CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

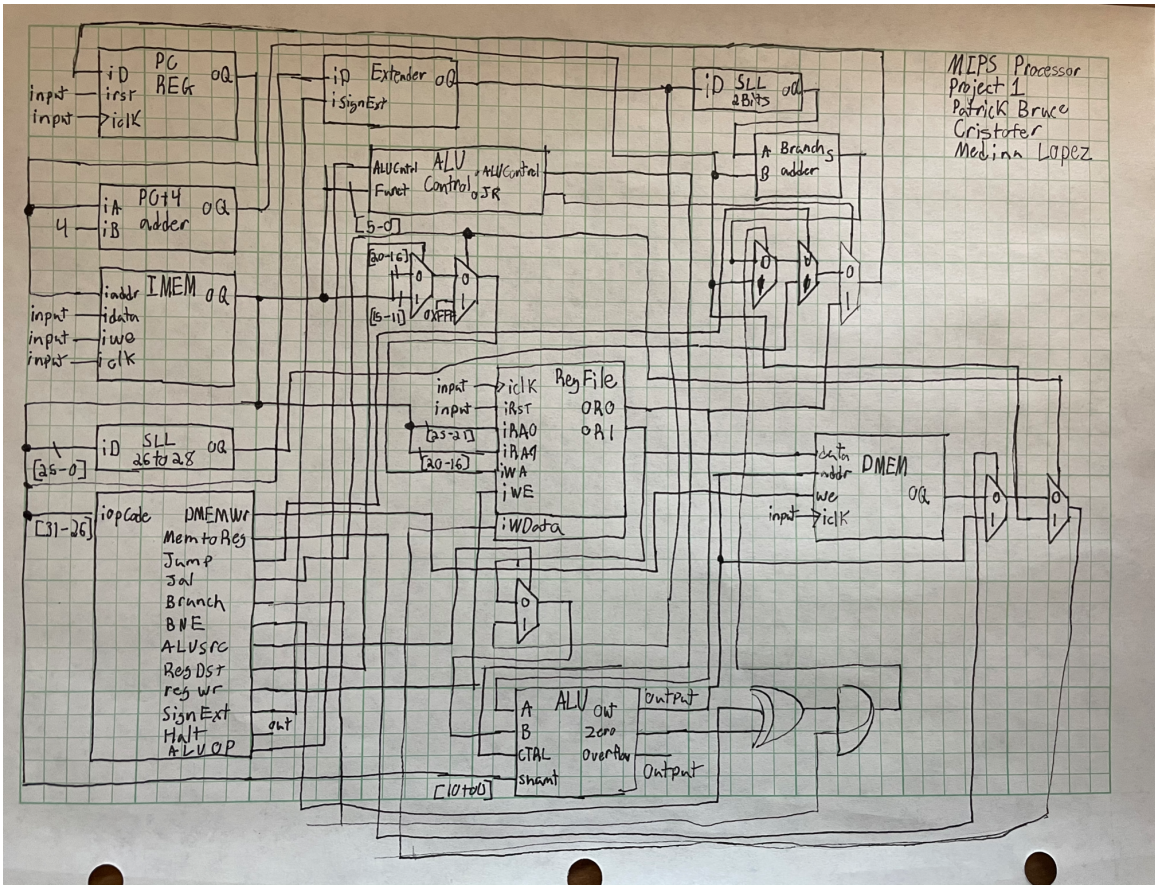
Team Members: Patrick Bruce

Cristofer Medina Lopez

Project Teams Group #: Group 4 Section 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.



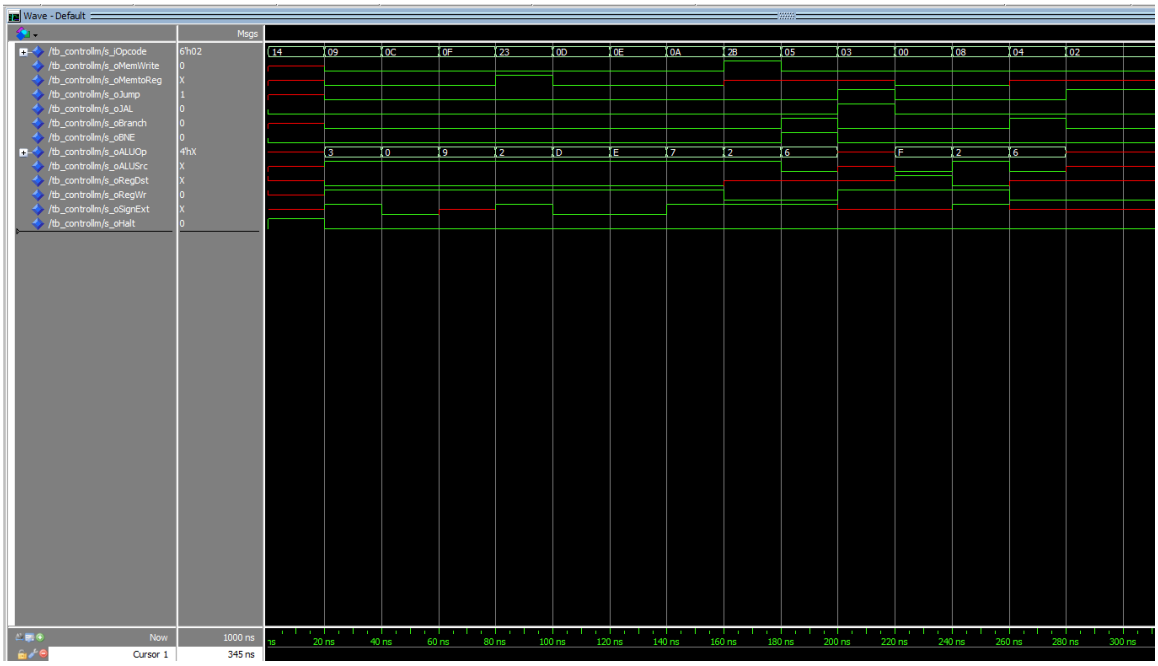
[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the

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.

This links to the spreadsheet with the control signals.

https://drive.google.com/file/d/1TskGaTIUN3AKYWd8LJw4zkew_aqAQDng/view?usp=sharing

[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).



Each instruction outputs the correct set of control signals as defined by the spreadsheet.

[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 5 control flow possibilities that our instruction fetch logic must support: Jump, jump register, branch, branch not equals, and the default case (PC+4). Jump and link is the same as a jump from the fetch perspective. The control unit and the register file handle storing the current PC+4.

A jump occurs when the jump signal is high and results in the input “jump immediate” being output as the PC on the next cycle.

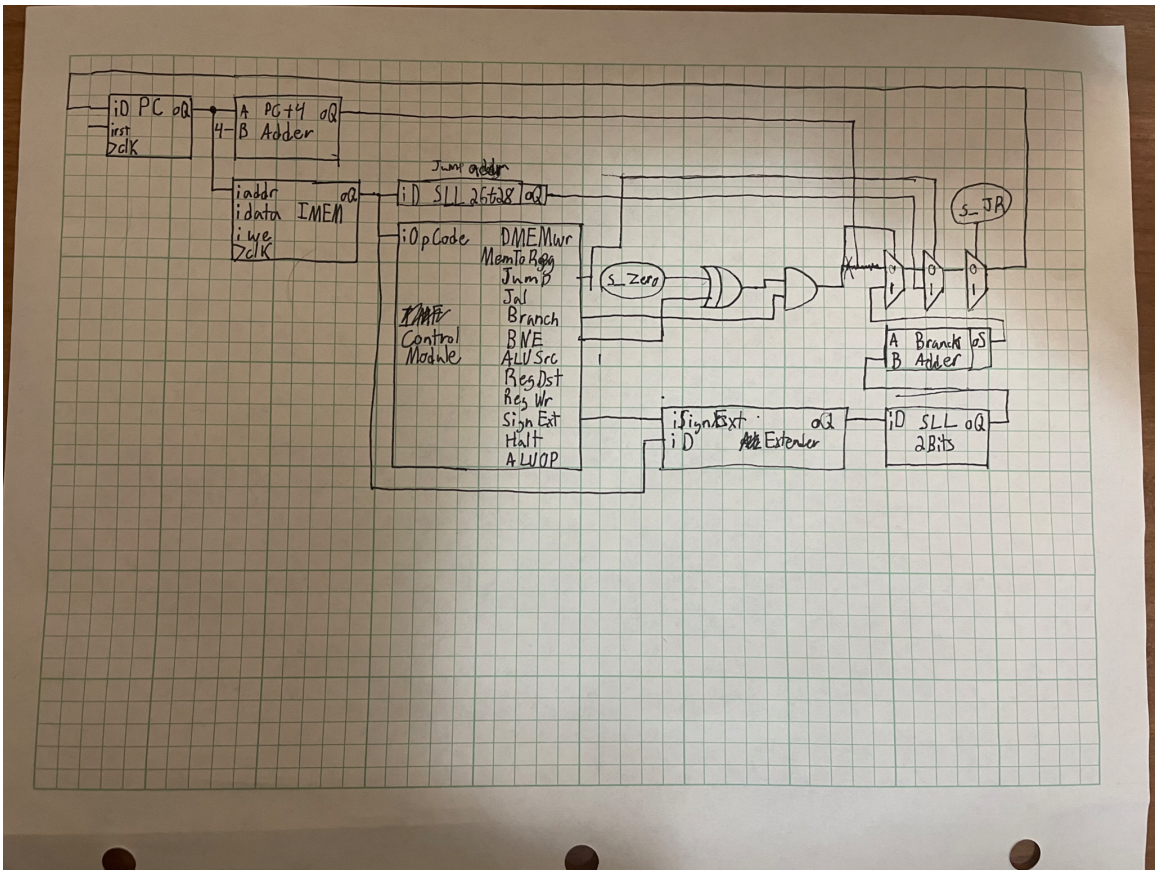
A branch occurs when the branch signal is high. For a BEQ instruction, the BNE signal needs to be set low. When this occurs and the zero signal from the alu is high and the previous PC is added to the branchAddr and output by the fetch system.

For the BNE instruction, the BNE signal needs to be set high along with the branch signal. When this occurs and the zero signal from the alu is low; the previous PC is added to the branchAddr and output by the fetch system.

For jump register, if the jump register signal is high then the output of the Fetch signal is the value of the output of the register file.

In the case where none of the previously mentioned control signals are set high, then the output of the fetch system is the output at the previous clock cycle plus 4.

[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?



There are two additional control signals that are necessary for the operation of the processor:

JAL: This causes reg write to use the current value of PC + 4. This is not seen from the perspective of the fetch system however is necessary for its functionality.

BNE: xored with the zero signal so on BNE it only branches when the two values are not equal.

Other signals that are derived from various sources are needed for the fetch circuit to operate including:

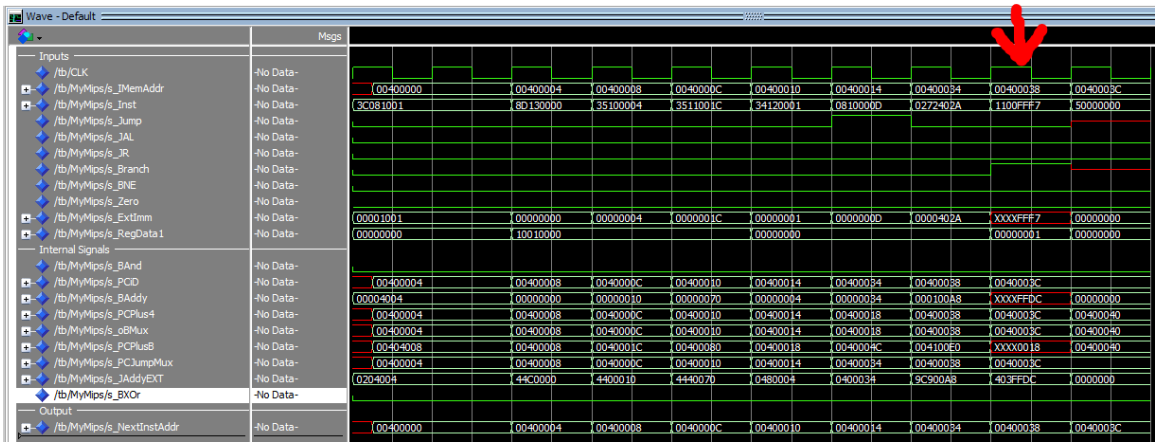
Zero: 1 if the output of ALU. Used for BNE and Branch

JR: connects to a mux that will drive the PC register to the value of the register output.

[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.

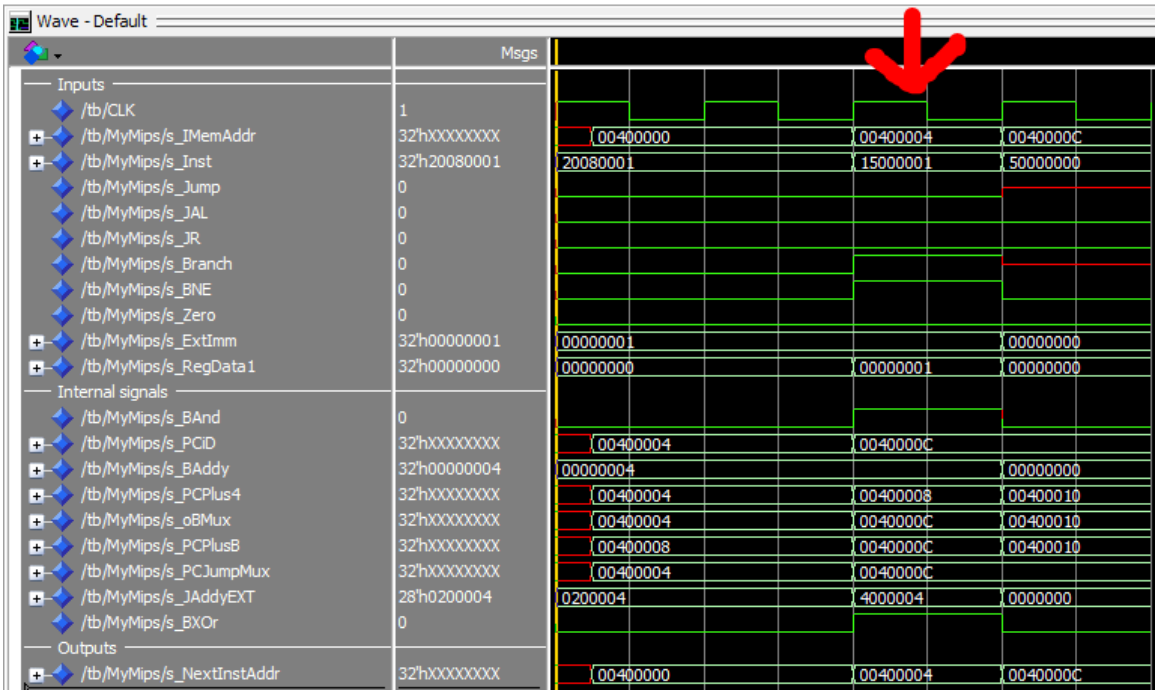
This report was completed after the compilation of the processor. In preparation for project part 2, the Fetch unit was broken up. For this reason there is no fetch unit to test and the results of the previous fetch unit test bench are lost and would no longer be reflective of the actual fetch system in the processor. For these reasons, to demonstrate the functionality of the fetch system will be demonstrated by running the testing framework with mips programs j_0.s, jr_1.s, beq_1.s, and bne_0.s.

beq test - the accuracy of the waveform is indicated by the passing of the test bench.



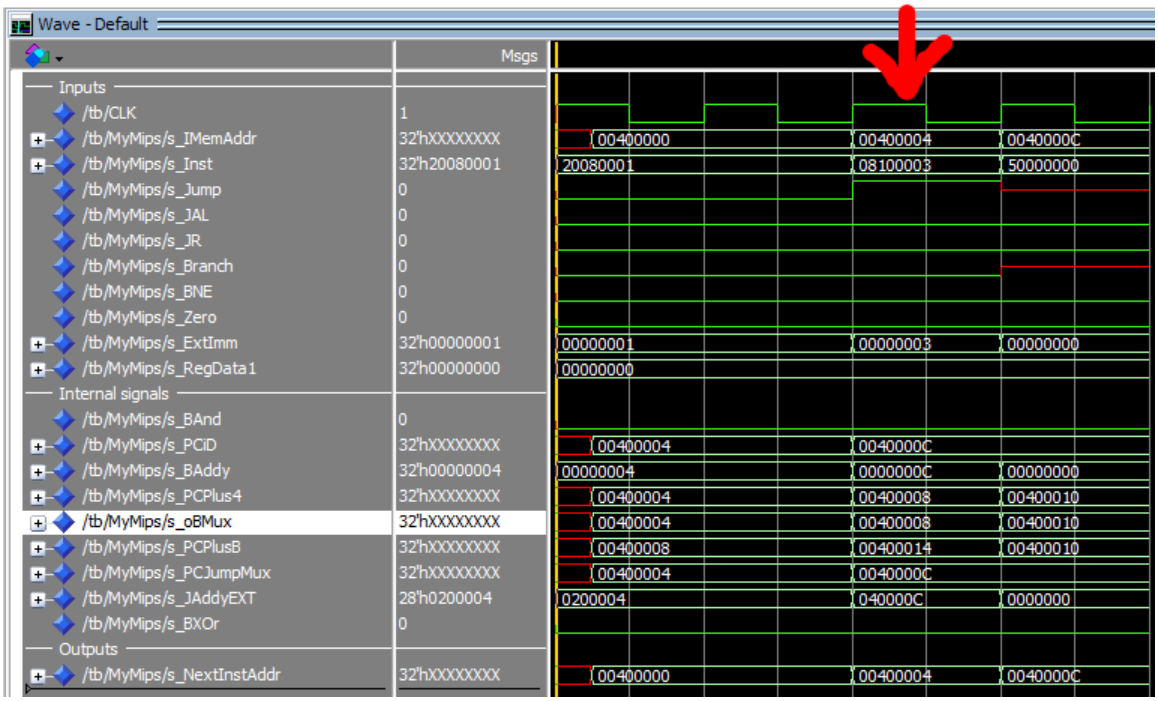
The ModelSim Waveform above also shows the functionality of the BEQ instruction. In the test framework above, the cycle marked by the red arrow indicates a BEQ instruction. The inputs of interest are the branch signal and the zero signal. Branch signal is 1 but the zero signal is zero so the processor correctly does not branch. Instead the output is just PC + 4 on the next cycle.

bne test - the accuracy of the waveform is indicated by the passing of the test bench.



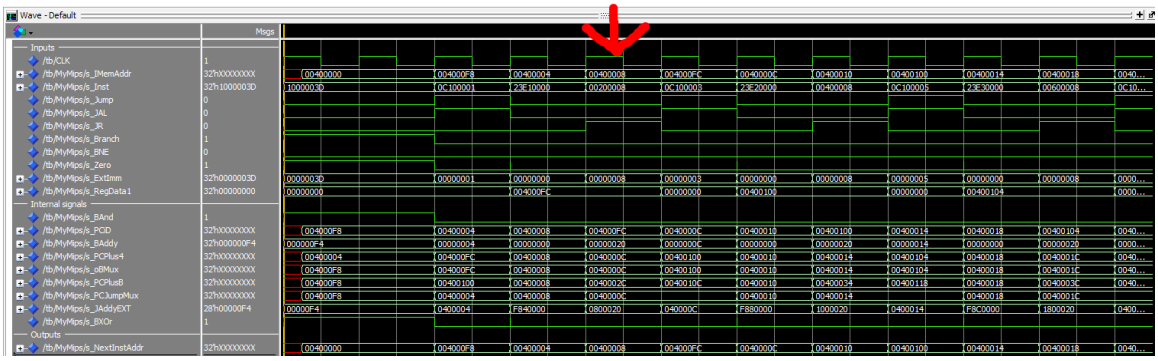
The functionality of the processor above is also shown by the ModelSim waveform itself. the cycle indicated by the red arrow a BNE instruction occurs. The branch signal signal and the BNE signal were both high which indicates a BNE. The zero signal is low indicating that the processor should branch which is done by outputting the branch address plus the current address on the next cycle.

j test - the accuracy of the waveform is indicated by the passing of the test bench.



The ModelSim waveform also shows the functionality of the processor. The jump instruction is executed on the clock cycle indicated by the red arrow. The jump signal goes high and the next instruction is set to the jump immediate (the least significant 26 bit of the instruction) rather than PC + 4 on the next cycle.

jr test - the accuracy of the waveform is indicated by the passing of the test bench.



The waveform also indicates the success of the JR instruction for the fetch subsystem. On the cycle indicated by the arrow a JR instruction occurs. The output of the Fetch system on the next cycle is the value of RegData1 rather than PC +4 which is the expected behavior.

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

The srl instruction will shift a specified number of bits in the right direction and fill the leading upper bits with zeros. The sra instruction will shift in the same direction; however will maintain the sign of the number that is being shifted.
The sla instruction will shift bits to the left and carry over whatever is in the 0-bit position. MIPS doesn't have a sla instruction because there is a finite amount of instructions that can be used and sla wouldn't provide anything advantageous or useful to warrant taking up space in the ISA for that instruction.

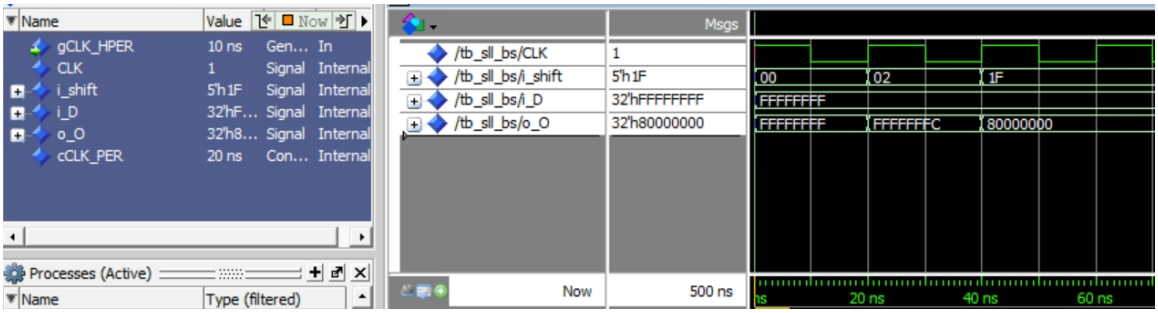
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

We have had to use two separate files to implement the arithmetic(sra) and logical(srl) instructions. We have used a multiplexor that will choose several options for shifting depending on a control which is given by a shift input. In order to drive the arithmetic operation, we used a 2-1 mux that is controlled by the most significant bit. Depending on the sign of the MSB, it will be either positive(0) or negative(1). The logical operation does not include a mux and simply operates as by taking 32 bits from a 64 bit value depending on the shift control.

[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 barrel shifter design we implemented does not support both left and right shifting. We implemented separate components to perform the different shifting operations. Our left and right shifters work by appending 32 zeros (or ones in the case of the arithmetic shift) to the corresponding left or right side of the signal. Each output is connected to a 32 to 1 mux that takes in 32 bit of the extended input signal. The select signals of these muxes is the shift amount.

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



There are three files for each different shift instruction. The waveform above is simulating the component that performs the shift left logical operation included in the proj_sll.vhd file. The waveform shows the signals i_shift, i_D and o_O, which are shared across all three files for the shifting operations. Each will have two inputs and an output. One 5-bit input which will control the shifting amount and a 32-bit input which will be used for shifting. The shifting amount can vary from no shift - “0000,” or shifting all 32 bits out - “1111.” There is a 32-bit output which will be the resulting output after the shifting operation is completed. As demonstrated in the waveform, an input of 0xFFFFFFFF is used to test shifting. Different test cases test it with 0 (no shift), 2 (shift 2 bits left), and 31 (shift 31). With the output, we can see that the first test case maintains the same input in the output. The second test case, the last 4 bits are ‘1100.’ The third test case hase ‘1000’ for the upper 4 bits on the output. Given the waveform results, we can conclude that out shifter designs are working properly.

Other shifting operations share the same setup in the previous instruction, a 5-bit shift control input (i_shift), 32-bit input for any number that is being shifted (i_D) and a 32-bit output that is the number being shifted (o_O).

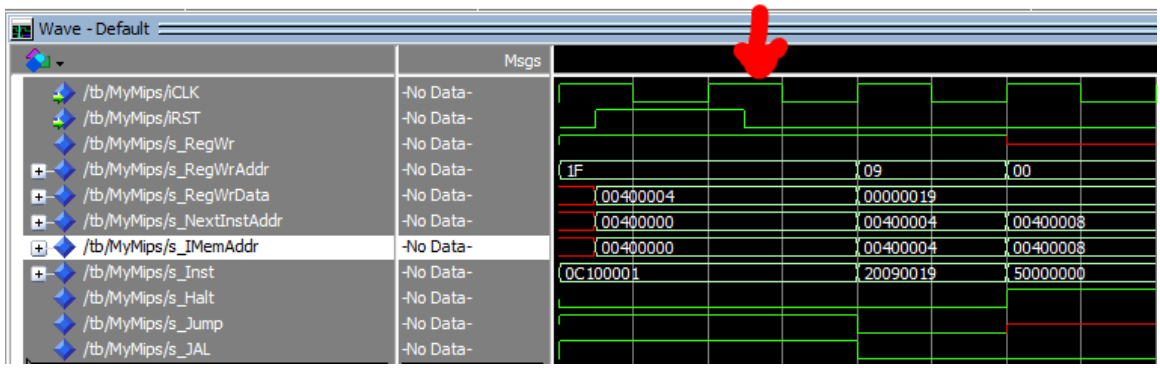
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

We needed to include some additional control signals and components to help with setting up the datapath for specific operations that we may have overlooked or did not consider at the start of the project. There are signals that were used to assist with running operations like jump register, branch not equal and jump and link. We added a JAL, JR,

BNE control signal to help control muxes that we used to drive specific operations to obtain the correct output that we will need for the program counter.

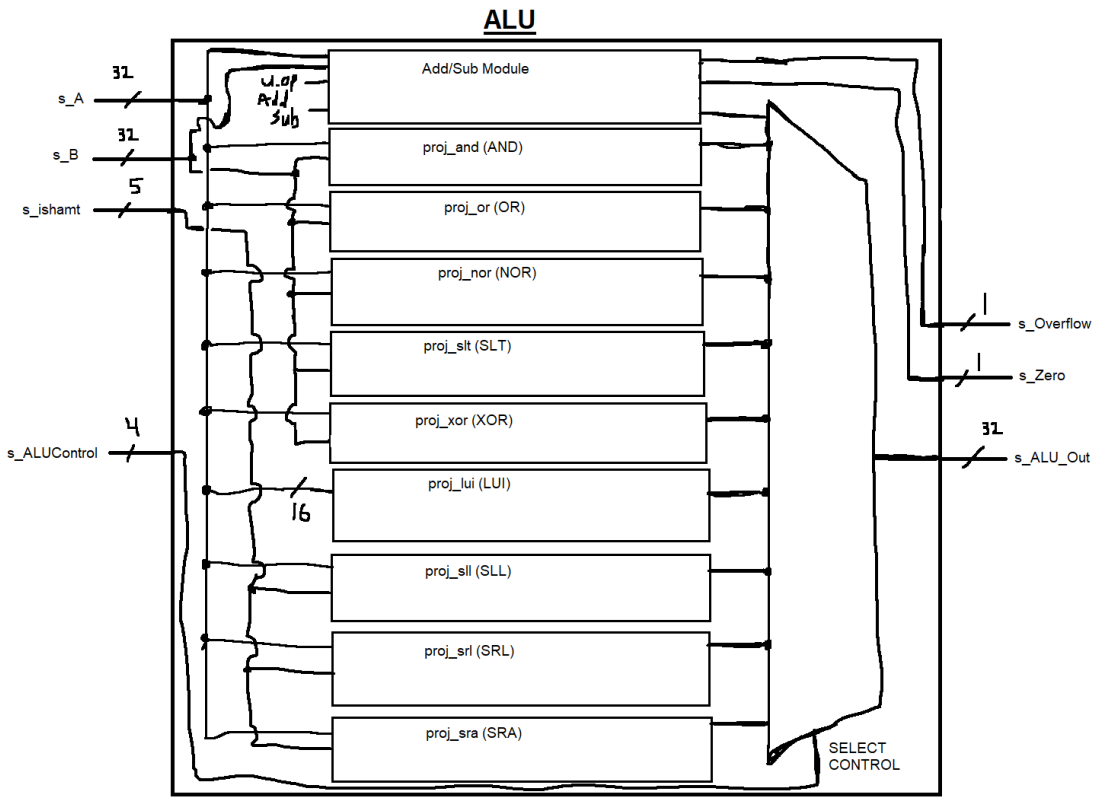
[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

The JR and BNE control signal and functionality have been explained previously with waveforms in part 2 b.iii. The JAL operation's correspondence to a Modelsim waveform can be observed in the wave form below.



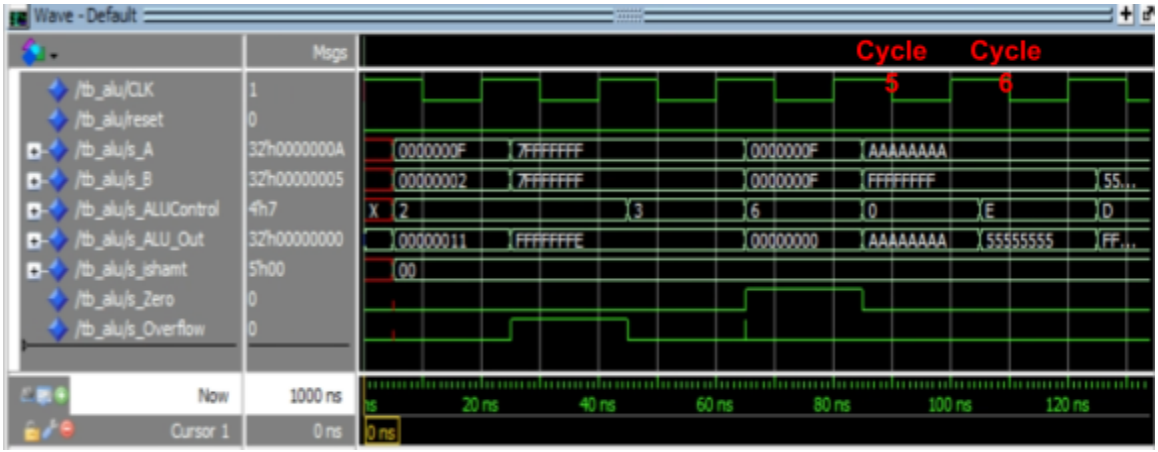
The cycle indicated by the arrow is a JAL instruction. During this cycle the value of PC + 4 is driven to the register file input, "RegWrData" and the address is driven to be 35 which is correct. Also, the processor performs a normal jump which in the case is specified to be to the next instruction by the jump immediate.

[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 can be checked when the last carry out bit and the 2nd to last carry bit are compared. When the both carry bits are different, then there is overflow. Overflow in this ALU is calculated by using an XOR gate to compare the last two carry bits. Zero is calculated by checking if the ALU output is zero. The Zero signal will then output one, otherwise will remain zero. The slt instruction is implemented by comparing the two inputs to the instruction. If the first input is less than the second input then the ALUOut output signal will be one, otherwise it will be zero.

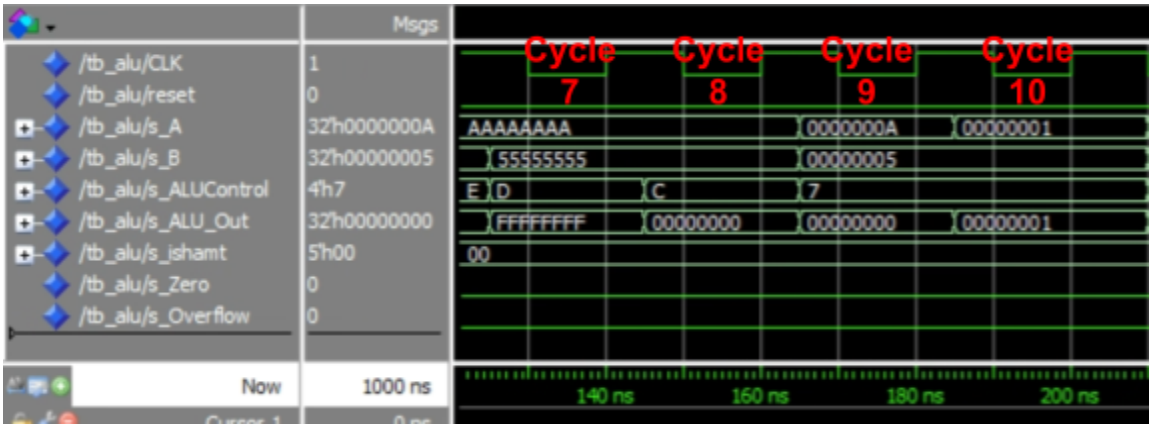
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



With the implementation of our ALU component, we will be able to execute R-type and I-type instructions. The ALUControl signal will determine which instructions are executed and which inputs and outputs will be important. In our waveform we are attempting to test behaviors that we will expect from our ALU component. We will be testing how outputs like zero and overflow behave to make sure branch and unsigned instructions, that are dependent on these signals, will be able to work. There are 4 inputs(s_A, s_B, s_ALUControl, s_ishamt) and 3 outputs (s_ALU_Out, s_Overflow, s_Zero). The s_A and the s_B inputs are both 32-bits and are the two values that are used to compute the ALU result depending on the operation. The s_ALUControl input is a 4-bit control signal that helps control which ALU operation should be executed. The s_ishamt input is a 5-bit signal that is used to indicate the shifting amount for shifting operations. As for the outputs, we have s_ALU_Out which is a 32-bit number. The signal is the result of the s_A and s_B inputs after completing an ALU operation. The Zero output is a 1-bit signal that is calculated to be ‘1’ when the ALU result is zero and ‘0’ anytime else. The Overflow output is a 1-bit signal that is used to check whenever overflow is detected. The Overflow and Zero outputs are calculated whenever the add, addu, sub and subu operation are performed. These operations specifically utilize these signals to help drive instructions, such as branching.

Based on the information of our waveform from our ALU testbench. There are several test cases that will test the different instructions and whether the output signals will work appropriately. The first case checks a simple adding operation with 15 and 2. The second and third will check whether overflow is occurring for the add and addu instructions. The next one will check for zero and the one after that will check the AND operation. After reviewing the many test cases, we were able to conclude that the 32-bit ALU component is working as we have expected.

Each of the following cycles test one of the internal instructions. Through the following I will justify their correctness.



Cycle 5:
0 xAAAAAAAA AND 0xFFFFFFFF
Expected = 0 xAAAAAAAA
ALUOUT = 0xAAAAAAAA

Cycle 6:
0 xAAAAAAAA XOR 0xFFFFFFFF
Expected = 0 x55555555
ALUOUT = 0x55555555

Cycle 7:
0 xAAAAAAAA OR 0x55555555
Expected = 0 xFFFFFFFF
ALUOUT = 0xFFFFFFFF

Cycle 8:
0 xAAAAAAAA NOR 0x55555555
Expected = 0 x00000000
ALUOUT = 0x00000000

Cycle 9:
0x0000000A SLT 0x00000005
Expected = 0x0
AUOUT = 0x0

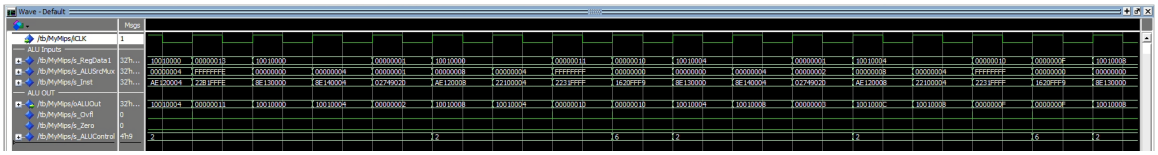
Cycle 10:
0x00000001 SLT 0x00000005
Expected = 0x1
AUOUT = 0x1

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

Due to the ALU being the final piece of our processor to be completed, it was tested with the processor as a whole. This was done by testing each of the individual provided test

bench programs. After the processor passes each of the instruction specific test benches a final test is performed using fib.s. These tests were sufficient because they ensured that the ALU is functional with many different inputs including edge cases.

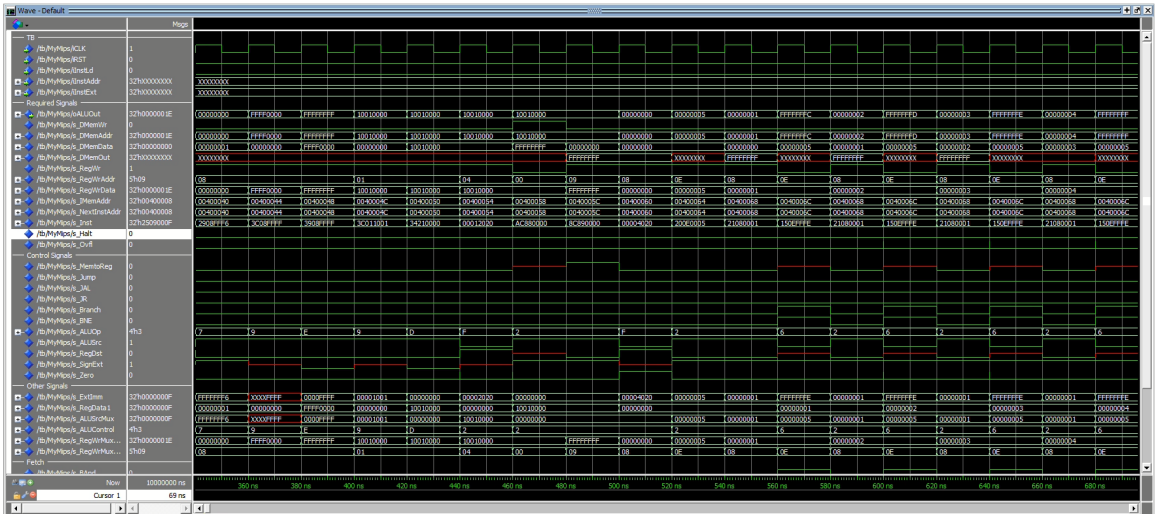
The following figure is a waveform of the Fibonacci.s test which uses many of the operations. Passing this test as well as all of the other test benches ensures the correctness of the ALU.



[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.

[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.

The test was created and ran successfully on MARS and in the processor. Both of which had identical outputs. The wave form is included to show examples of the processor meeting expectations.
The waveform of our base test program will start the first instruction at 0 ns. The first instruction will be add $8, $0, $0. As we can see, the s_RegWrAddr is set to 8, which will write to the 8th register in the register file as we would like. The next instruction at 40 ns is addi $8, $8, 15, we can expect $8 = 15. The signals oALUOut and s_RegWrData show to be 0x0000000F which means that our first two instructions were able to execute correctly. Our next instruction will be addiu $9, $8, 15 at 60 ns. As we can, signals appear to look correct, s_RegWrAddr = 9, s_RegWrData = 0x0000001E = 30. This is exactly what we were expecting - we wanted to add 15 from $8 with immediate 15 to get a final value of 30 in $9. Another instruction is nor $13, $12, $zero at 220 ns. The waveform shows that the signals s_RegWrAddr = 13, s_RegWrData = 0xFFFF0000. We plan to NOR the data in $12 => 0x0000FFFF and $0. oALUOut and s_RegWrData = 0xFFFF0000, which shows the NOR instruction was correctly applied to the inputs and saved to the $13.

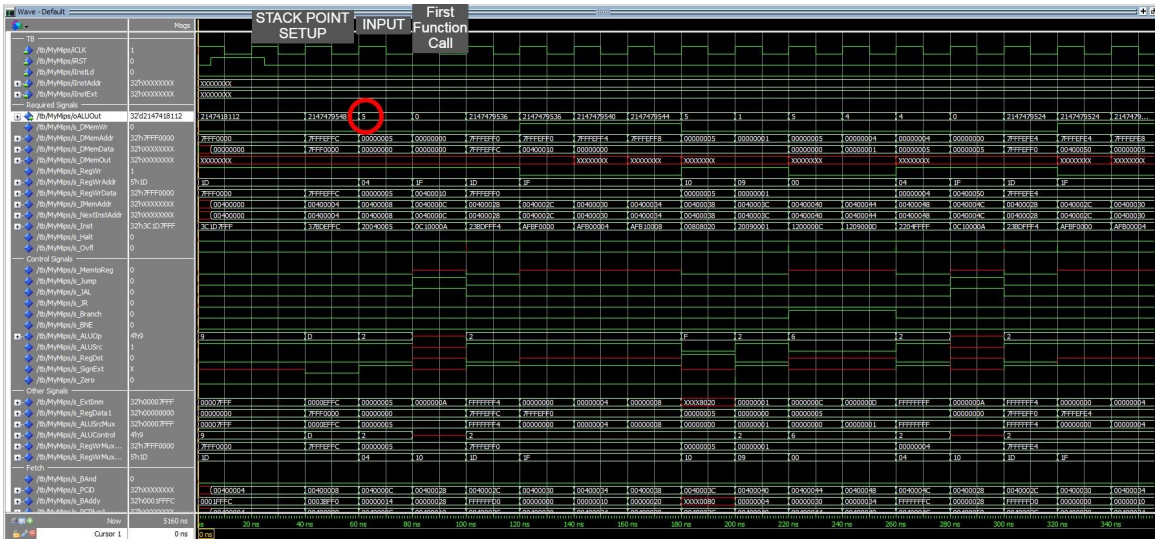


[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

This program was shown to be successful in mars and on the processor through the test framework. This is clearly visualized by the model sim waveform.

To begin with the program sets up the stack point to the value the MIPS has its stack pointer set to by default. This is done by simply loading the top of the data first with LUI and then the bottom with ori. The next line shows the input to the function which is circled in red. It is five to show that the fifth fibonacci number will be calculated which will require 1 initial function call followed by 4 recursive calls. The next instruction is the first function call as indicated by the jal signal being set high.

The second waveform shows the success of the program. The output is written to $4 such that a syscall would print it. The output circled in red is 5 and 5 is indeed the 5th fibonacci number.

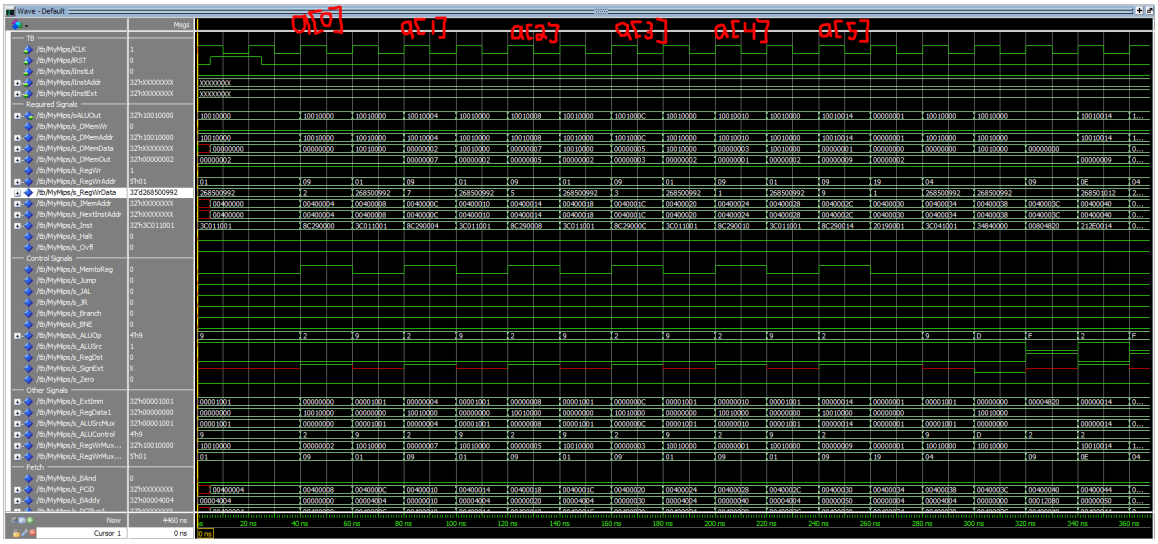


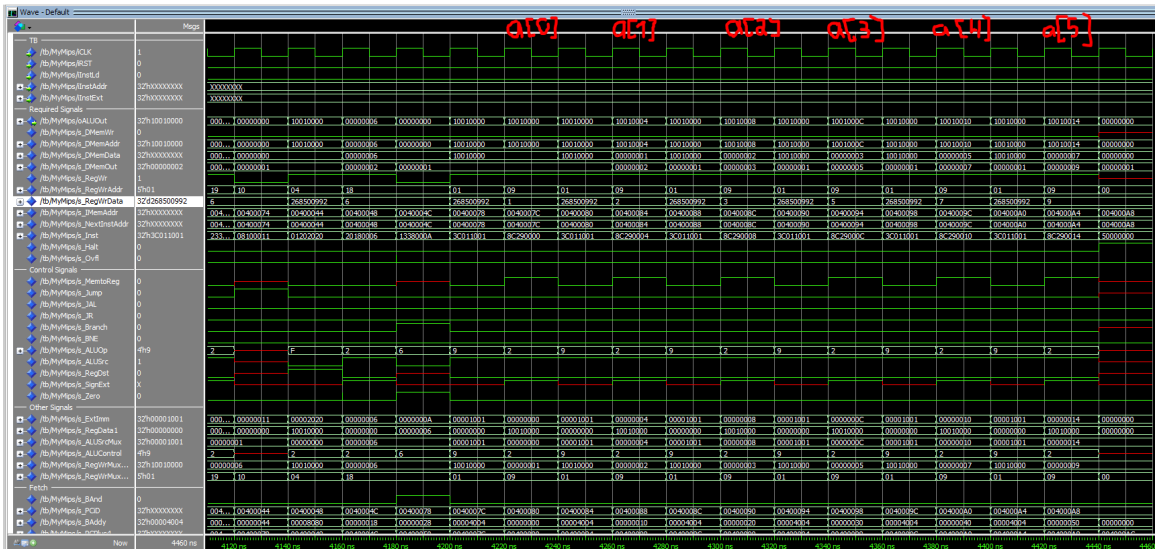
[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

The bubble sort program has been shown to be successful due to the output on mars being correct, running on the test framework on our processor successfully, and by the waveform that the test framework produced. To begin the program each of the array elements are loaded into $t1 just to show their order in the waveform. This can be viewed

by looking at the s_RegWrData under the array indices labeled at the top of the waveform in red. It is clear that in the beginning of the program the data is in the wrong order. It is clear that the data being displayed is contiguous in memory because each DMemAddr is 4 more than the one before it and the data is 4 bytes in size per entry.

The second wave form reveals the success of the program ran on our processor. The same data memory addresses are accessed again and reveal that they are now in the correct order. The accesses are again labeled above the wave form to show the order of the array elements in memory and the value stored there is driven to s_RegWrData.

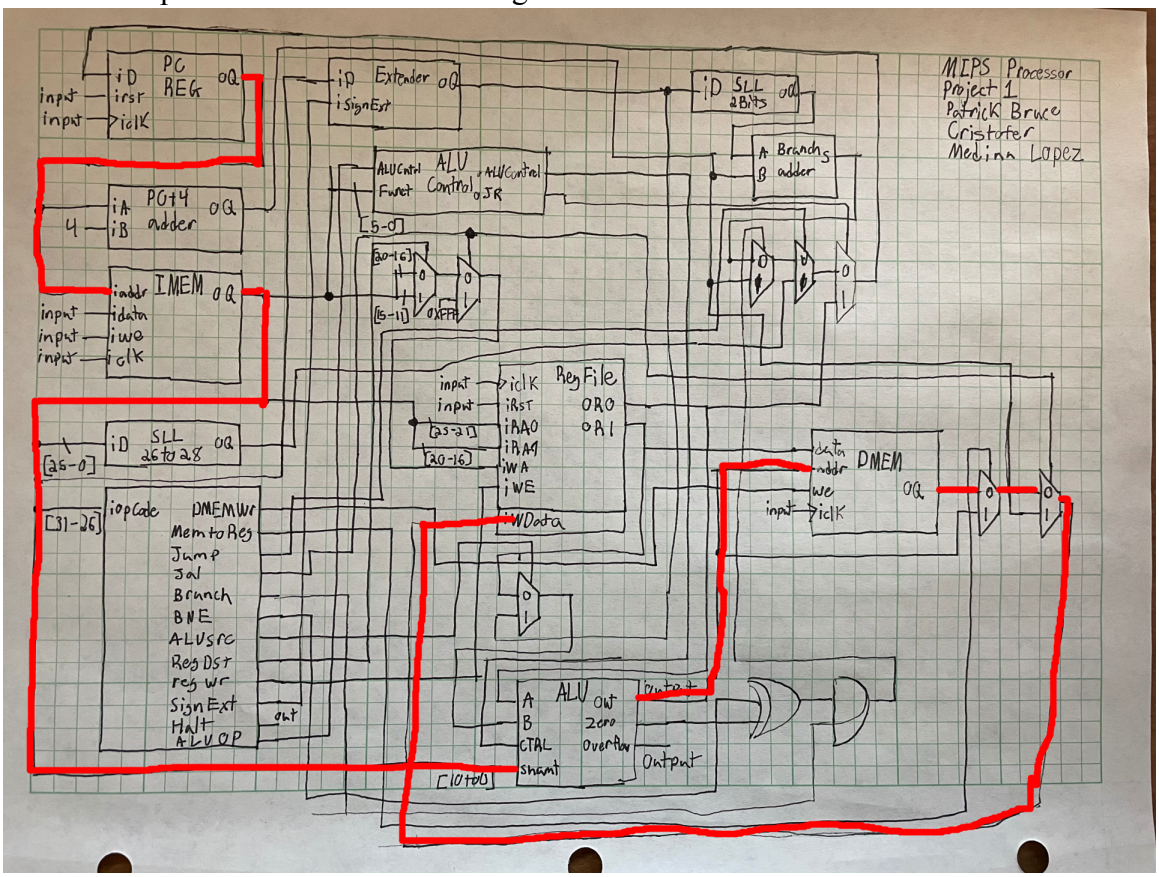




[Part 4] report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematics. What components would you focus on to improve the frequency?

The maximum frequency according to the timing.txt file is 27.72 MHz.

The Critical path looks like the following:



Critical path: PC REG -> IMEM -> ALU -> DMEM -> RegSrcMux -> Jal mux -> regfile

Improving the performance of any component on the critical path will improve the speed of the max frequency. In particular the best component to improve would be the mem component because it is on the path twice and the single biggest contributor to the time.