# **CprE 381: Computer Organization and Assembly-Level Programming**

# **Project Part 1 Report**

Team Members: \_\_\_\_\_\_\_\_\_\_\_\_\_\_Sam Burns\_\_\_\_\_\_\_\_\_\_\_\_\_

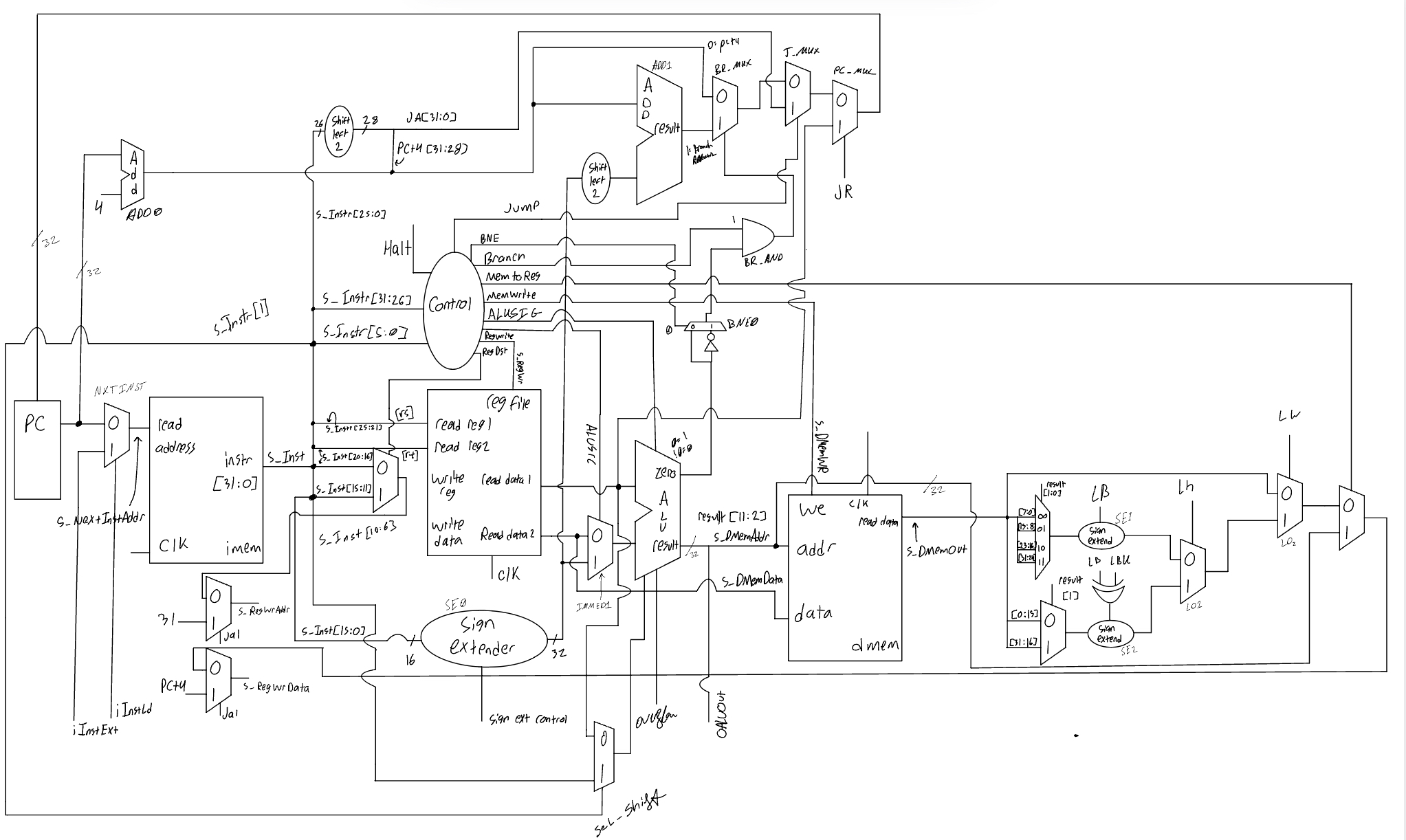
\_\_\_\_\_\_\_\_\_\_\_\_\_\_Isaiah Aldiano\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Project Teams Group #:\_\_\_\_\_\_\_\_\_\_A\_01\_\_\_\_\_\_\_\_\_\_\_

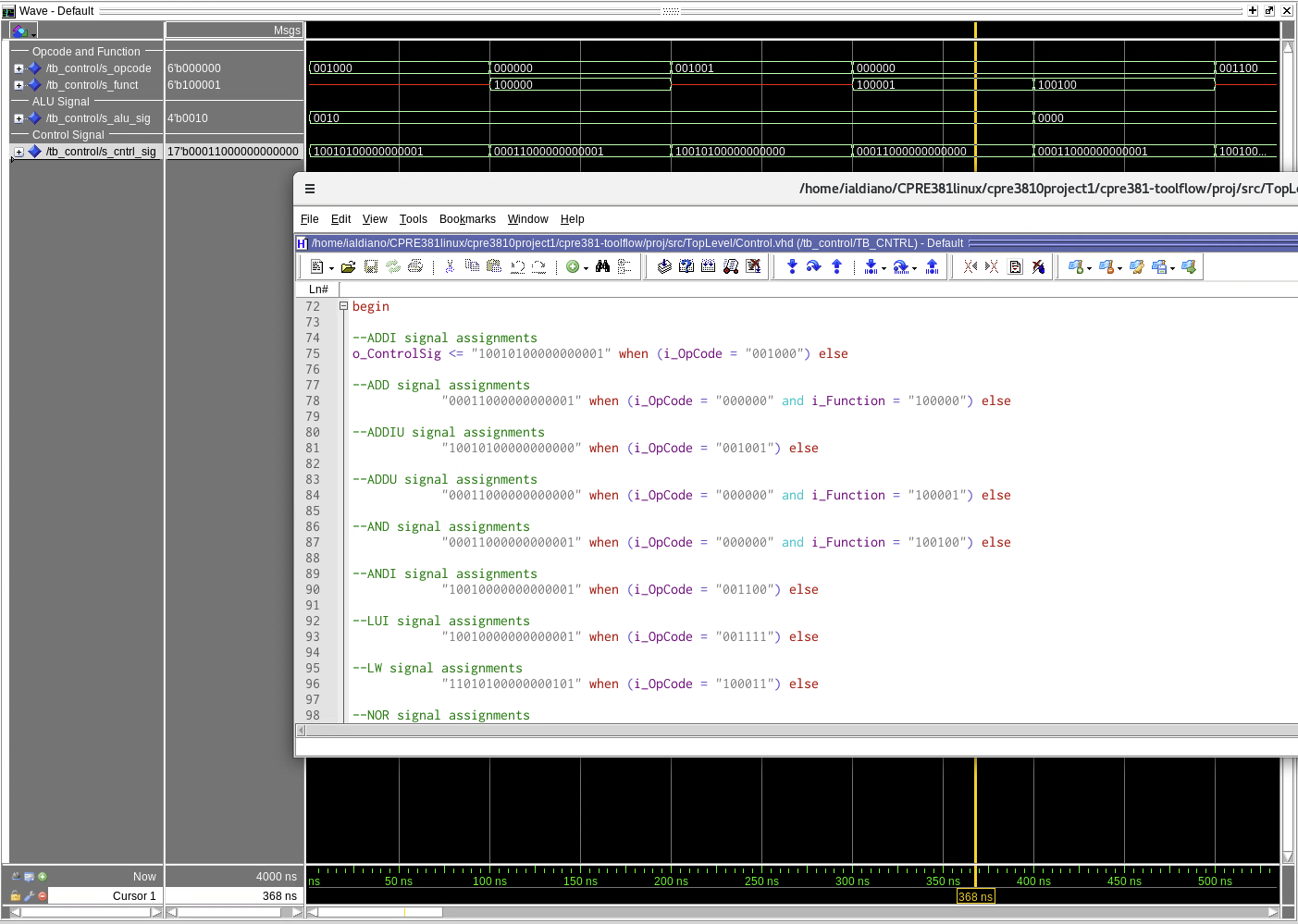
***Refer to the highlighted language in the project 1 instruction for the context of the following questions****.*

[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.

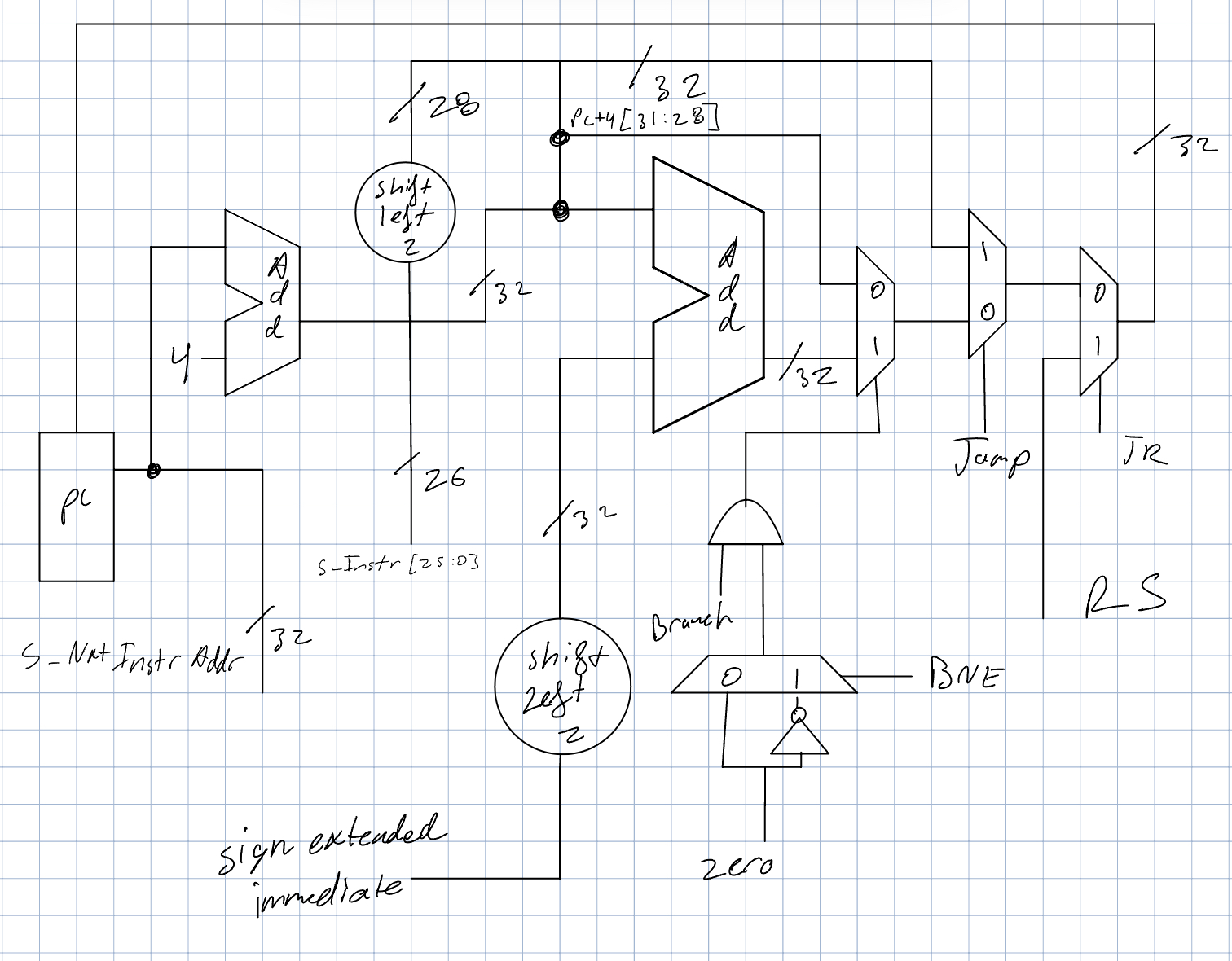
[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually and show that your output matches the expected control signals from problem 1(a).



[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.

Out of the 34 instructions our processor implements there are 5 fetch related instructions and 6 control flow possibilities: regular increment, jump, jump and link, jump register, branch on equals, branch not on equals.

* Regular Increment
  + Branch = 0
  + Jump = 0
  + JR = 0
  + JAL = 0
  + RegWrite, MemWrite = 0
  + Other controls = x
* Jump:
  + Jump = 1 (([31:28] of PC+4) + (inst J address -> sl 2))
  + JR = 0
  + JAL = 0
  + Branch = x
  + RegWrite, MemWrite = 0
  + Other controls = x
* Jump and Link
  + Jump = 1
  + JR = 0
  + JAL = 1
  + Branch = x
  + RegWrite = 1 (Storing PC + 4 into $ra)
  + MemWrite = 0
* Jump Register
  + JR = 1
  + Jump, Branch = x
  + Read Reg 1 value -> [25:21]
  + RegWrite, MemWrite = 0
  + Other controls = x
* Branch on Equals (Compares 2 values from RegFile)
  + Branch = 1
  + BNE = 0
  + Jump = 0
  + JR = 0
  + ALUSrc = 0
  + RegWrite, MemWrite = 0
  + Other controls = x
* Branch on Not Equals (Compares 2 values from RegFile)
  + Branch = 1
  + BNE = 1
  + Jump = 0
  + JR = 0
  + ALUSrc = 0
  + RegWrite, MemWrite = 0
  + Other controls = x

[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?

Control signals needed for the fetch logical are Branch, BNE, Jump, and JR

[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.



In order to test all the functionality of the Fetch Module I first started with resetting the module, which is demonstrated one quarter of the way through the first clock cycle above. The output of the PC gets set to the same value that the PC is initialized to in MARS. Showing that it is working. Next I incremented the module to make sure that the base case of PC = PC + 4 is working correctly , which it was. Next I tested what happens when trying to branch while the zero flag is not set, which correctly results in another increment of PC = PC + 4. Next I performed a branch operation which correctly resulted in signextended\_imm<<2 + (PC+4). Next I performed the same set of operations for BNE instructions, which also worked correctly. Next I tested the Jump and Jump Register instructions which both worked correctly. This testbench reflects all control flow possibilities of the fetch logic working as expected.

[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 a 32 bit value to the right while shifting IN 0’s where as SRA a 32 bit value to right while shifting in the MSB (31st bit) to keep the value signed. SLA would be a redundant instruction to add to MIPS because shifting left shifts out the MSB negating the need for the arithmetic instruction

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

32 bit data value = i\_data

shift amount = i\_shamt

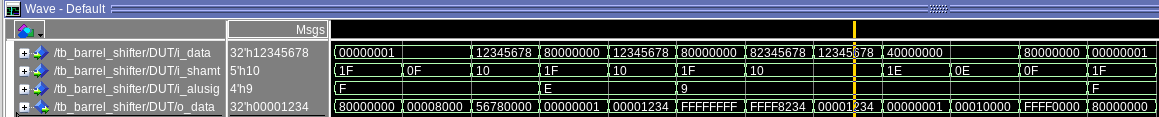
Logical shifts are implemented using the unsigned shift right/left function that takes in input some N length (i\_data) vector and the amount to shift by (i\_shamt). Similarly, arithmetic shift operations use the shift function but instead use the signed shift function. An arithmetic shift left is redundant so only SRA/SRAV instructions utilize a signed shift. The shift operation performed is determined by the ALU signal (i\_alusig). Each bit of the ALU signal is checked and then outputs the corresponding operation to the output signal.

Our barrel shift supports both left and right variable operations by utilizing an external mux of i\_shamt before it is sent to the barrel shifter. This reduces the size of the barrel shift by allowing for immediate and variable shifts to be performed using the same ALU signal.

[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 output of the barrel shifter is determined by the ALU signal. By checking the bits of the ALU signal and adding a case for that specific signal the barrel shifter can perform proper SLL/SLLV operations.

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



i\_data = input value to be shifted, i\_shamt = shift amount, i\_alusig = determines shift type, o\_data = operated shift value

i\_data = 0x0000\_0001

i\_shamt = 0x1F (31)

i\_alusig = 0xF (SLL/SLLV)

o\_data = 0x8000\_0000

A shift left operation is chosen with the ALU signal, shamt is set to 31 bits, and the o\_data matches the expected result.

i\_data = 0x82345678

i\_shamt = 0x10 (16)

i\_alusig = 0x9 (SRA/SRAV)

o\_data = 0xFFFF\_8234

A shift right arithmetic operation chosen by ALU signal, shamt set to 16 bits, o\_data matches the signed 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.

We madre several major design decisions, including the following:

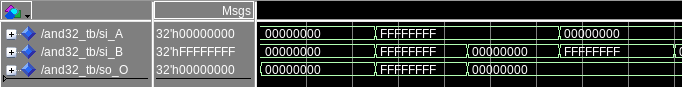
* Dmem Loading
* LUI
* Zero
* ALU MUXing
* 32 bit gates for bitwise operations

The design approach for these modules was to implement them in the most simple way, that way they would be easily integrated and tested. However, the N-bit digital logic type operations (AND,OR,XOR,NOR) were each tested with their own testbench found below. The testbench for LUI is also found below.

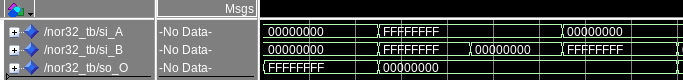
We reserved the DMEM loading to be tested using the testing framework, however the others were tested in the following ways: zero functionality, as well as ALU muxing, were left to testing using the ALU testbench (results and elaboration found below)

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

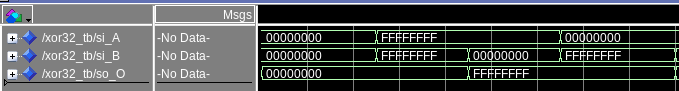
Each of the basic, 32-bit, 2-input, bitwise operations are fully tested below and are labeled. Each of the modules performs as expected.



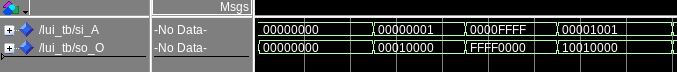
AND



NOR

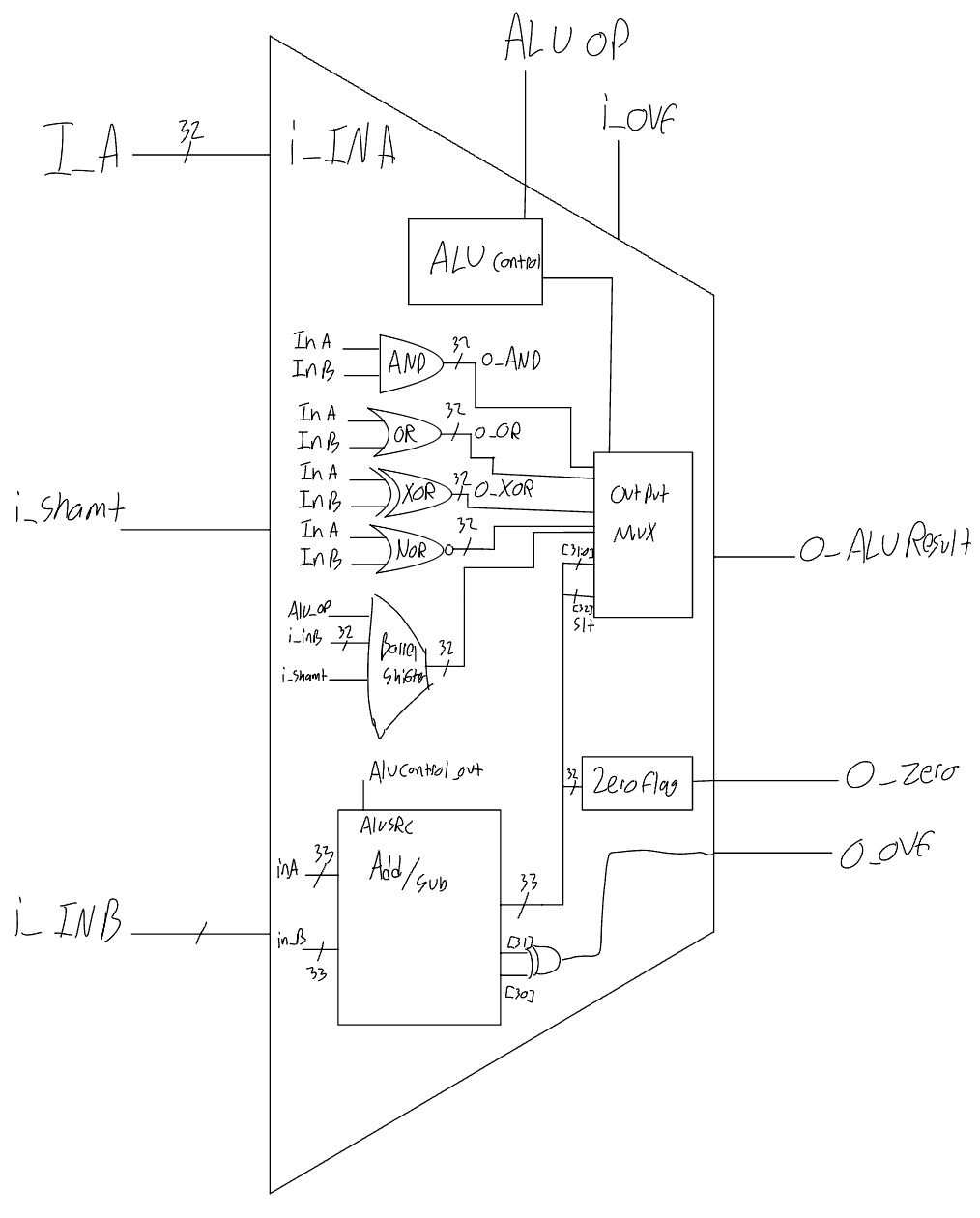


XOR



LUI

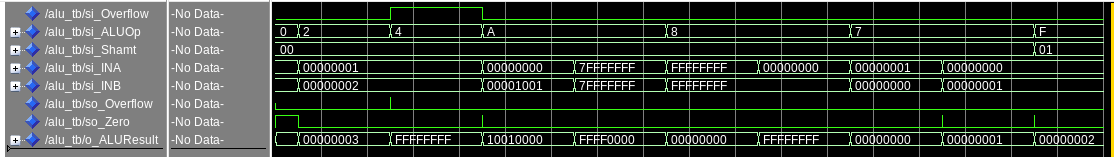
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is slt implemented?



We calculate zero by using a separate module that sets the output when the input is equal to zero, and the output is deasserted for all other cases. This module is hooked up to the lower 32 bits of our adder/subtractor module. Overflow is calculated by xor-ing bit 31 and bit 30 of the output of the adder/subtractor module. SLT is calculated by using the subtraction functionality, and extending the adder/subtractor module to 33 bits, both inputs are sign extended at the 32nd bit for to become 33 bits in width, then we take the 33rd bit to be the bit zero of our 32 bit SLT signal.

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

Out first implementation of the ALU only implemented the add/subtract functionalities. The reasoning behind this was that it would give us time to finish all other components and integration of the processor to begin testing instructions as early as possible. The justification for this was to identify potential errors in our design, and not just dedicate all our time to the ALU without knowing if other components would be sources of error. Therefore our first testbench of the ALU was implemented by replacing the ALU with the adder/subtractor module from our second datapath. Once this worked, we got to testing our processor as a whole with simple add instructions, adding functionality incrementally from there. The testbench below shows the final functionality check of our ALU; once we verified that other major modules such as control and fetch worked in our integrated design, we began adding the rest of the functionality shown below.



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

I believe that our testing plan is comprehensive because I essentially test all of the functional blocks of the ALU with this test bench. For the first two cycles show an add and and subtract of 2 and 1 respectively. On the subtract, the overflow is enabled, which shows that unsigned and signed operations work in the first two cycles. The third cycle shows that LUI is working. And the next cycle shows that overflow can properly be disabled. The fifth and sixth cycles show that the XOR operation works, which confirms that all other bitwise operations will be successful, as they were already separately tested and integrated in the same way as XOR, meaning they will also work. The 7th and 8th cycles show that SLT works properly, and the final cycle shows that we have implemented our barrel shifter successfully, no further testing for this component is needed, as it was already separately tested as well.

[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.

[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.



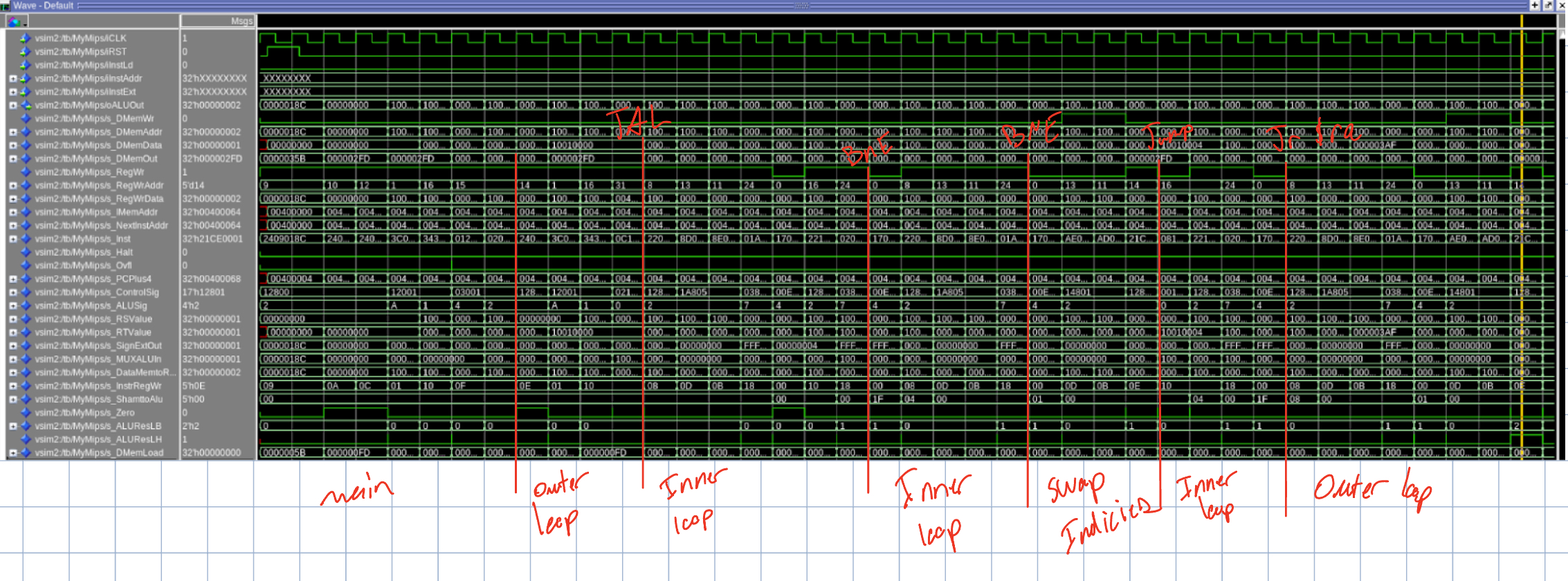
Base tests performed all shifting operations, all arithmetic operations, and all bitwise operations. The test is split up into three different sections and the waveforms align with the correct ALU signal and control signal.

[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.



CF tests performed a BNE and BEQ as well as created an activation record of 4 using JAL write to $RA the next instruction value. When the new label was reached the $SP was decremented and the next $RA value was pushed onto the stack. After the last JAL the instruction JR $RA returned to the previous jump location, the stack pointer was incremented, $RA was loaded with the previous return value, and $JR $RA walled again. In the last JAL register $t2 was changed from 1 to 0 to allow for a branch to the exit the program.

[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm ([link](http://en.wikipedia.org/wiki/Bubble_sort)). Name this file Proj1\_bubblesort.s.



This portion of bubble sort shows the program entering a double nested loop, swapping two values at adjacent indices, and escaping to the outer loop due to an indice swa occurring. After the last $ra the program repeat by entering the inner loop again which is observable with the repeating of the same s\_RegWrAddr registers as the first outer loop into inner loop

[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?

According to the timing report from the synthesis command, the maximum frequency our processor can run at is 25.56MHz. Our critical path is traced on the schematic below. The components we can focus on to improve the accuracy are the barrel shifter and the mux in the register file. The writeback logic could also be improved as well. I believe professor Duwe mentioned something about being able to split dmem up into two cycles as well, which would certainly be something to look into for project 2 as well.

The annotated diagram with the critical path of the processor can be found below.

