CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: ____Rian Lamarque____________________

__________Joe Hunter______________________

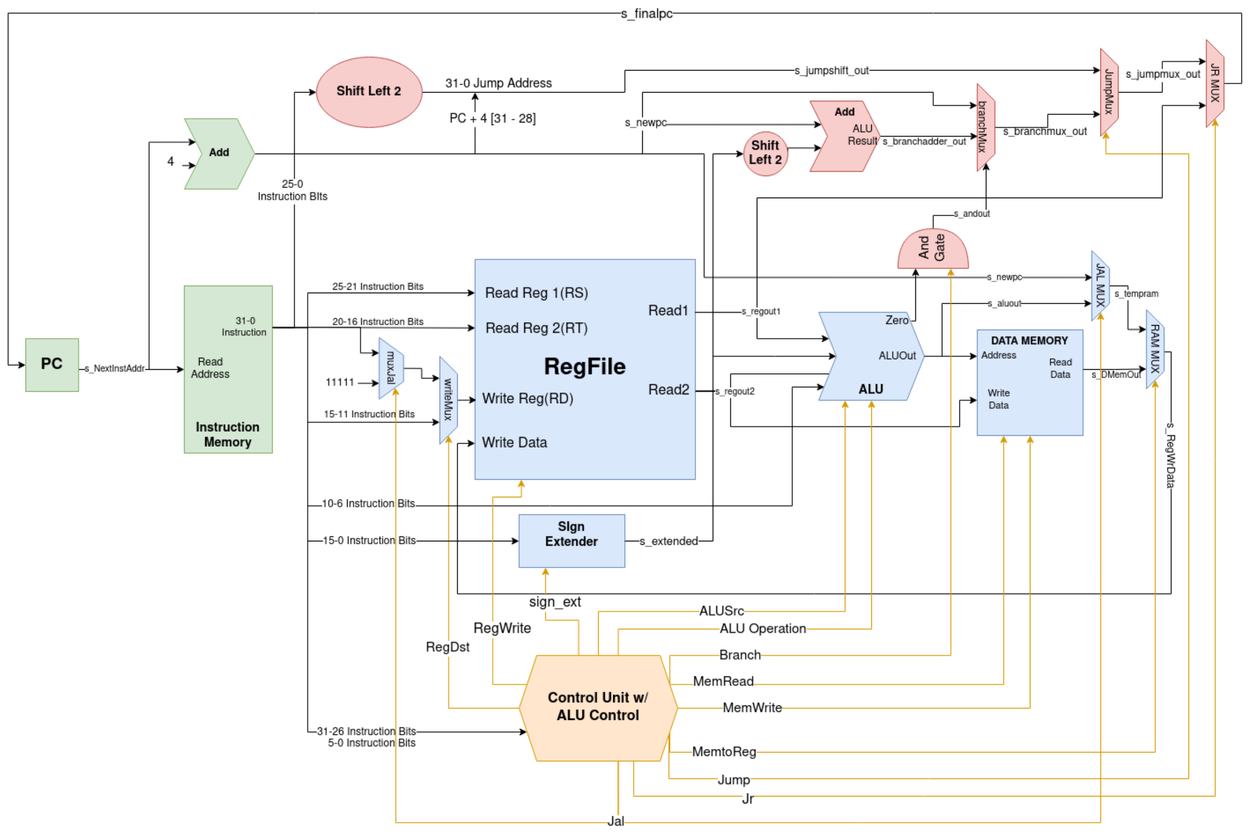
__________David Wolfe____________________________

Project Teams Group #:__Section 5 Group 8_________

Red = Haven't done yet
Green = The question has been answered
Yellow = Answer

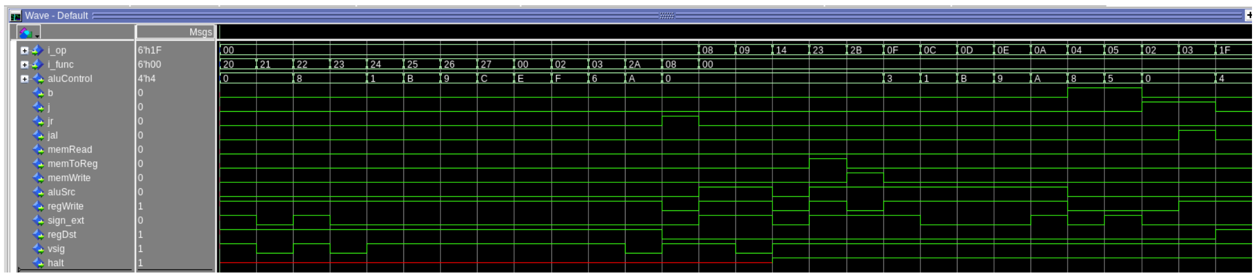
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

[Part 1 (d)] Include your final MIPS processor schematic in your lab report.--Need to add the MIPS processor schematic



[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. Done in our folder named: Proj1_control_signals.xlsx

[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).



As you can see we have the correct control unit signal that correlate correctly with our Mips processor schematic.

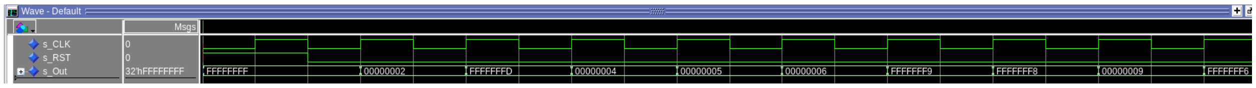
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

Our control flow possibilities need to include standard incrementing by 4, incrementing by a given offset from an immediate value, and jumping to an entire 32 bit address. We need these three different modes because our instruction set supports 4 byte instructions, branching, and jumping. When we aren't branching or jumping we need to increment the fetch instruction address by 4 bytes to advance to the next instruction. When we need to branch we will be jumping forward or backward a set number of instructions. Finally, when we jump we will be given a full address to jump to. Having these three types of instructions dictates that we also have three control flow possibilities.

[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?

Included in the full MIPS processor diagram

[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. --Need a fetch logic testbench



```
s_CLK   0
s_RST   0
s_Out   32'hFFFFFFFF   FFFFFFFF  00000002  FFFFFFFD  00000004  00000005  00000006  FFFFFFF9  FFFFFFF8  00000009  FFFFFFF6
```

Based on the waveform output we can see the fetch logic retrieved values from memory correctly.

[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 data value right and only appends 0s to the most significant bits. However SRA shifts the most significant bits and replaces the shifted bit with whatever the original most significant bit was. MIPS doesn't have a SLA because adding 1s to the least significant bits would be pointless and cause the data values to change.

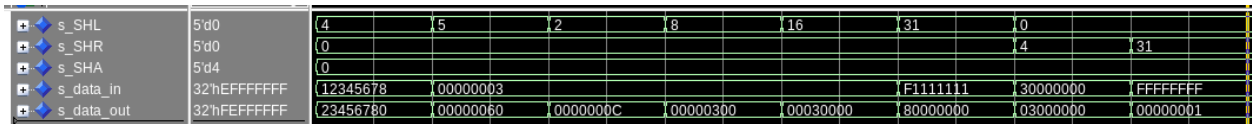
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

Our ALU uses a barrel shifter to handle both arithmetic and logical shifting operations. We have three different 5 bit control signals for each different shift. Each signal tells the shifter how much the data value needs to be shifted.

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

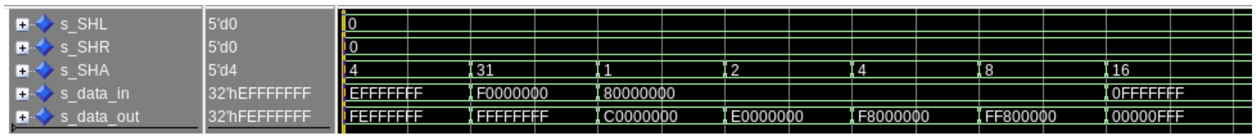
To enhance this shifter I would add a way to reverse the bits, add a new control signal that tells it to use the reverser, send it through the shifter, and add a way to send those bits back through the reverser.

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



```
s_SHL       5'd0           4         5         2         8         16        31        0
s_SHR       5'd0           0                                                           4         31
s_SHA       5'd4           0
s_data_in   32'hEFFFFFFF   12345678  00000003                                F1111111  30000000  FFFFFFFF
s_data_out  32'hFEFFFFFF   23456780  00000060  0000000C  00000300  00030000  80000000  03000000  00000001
```

These values are the shift left logical and shift right logical. SHL and SHR are the bit shift amounts.



```
s_SHL       5'd0           0
s_SHR       5'd0           0
s_SHA       5'd4           4         31        1         2         4         8         16
s_data_in   32'hEFFFFFFF   EFFFFFFF  F0000000  80000000                                0FFFFFFF
s_data_out  32'hFEFFFFFF   FEFFFFFF  FFFFFFFF  C0000000  E0000000  F8000000  FF800000  00000FFF
```

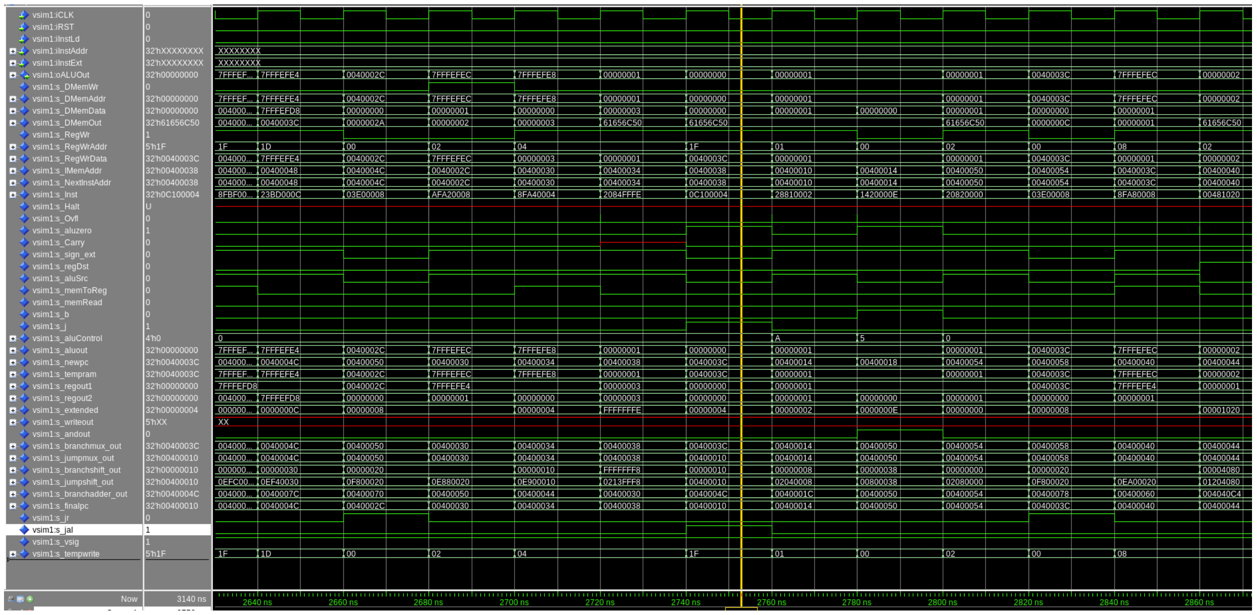
These values are for shift right arithmetic operations. SHA is the bit shift amount.

[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 added three additional muxes into our datapath processor. Two of those muxes are for our JAL instruction. We implemented the first one right before our writeMux and the second one before our RAMmux. The third mux we added was for our JR instruction. It was added right after our JumpMux.

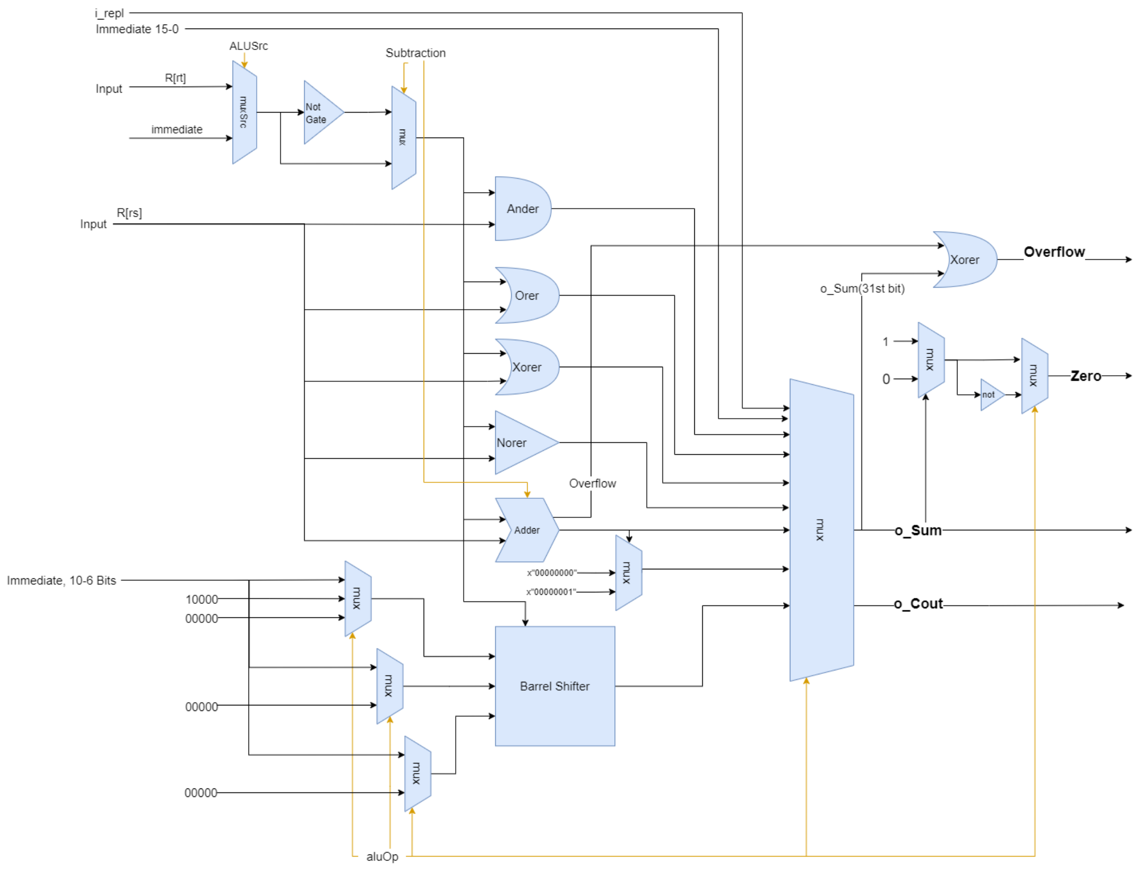
[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

When running different jump, jump and link, and jump register instructions we are able to observe the signals from all three of our new muxs changing accordingly. This is from our Proj1_cf_test.s test file where this particular part in the program shows our muxes working correctly with our different jumps. When jal is 1 we can see the jumpMux, branchMux, and finalPC all have the correct values.



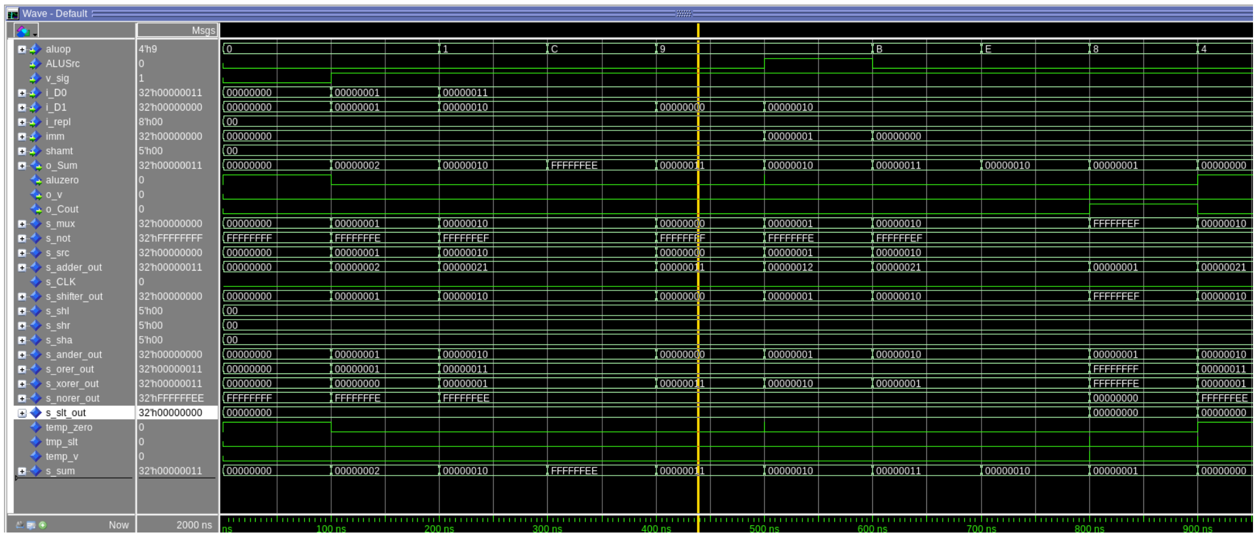
```
vsim1:iCLK                 0
vsim1:iRST                 0
vsim1:iInstLd              0
vsim1:iInstAddr            32'hXXXXXXXX
vsim1:iInstExt             32'hXXXXXXXX
vsim1:oALUOut              32'h00000000
vsim1:s_DMemWr             0
vsim1:s_DMemAddr           32'h00000000
vsim1:s_DMemData           32'h00000000
vsim1:s_DMemOut            32'h61656C50
vsim1:s_RegWr              1
vsim1:s_RegWrAddr          5'h1F
vsim1:s_RegWrData          32'h0040003C
vsim1:s_IMemAddr           32'h00400038
vsim1:s_NextInstAddr       32'h00400038
vsim1:s_Inst               32'h0C100004
vsim1:s_Halt               U
vsim1:s_Ovfl               0
vsim1:s_aluzero            1
vsim1:s_Carry              0
vsim1:s_sign_ext           0
vsim1:s_regDst             0
vsim1:s_aluSrc             0
vsim1:s_memToReg           0
vsim1:s_memRead            0
vsim1:s_b                  0
vsim1:s_j                  1
vsim1:s_aluControl         4'h0
vsim1:s_aluout             32'h00000000
vsim1:s_newpc              32'h0040003C
vsim1:s_tempram            32'h0040003C
vsim1:s_regout1            32'h00000000
vsim1:s_regout2            32'h00000000
vsim1:s_extended           32'h00000004
vsim1:s_writeout           5'hXX
vsim1:s_andout             0
vsim1:s_branchmux_out      32'h0040003C
vsim1:s_jumpmux_out        32'h00400010
vsim1:s_branchshift_out    32'h00000010
vsim1:s_jumpshift_out      32'h00400010
vsim1:s_branchadder_out    32'h0040004C
vsim1:s_finalpc            32'h00400010
vsim1:s_jr                 0
vsim1:s_jal                1
vsim1:s_vsig               1
vsim1:s_tempwrite          5'h1F
                     Now   3140 ns
```

[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 overflow is calculated by looking at the output of the ALU and the overflow from the ripple adder which is also how we are determining the slt instruction. Using these two values you can xor them to find the overflow value of the ALU. Zero is calculated by using the output of the ALU and muxing it with 1 or 0 and then setting an output. From there another mux to pick if you want to output or the ‘not’ of the output.

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



```
Wave - Default
                      Msgs
aluop            4'h9
ALUSrc           0
v_sig            1
i_D0             32'h00000011
i_D1             32'h00000000
i_repl           8'h00
imm              32'h00000000
shamt            5'h00
o_Sum            32'h00000011
aluzero          0
o_v              0
o_Cout           0
s_mux            32'h00000000
s_not            32'hFFFFFFFF
s_src            32'h00000000
s_adder_out      32'h00000011
s_CLK            0
s_shifter_out    32'h00000000
s_shl            5'h00
s_shr            5'h00
s_sha            5'h00
s_ander_out      32'h00000000
s_orer_out       32'h00000011
s_xorer_out      32'h00000011
s_norer_out      32'hFFFFFFEE
s_slt_out        32'h00000000
temp_zero        0
tmp_slt          0
temp_v           0
s_sum            32'h00000011
Now              2000 ns
```

Using this waveform we can validate that many different instructions such as add, and, nor, xor and other instructions produce the correct output from our alu.

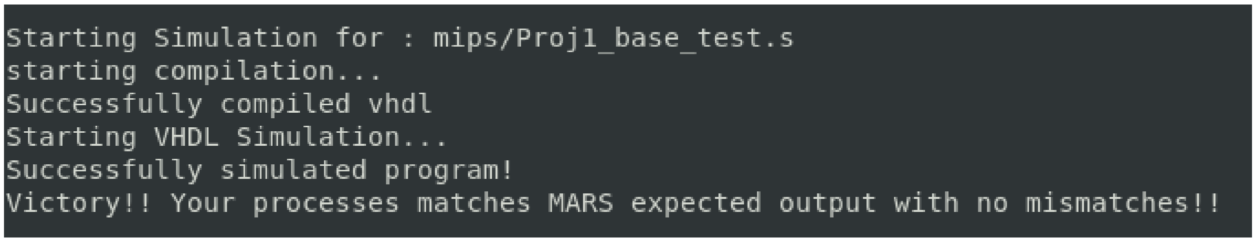
[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

When designing our test bench for the ALU we tried to cover basic and edge cases to ensure we had the most accurate design possible. We started by covering basic arithmetic including add,addi,and,or. After that we tested overflow to ensure we were setting the overflow signal appropriately. To supplement this testing we also ran our ALU against the testing framework to ensure we covered all relevant cases.

[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.

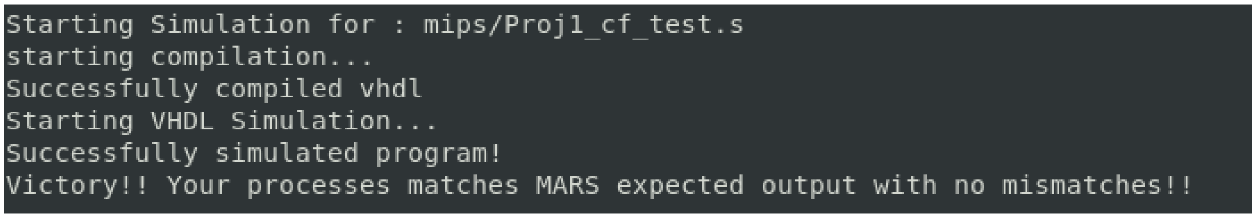
[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially,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

We supplied the .wlf files for all three of these MIPS programs in our 381-project1/wlf_Files directory. We know these are correct because we ran them in the testing framework and they came back with no unexpected mismatches or errors. This can be said for all three programs.



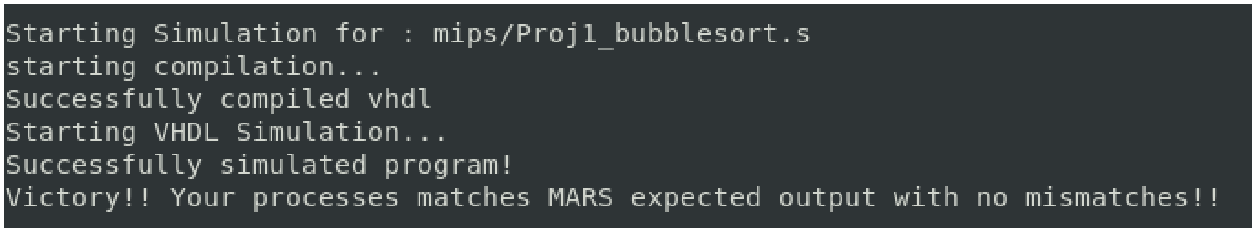
```
Starting Simulation for : mips/Proj1_base_test.s
starting compilation...
Successfully compiled vhdl
Starting VHDL Simulation...
Successfully simulated program!
Victory!! Your processes matches MARS expected output with no mismatches!!
```

[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.



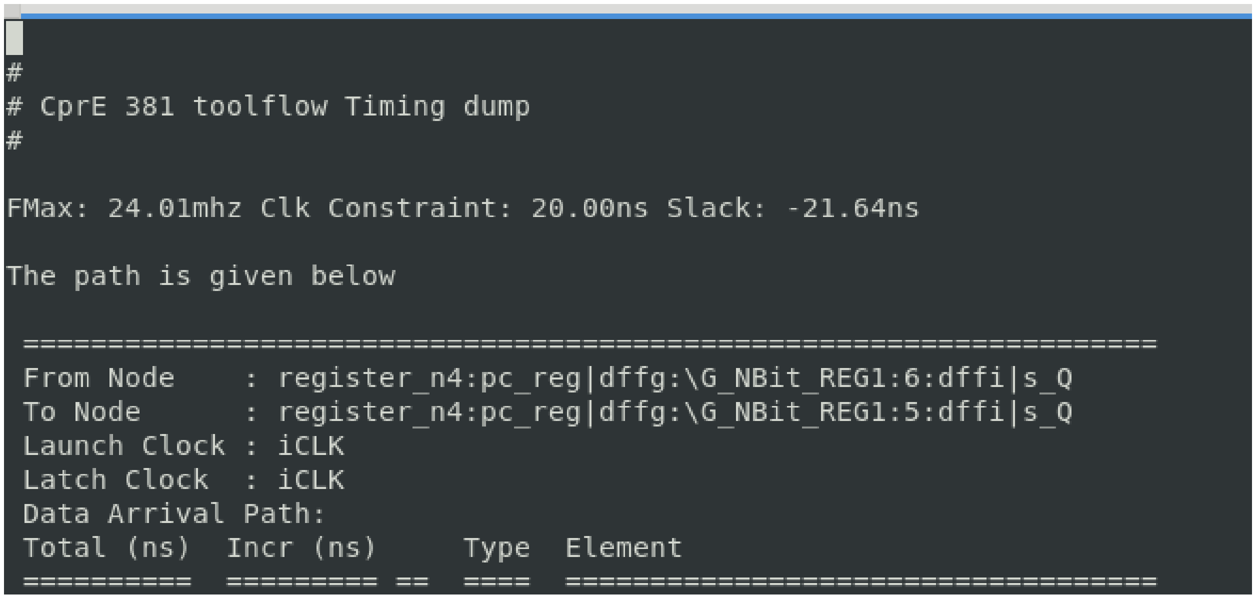
```
Starting Simulation for : mips/Proj1_cf_test.s
starting compilation...
Successfully compiled vhdl
Starting VHDL Simulation...
Successfully simulated program!
Victory!! Your processes matches MARS expected output with no mismatches!!
```

[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.



```
Starting Simulation for : mips/Proj1_bubblesort.s
starting compilation...
Successfully compiled vhdl
Starting VHDL Simulation...
Successfully simulated program!
Victory!! Your processes matches MARS expected output with no mismatches!!
```

[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?



```
#
# CprE 381 toolflow Timing dump
#

FMax: 24.01mhz Clk Constraint: 20.00ns Slack: -21.64ns

The path is given below

 ===================================================================
 From Node    : register_n4:pc_reg|dffg:\G_NBit_REG1:6:dffi|s_Q
 To Node      : register_n4:pc_reg|dffg:\G_NBit_REG1:5:dffi|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
```

After 3.5 hours running on the lab computers we got the results from our synthesis. We had a clock speed of 24.01mhz and our critical path appears to be for the branch instructions. We have attached the full timing.txt fill to our report submission.

I was able to trace the longest path as:
IMem → Control Unit → ALU → ALU Immediate → Check If ALU is Zero → Branch Mux → Jump Mux → Back to the PC Register

This path never went into the Data Memory and therefore cannot be LW/SW.