CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: ___Nihaal K Zaheer_____

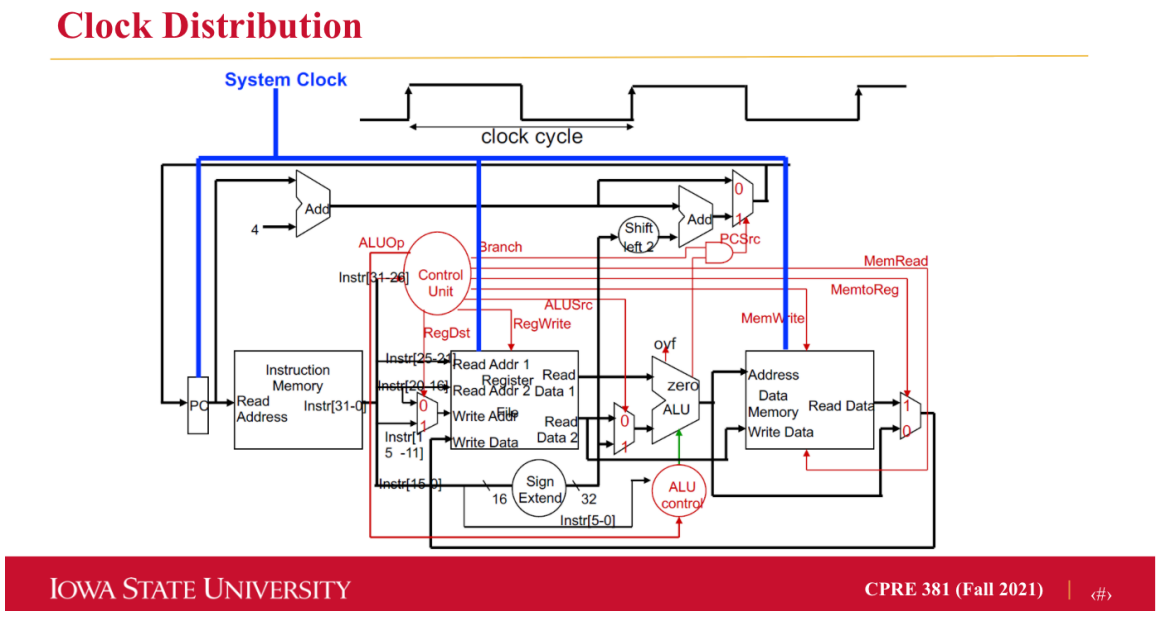
___Derek Lengemann_____

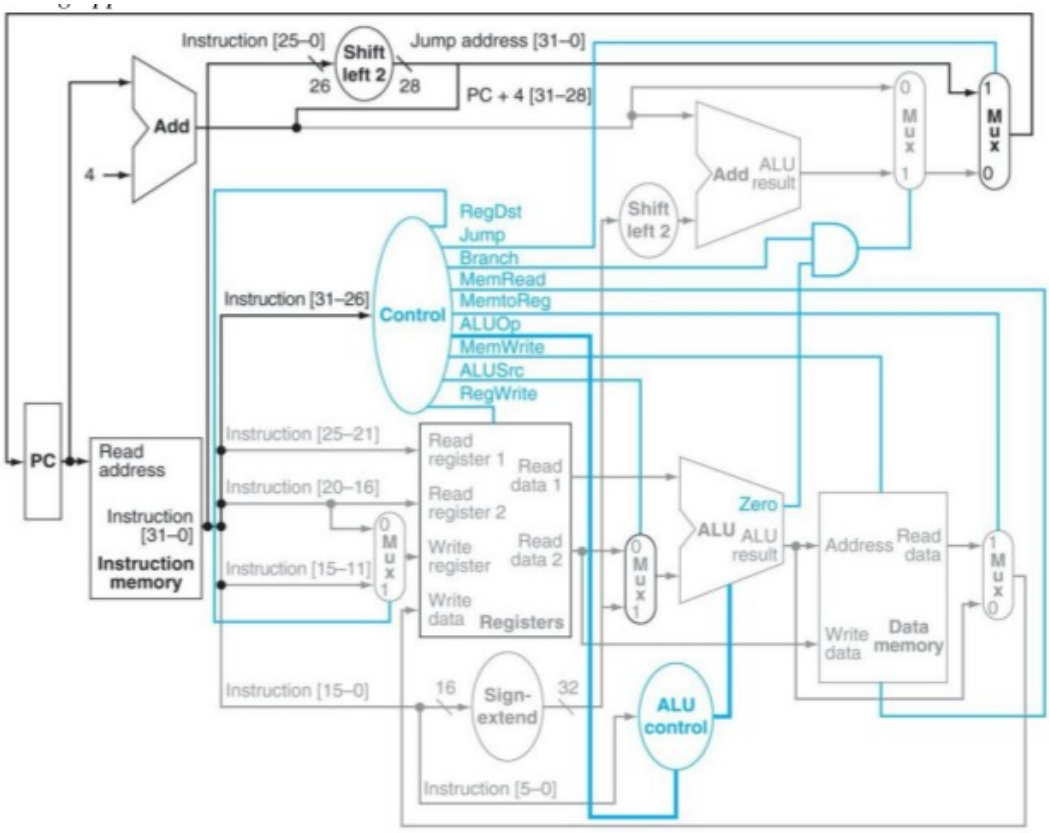
___Kareem Eljaam_____

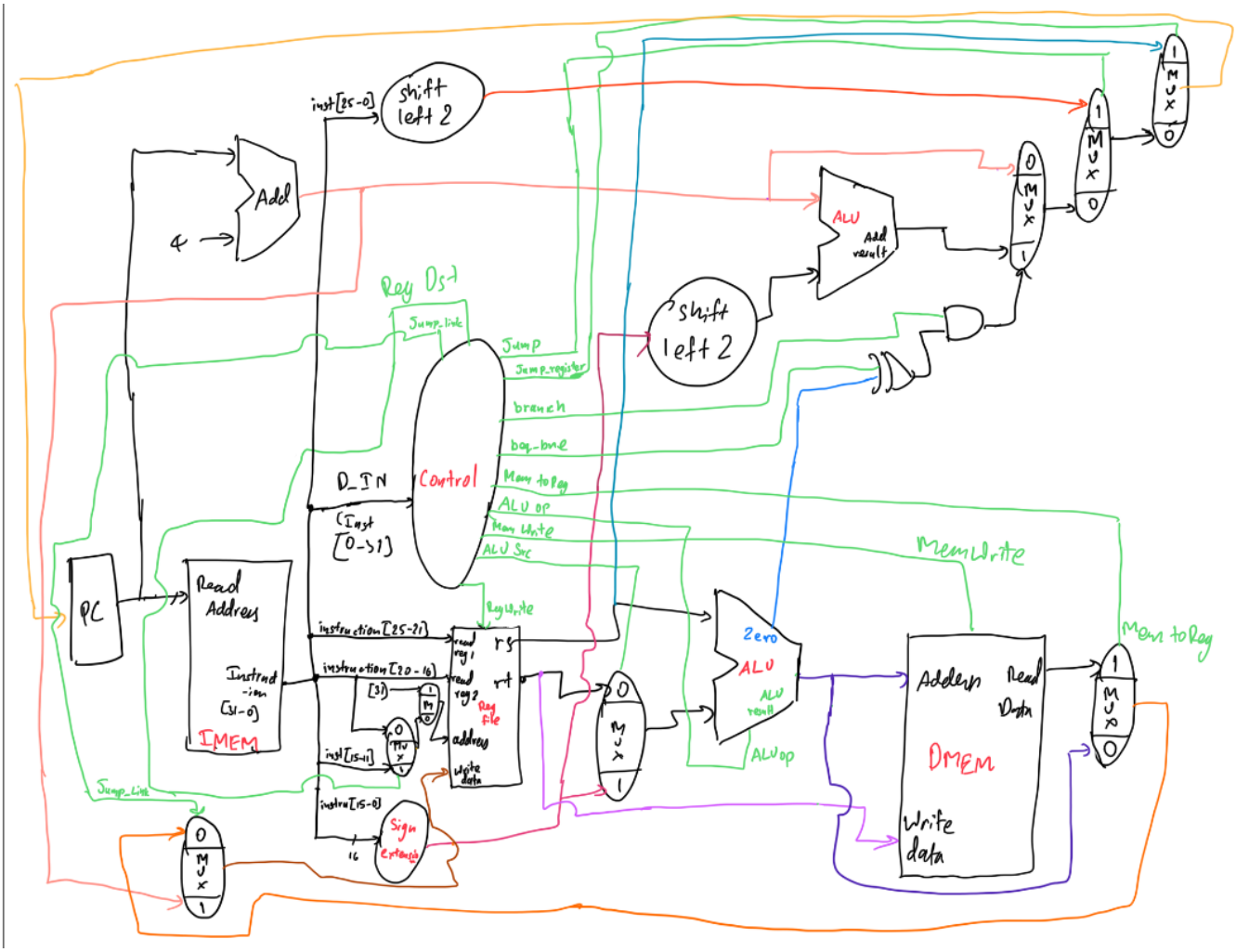
Project Teams Group #:_____4-5_____

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.




Instruction [25–0]
Shift left 2
Jump address [31–0]
26
28
PC + 4 [31–28]
Add
4
0
M
u
x
1
1
M
u
x
0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory



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.

Given in spreadhseet, Proj_1_control_signals.xls

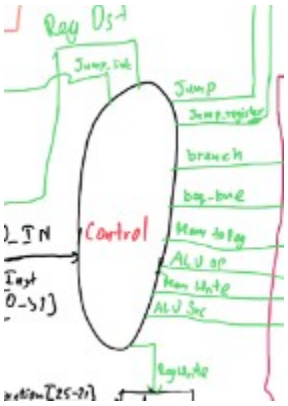
[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually, and show that your output matches the expected control signals from problem 1(a).

We made a do file that forces values to the signal we need to test out and runs the waves for a fixed amount of time. This verifies our design and shows us that the control unit works the way we want it to. We also made a test bench for the same

[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.

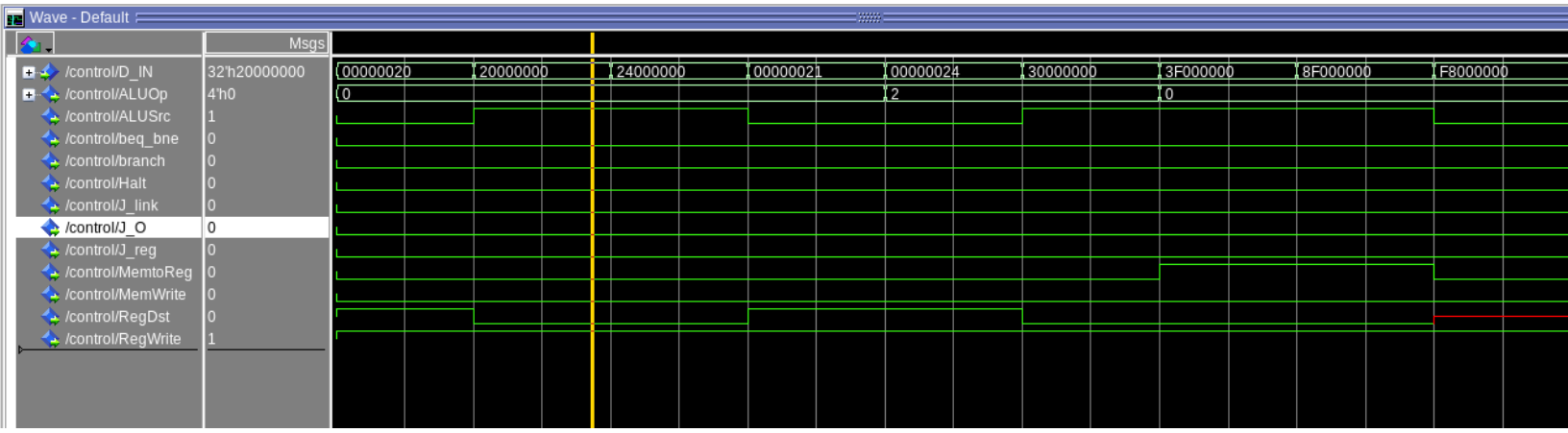
Control flow possibilities the fetch logic must support is jump, jump and link, jump register, branch etc. Basically any component that modifies the program counter.

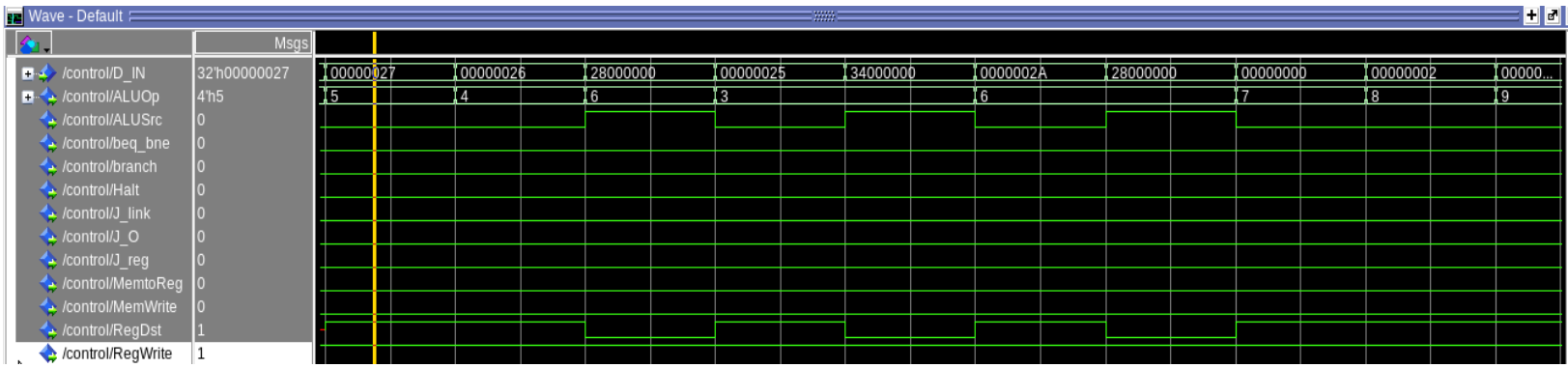
[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

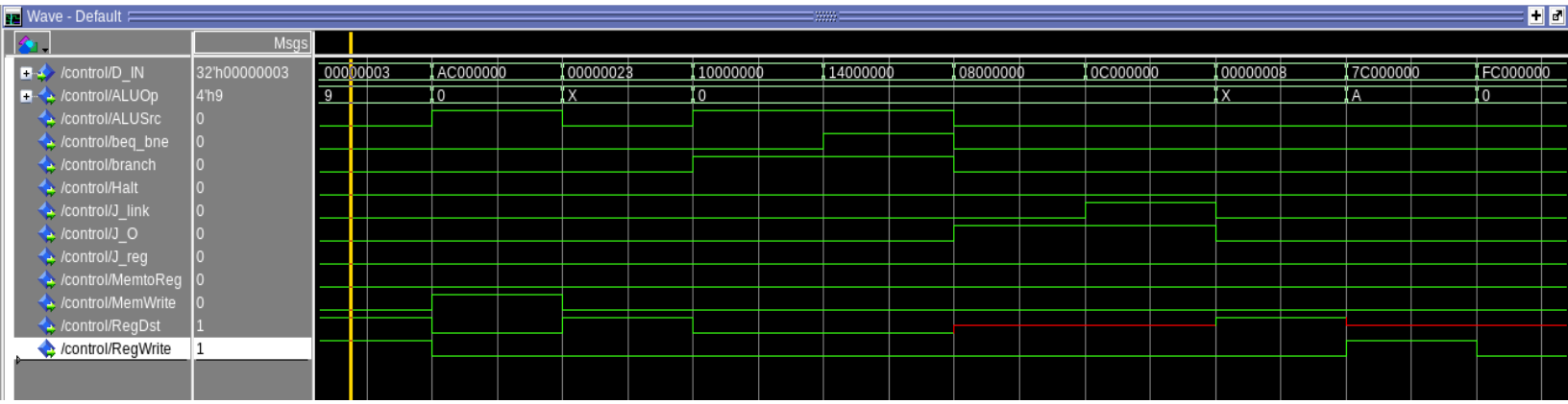


Additional control signals used are Jump, Jump_link, Jump_register, branch, beq_bne and halt.

[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.







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

Logical shifts will fill in any vacated bits after the shift with the value of ‘0’. Arithmetic shifts will fill in any vacated bits with whatever the value of the most significant bit is in the 32 bit value to be shifted. This is because logical shifts treat the number as an unsigned number whereas arithmetic treats it as a signed number. Shift left logical and shift left arithmetic have the exact same result. Thus MIPS does not need an sla instruction when sll will do the exact same thing and produce the same results.

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

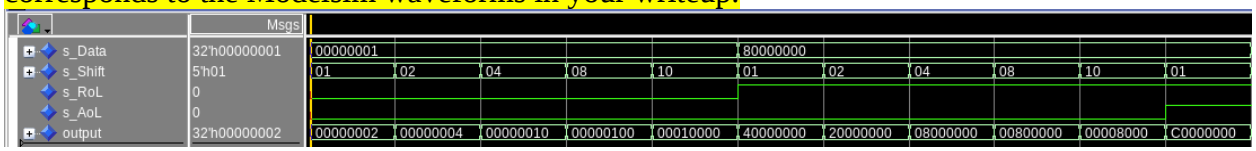
It does so by using a 2 to 1 multiplexer. Using the input i_AoL if the input is ‘0’ then the shift is logical and the output signal from the multiplexer will be ‘0’. If the i_AoL is ‘1’

then the shift is arithmetic, and the multiplexer will assign the output signal to whatever the most significant bit of the 32 bit number input is. This signal is then assigned to all the bits that have been vacated.

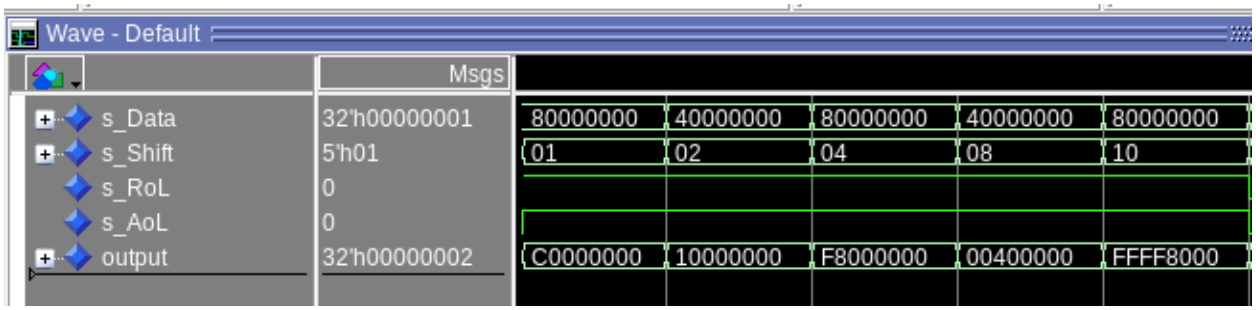
[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

The right barrel shifter can easily support the left shifting operations by flipping around the input 32 bit number to be shifted. If the shift is left, take the input 32 bit input and flip it around, meaning the least significant bit becomes the most significant bit and the most significant bit becomes the least significant bit and so forth. After that perform the shift as usual using the right barrel shifter. After the shift flip the output of the barrel shifter back around and you have the left shift that you desire. To implement this one can use multiplexers. A multiplexer to determine between whether inputting the normal 32 bit number to be shifted or the flipped around version into the barrel shifter. And then a multiplexer after the shift to determine whether to output the 32 bits directly from the barrel shifter or to output the flipped version of those 32 bits from the barrel shifter. The control inputs for these 2 to 1 muxes can be the same signal because when you want to shift left you will want to flip it around both before and after the barrel shift and when shifting right you will not want to flip it either time. This control signal can be input into the component to determine whether the user wants to go left or right.

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



The first five tests denote a test of the sll. The input is always the number 1 in 32 bits. In the first test it shifts by 1, in the second it shifts by 2, in the third by 4, and in the 4th by 8, and in the 5th by 16. These are all correct because 1 in hex shifted to the left 1 is 2. 1 in hex shifted by 2 is 4 in hex. When shifted 4 bits it becomes 10 in hex, when 8 it becomes 100 and when shifted 16 it becomes 10000. The next 5 tests denote a logical shift to the right. 8 in hex shifted 1 to the right is 4, shifted 2 bits is 2, shifted 4 bits is 08, shifted 8 bits is 008. And shifted 16 is 00008



These are all the tests for the sra function. These are all correct because the most significant bit is what gets shifted into the vacated spots. The first test is correct because 8 in hex shifted to the right 1 becomes C when all the other spots are filled with ones. The second is correct because 4 in hex shifted to the right then filled with zeros for vacated spots becomes 1. The third is correct because 8 is shifted to the right and the rest are filled with ones. The fourth is correct because it shifted the bits 8 to the right and

filled the rest with zeros when the most significant bit is 0. And the final is correct because it shifted to the right 16 while filling the rest with 1s.

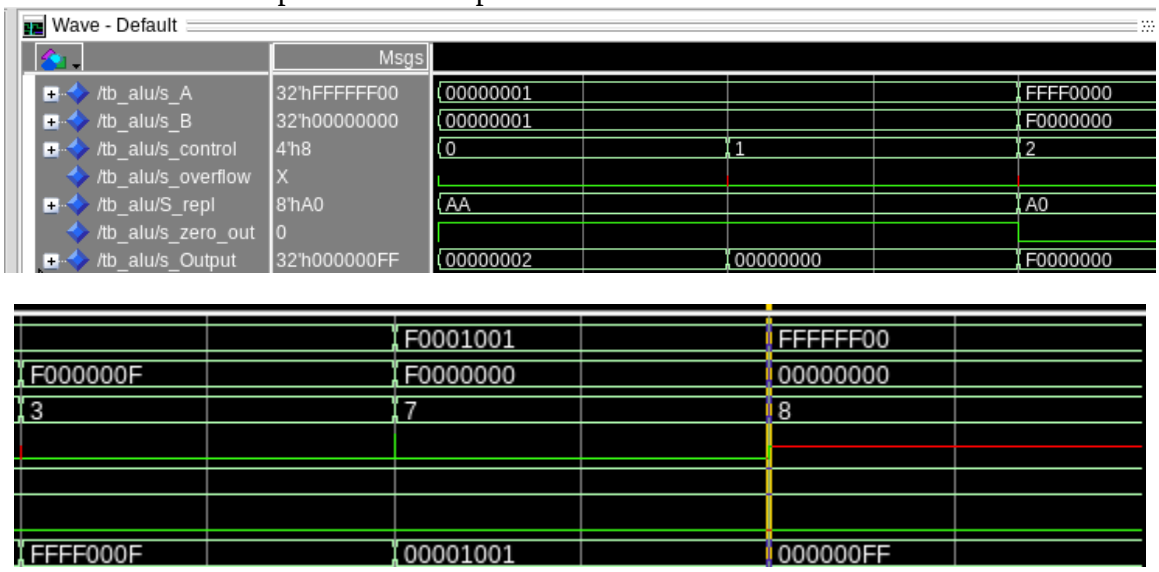
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

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

[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?

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

The waveform corresponds to our expected outcome.



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

[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.

[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?