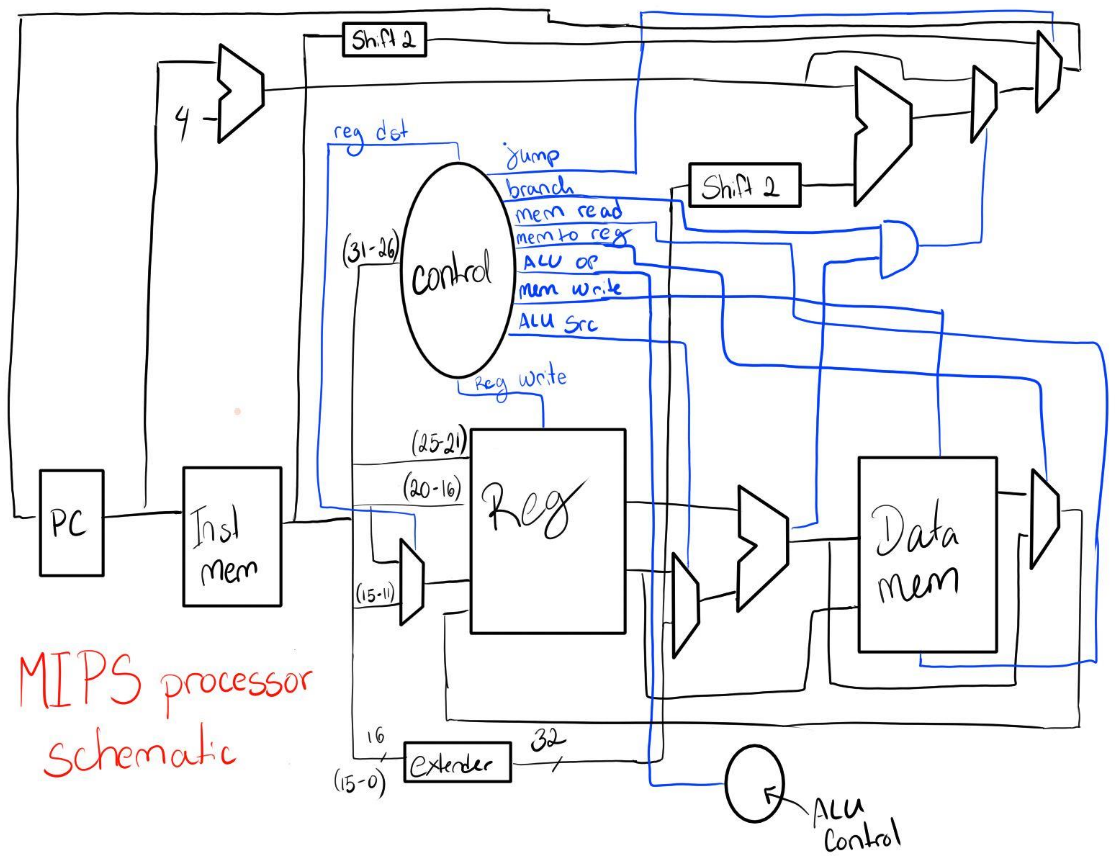
CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Lab Partners	Thai Pham______________________________
	Anh To
	____
	____
	____
	____
Lab Partner #	6

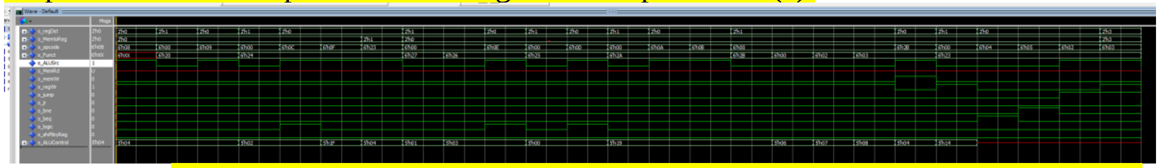
***Refer to the highlighted language in the project 1 instruction for the context of the following questions*.**

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



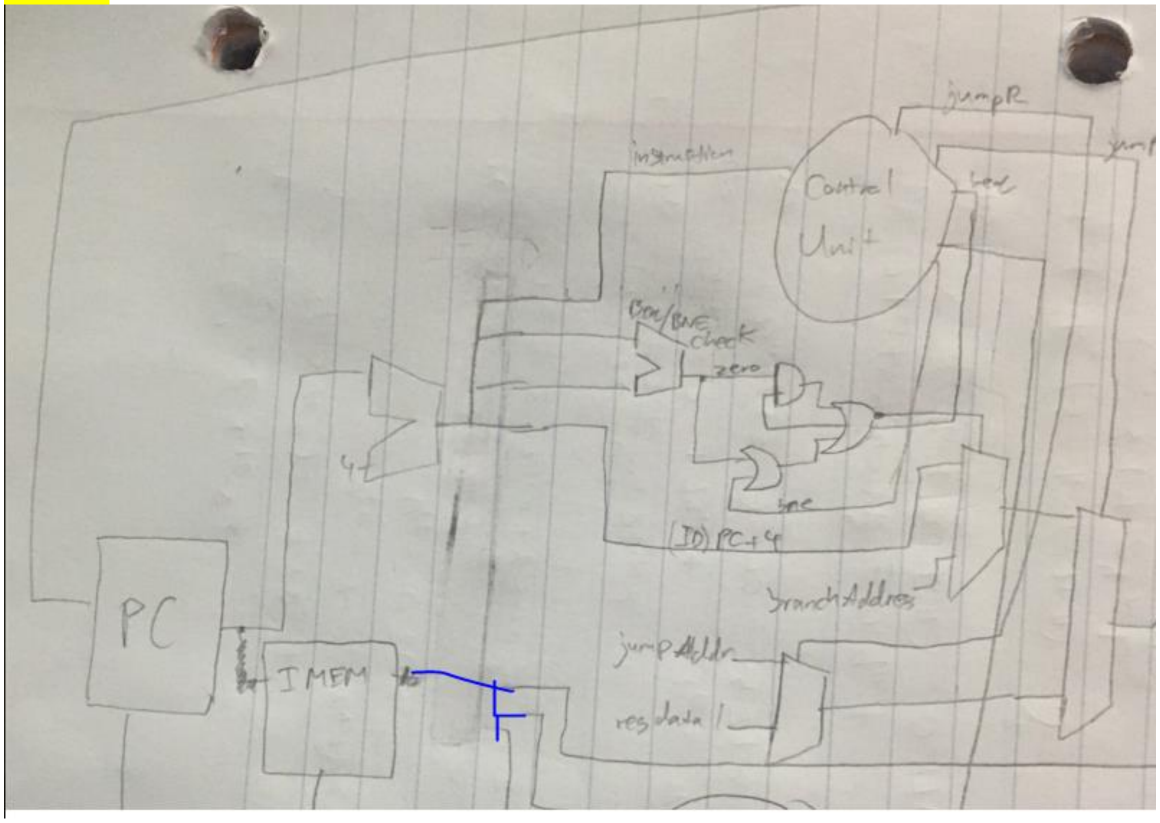
[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your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



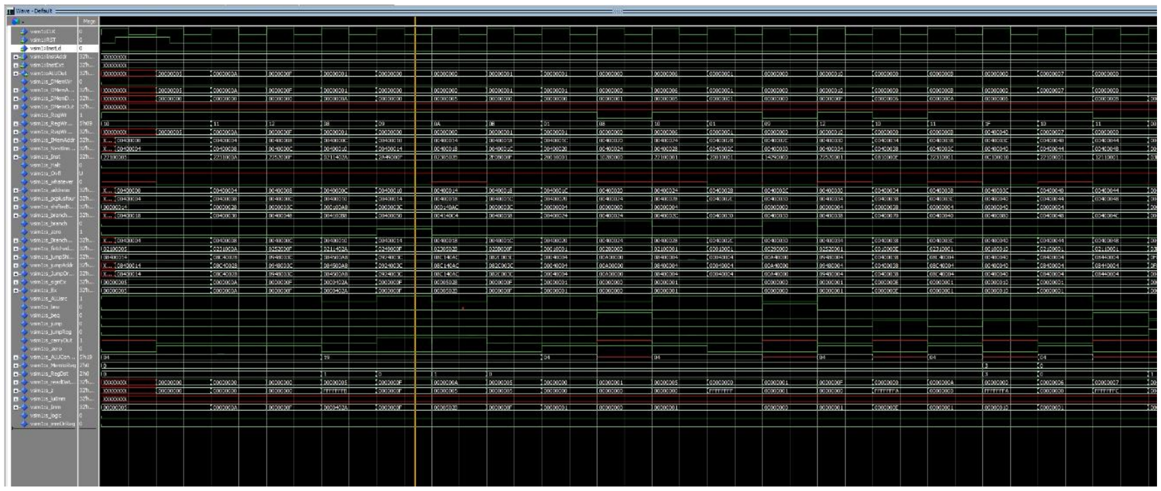
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement. It needed to tell what address to use next(pc+4 vs the branching address or the jumping label) and if the branching conditions are met. The branching and jumping instructions needed what was mentioned. The arithmetic instructions just use the ALU to compute the value that'll go into rd or rt depending on the instruction format. The signal regdst control these conditions and it also change the write reg to 31 for jal as well. Alusrc is use to decide whether to use a value in a reg for r format or an immediate for I format. Memwrite is only on for sw for obvious reasons.

[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?



Signals to know whether jr, j, beq, bne was needed to decide if pc+4 was to be use as the next address or to use the specify address in one of the mentioned instruction and if the condition was met to actually use those address in the case of the branching instructions. A logic signal was use to know whether the immediate should be zero or sign extend and a shifting signal for the shifting instructions.

[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 is the logfile after sucessfully running the cf tests. PC+4 and the address calculations and decisions are correct as well as the signals from the control units that goes into the ALU.

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

Shift Right Logical just shifts the register's bits to the right, filling in the new spots with 0s. Shift Right Arithmetic does a similar operation, but holds the sign bit as the first bit, so if a 1 is the most significant bit, it will hold that bit. The reason MIPS doesn't have a sla instruction is because the least significant bit isn't the sign bit, so it would serve no purpose

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

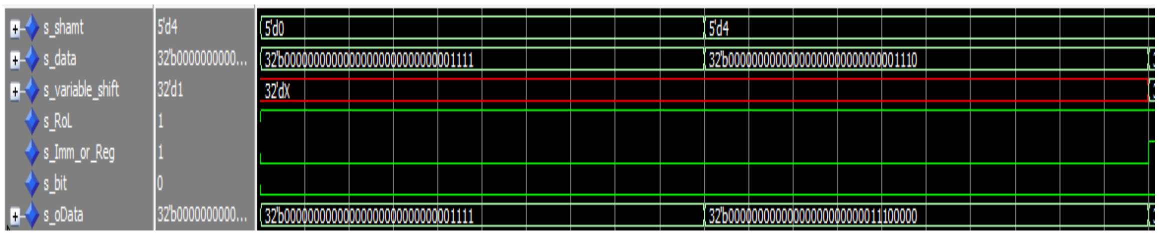
For the shifter, we have a select value called i_bit
If i_Bit is 0, then we use 0 as the incoming bit for Logical shifting

If i_Bit is 1, however, then we use the 31st bit as the incoming bit being shifted in

[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 matter was difficult at first, but turned out to be very easy.
All that needs to be done, is to reverse the data and flip flop it. Shift accordingly, and then reverse it back to normal, thus having a left shift operation through right shifting.

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

SLL



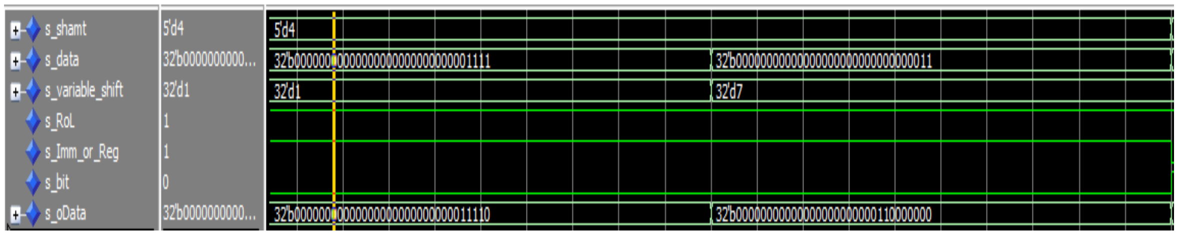
```
s_shamt           5'd4           5'd0                                  5'd4
s_data            32'b0000000000... 32'b00000000000000000000000000001111  32'b00000000000000000000000000001110
s_variable_shift  32'd1          32'dX
s_RoL             1
s_Imm_or_Reg      1
s_bit             0
s_oData           32'b0000000000... 32'b00000000000000000000000000001111  32'b00000000000000000000000011100000
```

The above demonstrates Shift left logic:

- ➢ This is indicated by RoL being 1 for shifting Left
- ➢ S_Imm_or_reg being 0 to use the immediate value for shifting
- ➢ And s_Bit being 0 for logical shifting

- the first set of numbers demonstrates a sll of 0, which obviously does nothing
- the second set of numbers demonstrates a sll of 4, we can observe from the the LSB, where there is only 1, 0. After the shift there is 5 zeroes, indicating a shift of four spaces

SLLV



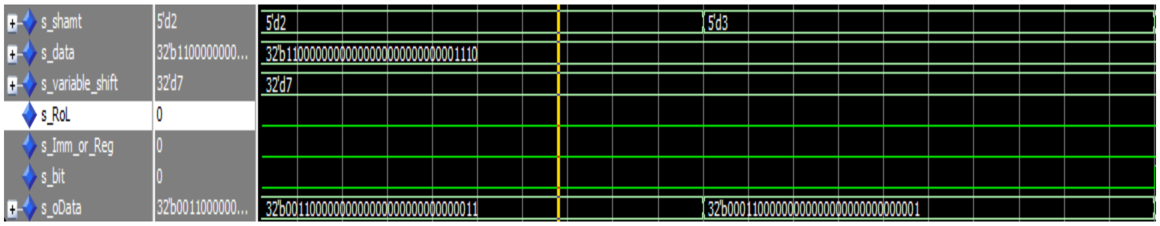
```
s_shamt           5'd4           5'd4
s_data            32'b0000000000... 32'b00000000000000000000000000001111  32'b00000000000000000000000000000011
s_variable_shift  32'd1          32'd1                                 32'd7
s_RoL             1
s_Imm_or_Reg      1
s_bit             0
s_oData           32'b0000000000... 32'b00000000000000000000000000011110  32'b00000000000000000000000110000000
```

The above demonstrates shift left logic variable:

- ➢ This is indicated by the same reasons as regular SLL, except the s_Imm_or_Reg is 1 to indicate we are using a variable instead of an immediate.
- The first set of numbers demonstrates a shift of 1, from the variable shift register being 1, we know it works because instead of shamt (4) shifting 4 times, it is only shifting once
- The second set of numbers demonstrates the same operation, but with value 7, making a shift of 7 places to the left.

SLA is essentially the same as SLL so it is redundant to show

SRL:

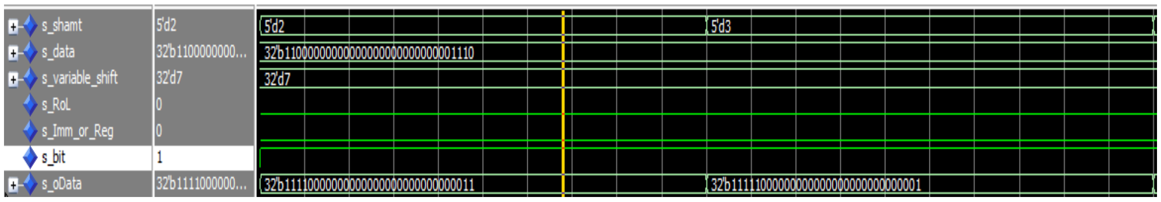


```
s_shamt           5'd2           5'd2                                  5'd3
s_data            32'b1100000000... 32'b11000000000000000000000000001110
s_variable_shift  32'd7          32'd7
s_RoL             0
s_Imm_or_Reg      0
s_bit             0
s_oData           32'b0011000000... 32'b00110000000000000000000000000011  32'b00011000000000000000000000000001
```

The above demonstrates SRL:

- ➢ This is indicated by the s_RoL being 0, to indicate a right shift
- ➢ S_Imm_or_Reg is 0 to indicate using the shamt instead of variable shift
- ➢ S_bit is 0 to indicate the logical shift
- The first set of numbers indicates a shift of 2 to the right, as expected 2 zeroes carry in from the left and the singular zero on the right and one of the 1's on the right dissapear from being shifted "off screen"
- On the second set of numbers, everything is the same except it is a shift of 3 instead, making another 1 go offscreen and a new 0 coming onto the data.
-

SRA:



```
s_shamt          5'd2            5'd2                                       5'd3
s_data           32'b1100000000...  32'b11000000000000000000000000001110
s_variable_shift 32'd7           32'd7
s_RoL            0
s_Imm_or_Reg     0
s_bit            1
s_oData          32'b1111000000...  32'b11110000000000000000000000000011  32'b11111000000000000000000000000001
```

The above demonstrates SRA:

- ➢ This is indicated by the same things as SRL, except s_bit is 1
- The first set of numbers indicates a shift of 2, where we see that instead of 0s, 1s instead come in from the left and we grow from 2 1's coming from the MSB side, to 4, 1s
- The second set of numbers is the same as the first but a shift of 3.

Srav and srlv are redundant to demonstrate since we know sra, srl, and sllv works

[For Teams >4]

[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 two design decisions you had to make.

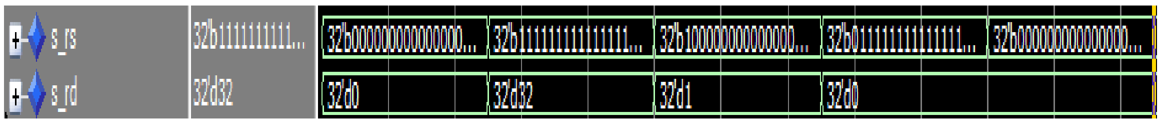
CLO and CLZ have essentially identical design:

The design approach we took was as follows:

- The only thing we need to care about are the leading bits, so we created an algorithm with the leading bits.
- If all the bits are 1, then return 32 otherwise do the following
- Check the first 31 bits, if they're all 1 or 0 depending on what the operation, return 31, and ignore the rest of the bits, if not move on down one bit.

[For Teams >4]

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

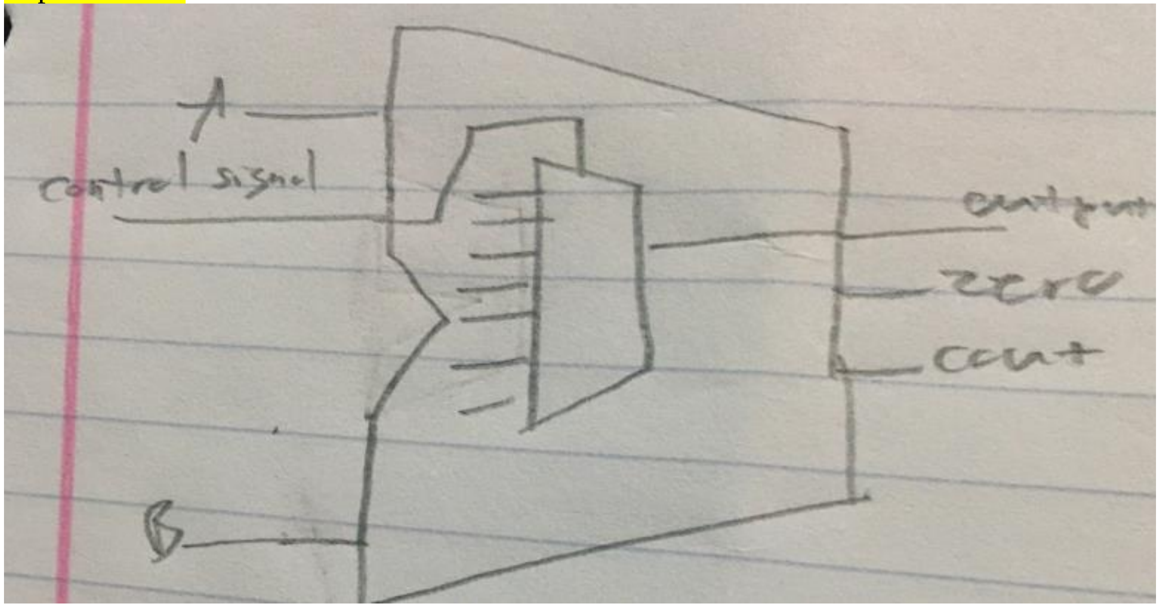


```
s_rs  32'b1111111111...  32'b0000000000000000...  32'b111111111111111...  32'b10000000000000000...  32'b01111111111111111...  32'b0000000000000000...
s_rd  32'd32             32'd0                    32'd32                  32'd1                    32'd0
```

The above represents an example of clo:

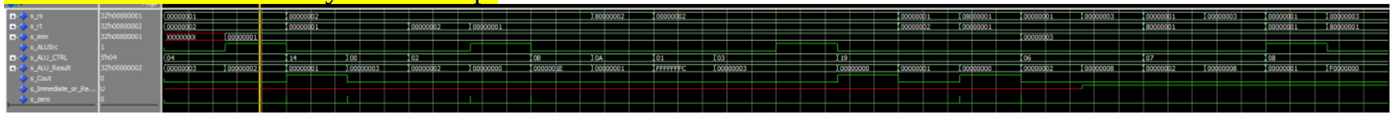
- The first operation represents clo when theres only zeros, return 0 as intended
- The second operation represents 32 ones, returns 32 as expected,
- The third operation represents a mix of 1s and zeroes, with only 1 leading zero and 1 ending zero, showing that only 1 is returned as expected,
- The fourth operation represents one zero with 31 trailing ones, showing 0 should be returned as expected.
- The fifth can't be seen, but 31 leading zeroes and 1 ending 1 returns as expected.

[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?



Zero is ORing all bits and negate result.
Overflow is Cout XOR most sig bit of output. Slt subtract first input with second and check most sig bit of the result. Slt returns true if the difference is negative.

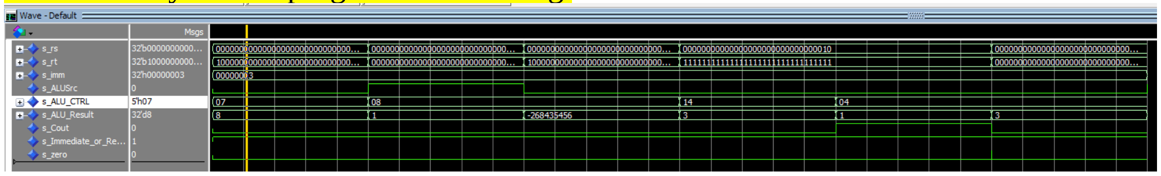
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



```
s_rs                 32'h00000001
s_rt                 32'h00000002
s_imm                32'h00000001
s_ALUSrc             1
s_ALU_CTRL           5'h04
s_ALU_Result         32'h00000002
s_Cout               0
s_Immediate_or_Re... U
s_zero               0
```

The first couple instructions in the testbench are add, addi, sub, or, and, andi, clz, clo, nor, and xor using simple values.

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



```
Wave - Default
                     Msgs
s_rs                 32'b0000000000...
s_rt                 32'b1000000000...
s_imm                32'h00000003
s_ALUSrc             0
s_ALU_CTRL           5'h07
s_ALU_Result         32'd8
s_Cout               0
s_Immediate_or_Re... 1
s_zero               0
```

Shifting instructions were tested with a 1 at the far left and right. Add and sub then used the biggest possible values.

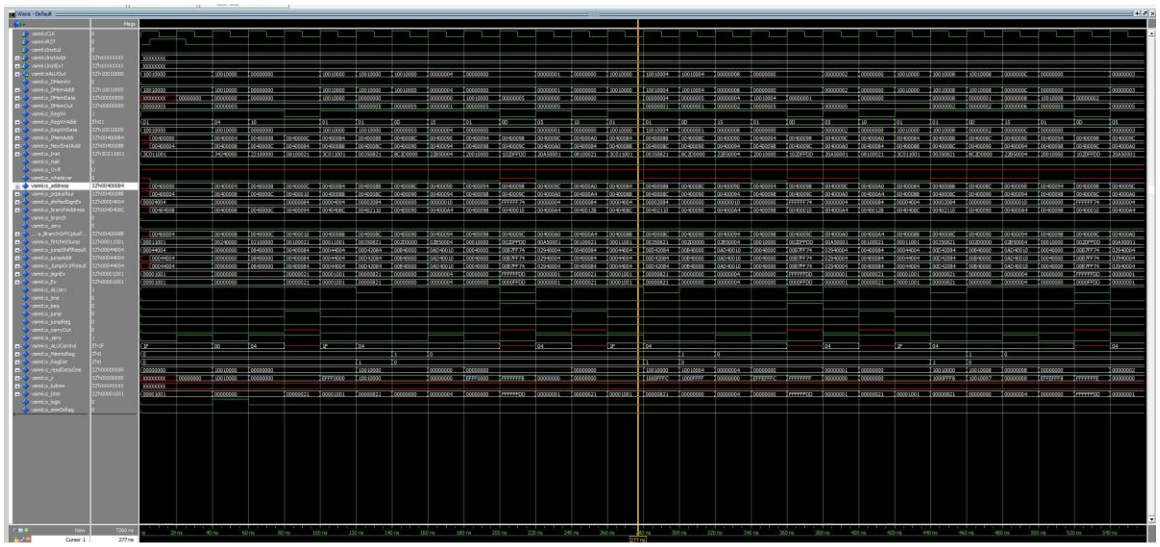
[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[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.

Every thing was correct on the framework. List of instructions are in our mips folder.

[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 2. Name this file Proj1_cf_test.s.

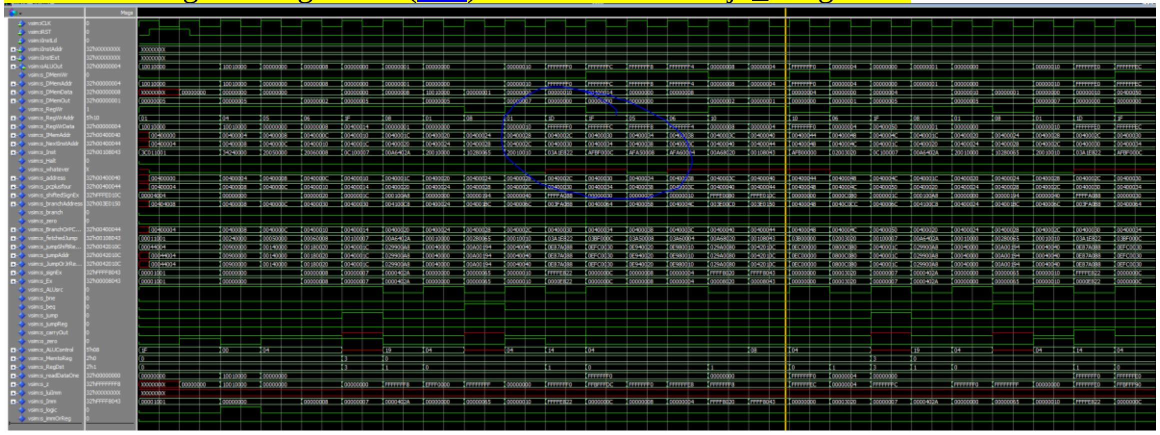
Every thing was correct on the framework. List of instructions are in our mips folder.
[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.



Framework matches mars simulation results.

[For Teams >4]

[Part 3 (d)] Create and test an application that sorts an array with *N* elements using the recursive MergeSort algorithm (link). Name this file Proj1_mergsort.s.



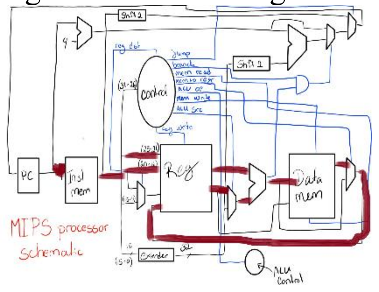
The merge sort is the only didn't work correctly as it puts the wrong value in the right register at cycle 10. It works on mars though.

[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?

FMax: 23.70mhz

The critical path is from imem=>reg>alu(clz)=>dmem=reg

Its obvious that our clz/clo component is the slowest component since they use a really big mux for "counting".


Shft 2
4
reg dst
jump
branch
mem read
memto reg
ALU op
mem write
ALU src
reg write
Shft 2
control
PC
Inst mem
Reg
Data mem
MIPS processor schematic
Extender
ALU control