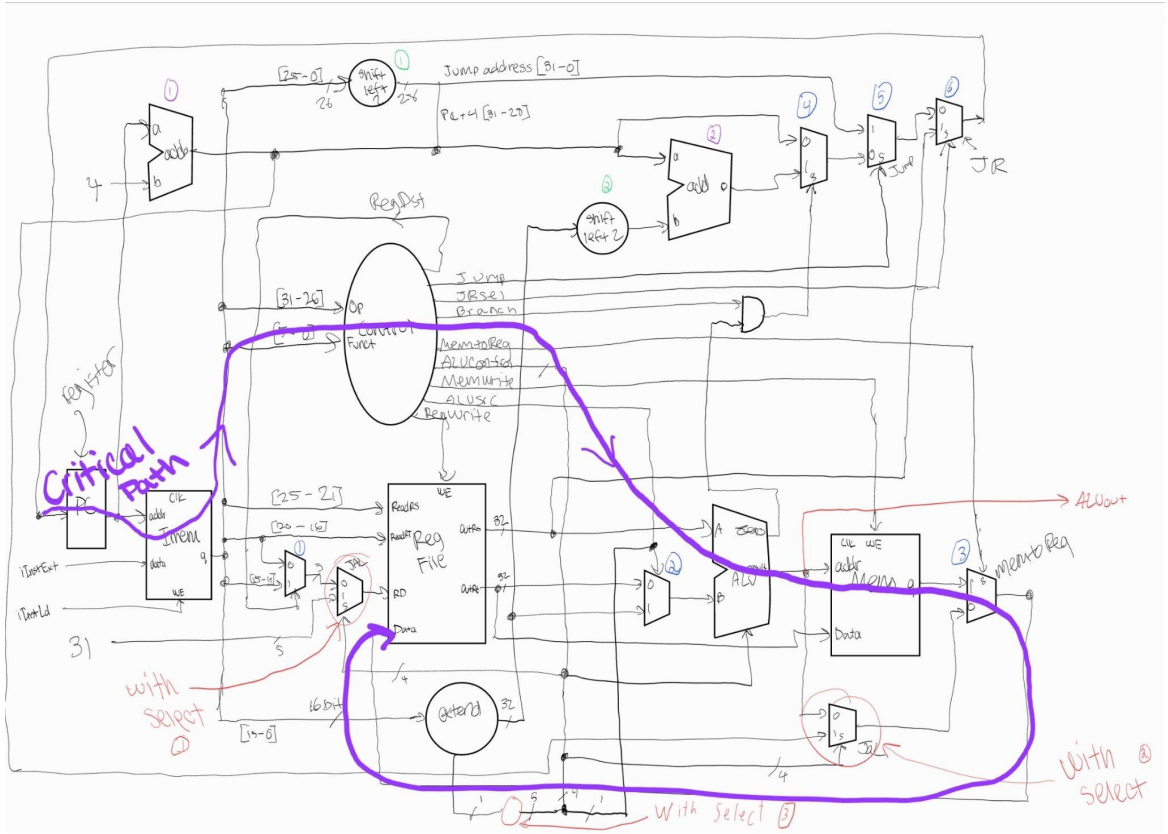
In the bottom waveform, a sample of the increment is being shown, which is this code:

```
increment:

        addi $s1, $s1, 1                        #increment t1
        sub $s5, $s6, $s0                       #subtract s0 from s6

        bne  $s1, $s5, loop                     #if s1 (counter for second loop) does not equal 9, loop
        addi $s0, $s0, 1                        #otherwise add 1 to s0
        li $s1, 0                               #reset s1 to 0

        bne  $s0, $s6, loop                     # go back through loop with s1 = s1 + 1
```

In the waveform, addi is being run as they first instruction, which increments $s1 from 0 to 1, which should result in 1, then sub is being run next which will take $s6, 9, minus $s0, 0, which should result in 9. Next, bne is run and, since $s1, 1, and $s5, 9, are not equal, it will go back and do another iteration of loop.

As can be seen, all of the outputs in the first iteration of the loop and the increment blocks work as intended and the same can be assumed for the rest of the iterations as it results in a proper, sorted array of numbers.

[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?



In the picture above, the critical path is detonated by the purple line which is as follows: PC -> I-Mem -> Control -> ALU -> D-Mem -> Mux (Blue #3) -> Reg File Write

Max frequency (FMax): 24.52mhz
Data Arrival Time: 43.772 ns
Slack: -20.788 ns

In order to improve the frequency the most, the main components that could be improved are the Control Logic unit and the ALU. This can be determined by looking at the increment time of the critical path timing information. In this, the control unit took up about 6.6 ns with two calls taking up 5 ns combined. Due to this, if the control unit were to be improved so that those two calls would be decreased down to the time that it takes for the other calls, around 0.2-0.3 ns, then the overall time could be improved by around 4.5 ns from this alone. As for the ALU, the total time spent on it in the critical path is 9 ns with certain calls taking over the average of about 0.25 ns. Some calls took as much as 0.554 and 0.648 ns. If these were all to be improved, the overall time could be further decreased by about 3-5 ns, depending on how much it is improved. These are the main improvements that could be made but, obviously, minimal improvements elsewhere would also cut down on total run time.