CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Membei_RSS: _____Edmund Lim_____

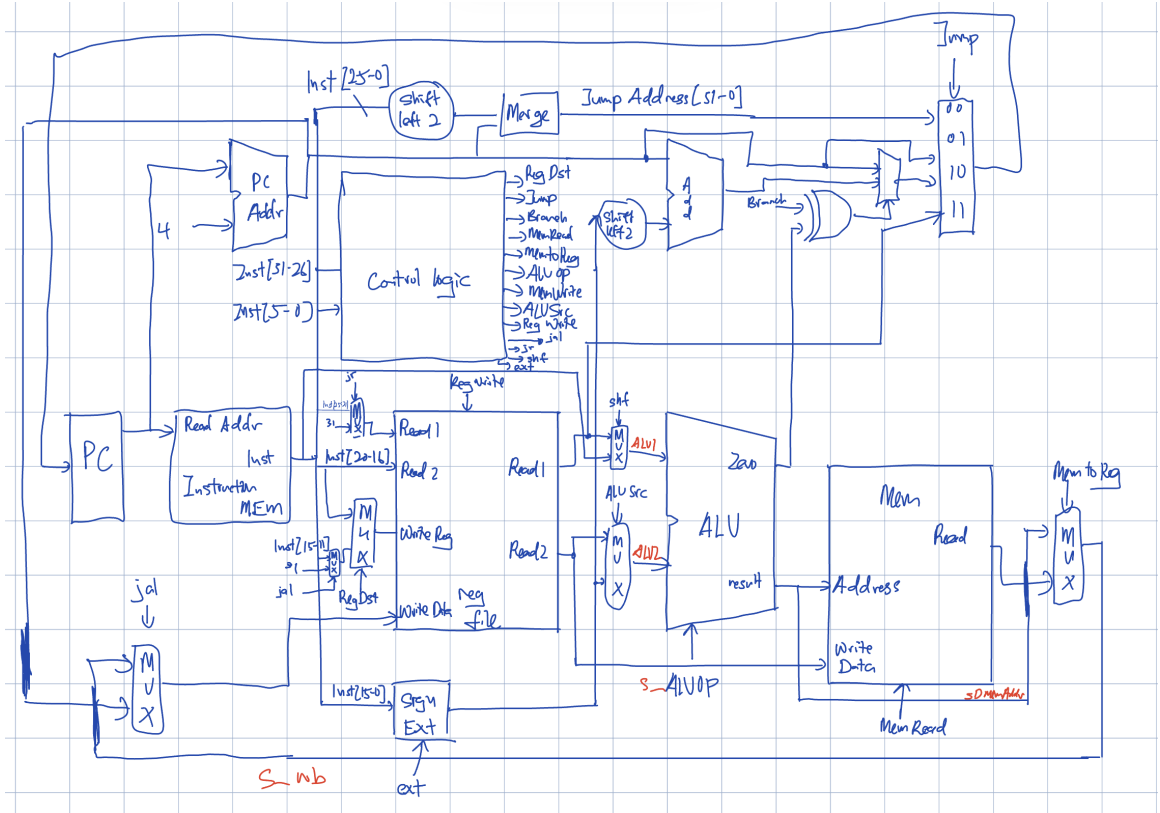
_____Ryan Cook_____

____Neil Prange_____

Project Teams Group #:_____

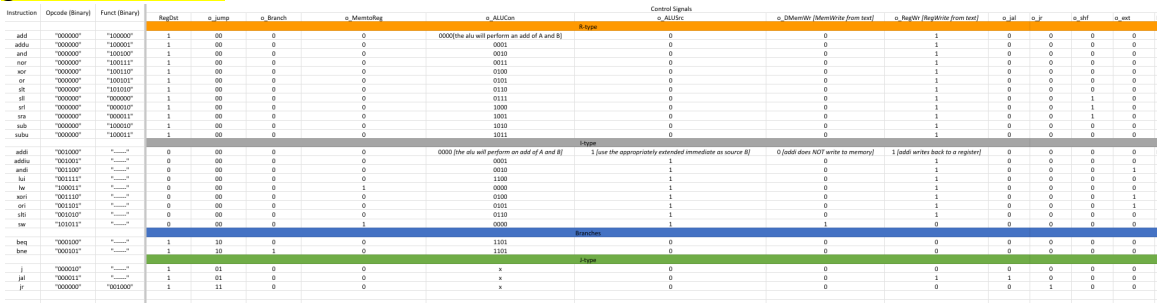
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

[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

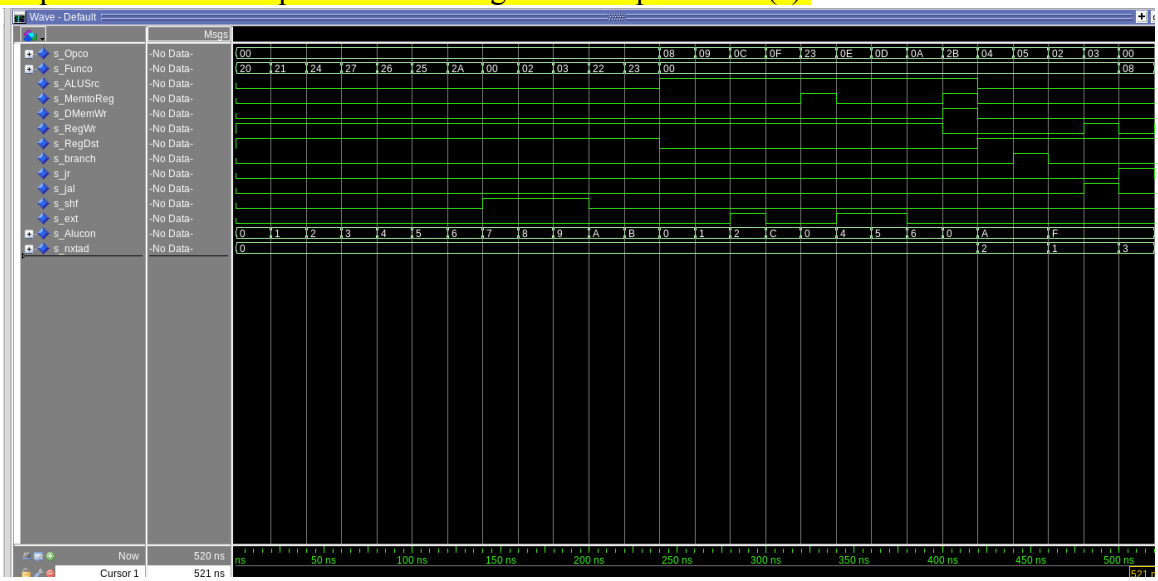
*N*M* table where each row corresponds to the output of the control logic module for a given instruction.



Instruction	Opcode (Binary)	Funct (Binary)	Control Signals											
			RegDst	o_jump	o_Branch	o_MemtoReg	o_ALUCon	o_ALUSrc	o_DMemWr [MemWrite from test]	o_RegWr [RegWrite from test]	o_jal	o_jr	o_shf	o_ext
			R-type											
add	"000000"	"100000"	1	00	0	0	0000(the alu will perform an add of A and B)	0	0	1	0	0	0	0
addu	"000000"	"100001"	1	00	0	0	0001	0	0	1	0	0	0	0
and	"000000"	"100100"	1	00	0	0	0010	0	0	1	0	0	0	0
nor	"000000"	"100111"	1	00	0	0	0011	0	0	1	0	0	0	0
xor	"000000"	"100110"	1	00	0	0	0100	0	0	1	0	0	0	0
or	"000000"	"100101"	1	00	0	0	0101	0	0	1	0	0	0	0
slt	"000000"	"101010"	1	00	0	0	0110	0	0	1	0	0	0	0
sll	"000000"	"000000"	1	00	0	0	0111	0	0	1	0	0	1	0
srl	"000000"	"000010"	1	00	0	0	1000	0	0	1	0	0	1	0
sra	"000000"	"000011"	1	00	0	0	1001	0	0	1	0	0	1	0
sub	"000000"	"100010"	1	00	0	0	1010	0	0	1	0	0	0	0
subu	"000000"	"100011"	1	00	0	0	1011	0	0	1	0	0	0	0
			I-type											
addi	"001000"	"------"	0	00	0	0	0000 [the alu will perform an add of A and B]	1 [use the appropriately extended immediate as source B]	0 [addi does NOT write to memory]	1 [addi writes back to a register]	0	0	0	0
addiu	"001001"	"------"	0	00	0	0	0001	1	0	1	0	0	0	0
andi	"001100"	"------"	0	00	0	0	0010	1	0	1	0	0	0	1
lui	"001111"	"------"	0	00	0	0	1100	1	0	1	0	0	0	0
lw	"100011"	"------"	0	00	0	1	0000	1	0	1	0	0	0	0
xori	"001110"	"------"	0	00	0	0	0100	1	0	1	0	0	0	1
ori	"001101"	"------"	0	00	0	0	0101	1	0	1	0	0	0	1
slti	"001010"	"------"	0	00	0	0	0110	1	0	1	0	0	0	0
sw	"101011"	"------"	0	00	0	1	0000	1	1	0	0	0	0	0
			Branches											
beq	"000100"	"------"	1	10	0	0	1101	0	0	0	0	0	0	0
bne	"000101"	"------"	1	10	1	0	1101	0	0	0	0	0	0	0
			J-type											
j	"000010"	"------"	1	01	0	0	x	0	0	0	0	0	0	0
jal	"000011"	"------"	1	01	0	0	x	0	0	1	1	0	0	0
jr	"000000"	"001000"	1	11	0	0	x	0	0	0	0	1	0	0

https://docs.google.com/spreadsheets/d/1qkAoft2xRUV5EvYDpneEvajP-X2-NqPU/edit?usp=sharing&ouid=106243027726741908963&rtpof=true&sd=true

[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).

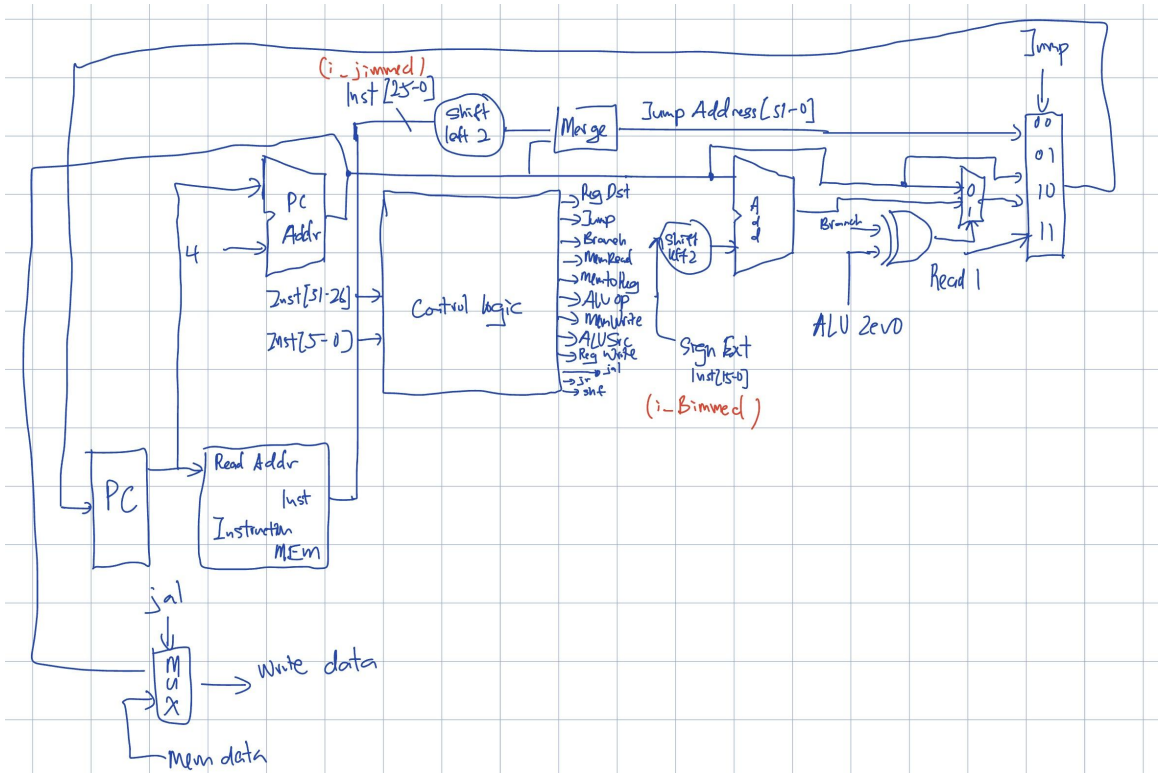


in this test bench the control logic output is tested for every type of instruction required for us to implement.

[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.

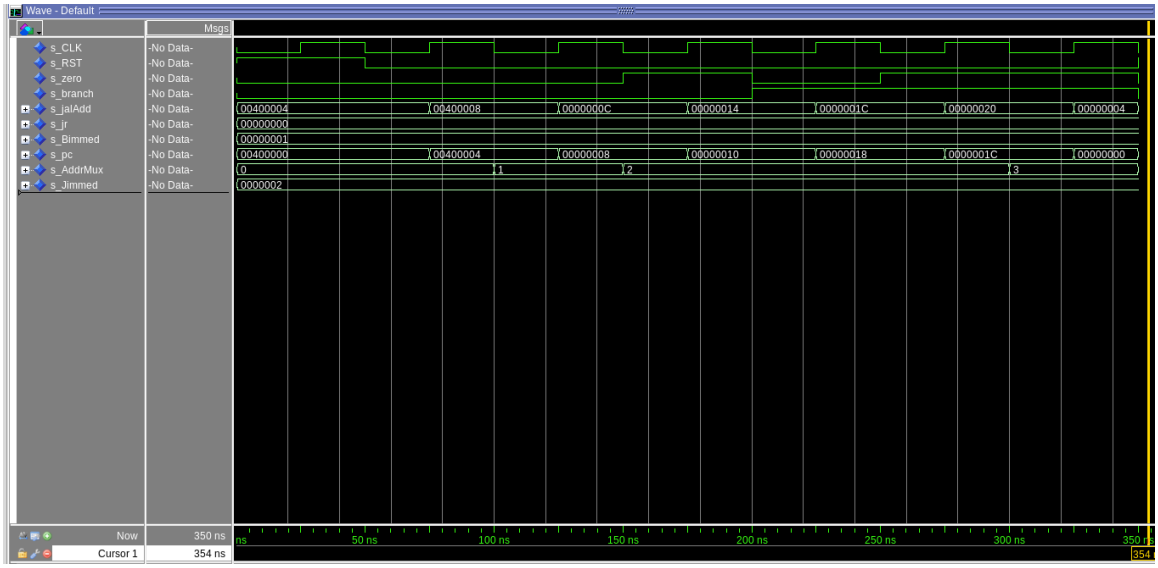
The different possibilities are pc+4, the jump address, branch address, and the jump register address. The pc+4 is for most instructions. Jump address is for normal jump and jump and link. The branch address is for bne and beq for use if the instruction is true. Finally, the jump register address is for the jr instruction.

[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?



branch and zero were preset to be in the fetch module in the implementation; they are xored together to be the signal bit for the branch or pc+4 mux. We created a 2 bit input in jump which controls what the next pc is. Then the other inputs are the two immediates for jump and branch then one last input in jal which controls whether the input to the register file is the address when the instruction is jump and link or mem/alu data when other 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.



In the testbench for the fetch logic it starts out with resetting the register file to the correct starting memory location for the fii_RSSt instruction which is 0x00400000 and that is sent as pc to the Imem. Then next is just an example of a normal R-type instruction which is PC+4 and that is output through pc as 0x00400004 which is correct. The next test is a jump or jump and link instruction where the register saved value would be 0x00400008 and with the jump immediate being 0x0000002 the correct address is 0x00000008 which is what is seen in the output of pc. Next I go through bne and beq where if the output of the zero input is one and branch is zero that is a beq instruction where the jump should take place and the other is when the zero input is 0 and the branch is 1 when the instruction is a bne instruction. both are tested and both have the correct addresses of 0x00000010 and 0x00000018 after those two a test to make sure branch would not happen under the wrong conditions where branch and zero were both one so the pc+4 address was taken which was 0x0000001C and is correct in the waveform then finally jr was tested and the input of jr was 0x00000000 as the input address and o_pc had the same value which shows the jr instruction working 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 - This instruction will shift the value right logically. It will ignore the most significant bit of the data by adding 0 to the most significant bit.

sra - This instruction will shift the value right arithmetic. It will keep the most significant bit value and it is shifting the value. For example, “1110” shifts right 1 time, the result will be “1111”.

The sla instruction does not exist in MIPS because we determine assigned value through the most significant bit.

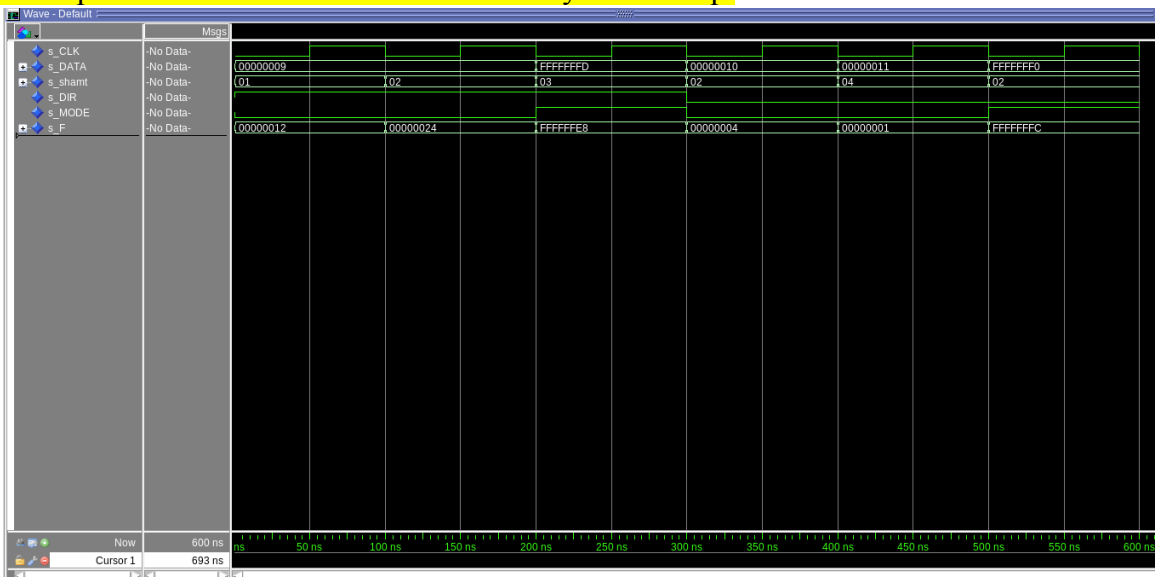
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

I declare two variables, i_DIR & i_MODE, which are used to determine the shifting direction and the shifting mode. i_DIR = 0 represents direction right; i_DIR = 1

represents direction left. i_MODE = 0 represents shift logically; i_MODE = 1 represents shift arithmetic. Then, I use dataflow to assign new shifted values. I fii_RSSt initialized the s_newData with all zeros and two variables s_zeros with all '0' and variable s_ones with all '1'. Then I use conditional statements to determine the type of shift. I combine i_data with s_zeros for logical left shift, logical right shift, and arithmetic left shift by calculating the i_shamt value and replacing othei_RSS with zeros. For the arithmetic right shift, I use a conditional statement to determine whether the most significant bit was '1' or '0' and use the corresponding value to combine with i_DATA.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.
Use two variables, i_DIR and i_MODE, to determine the shifting direction and mode. If the DIR is 1, the program will fii_RSSt assign the value of DATA(k-shamt) into regNew(k), a 32-bits signal, through a for-loop statement (k starts at 31 downto 0). There's another loop to assign the 0s into the rightmost bits that haven't been assigned yet.

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



Case 1 is SLL. It takes the value 9, and shifts logically 1 to the left, outputting 18, or 0x12

Case 2 is the same as previous, except shifting the value by 2, outputting 9 * 2^2 = 36.

Case 3 is sla . It shifts the value -3 by three, maintaining the sign bit. Output is 24.

Case 4 is srl 2. It shifts the value 16 right by two, outputting 4.

Case 5 is srl 4. It takes the value 17, and shifts it right by 4. Since some bits get lost when shifting out, the value rounds down to 1
Case 6 is sra 2. It takes the value -16, and shifts it right by 2, maintaining the sign bit. Output is -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 one design decision you had to make.

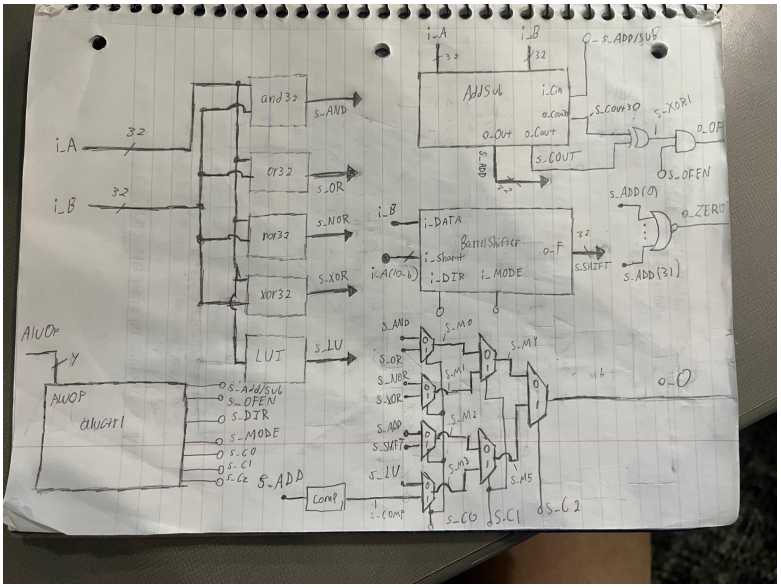
For this design, we ended up using several new components. These include:

- and32, or32, nor32, xor32, which all perform the bitwise operation described by their name
- LUI, which takes a 32 bit value and puts the bottom 16 bits into the top 16, and then forces the bottom 16 to 0
- ALU CTRL, which assigns the control bits for the other units in the ALU
    - One design choice here was the decision to make this value take 4 bits instead of 5
- 32 bit NOR gate, which takes the output of the adder and performs a 32 to 1 nor operation to control the zero flag
- MUXs, which decide which value the ALU output will take based on the instruction
- AddSub, which performs and add or subtract operations

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

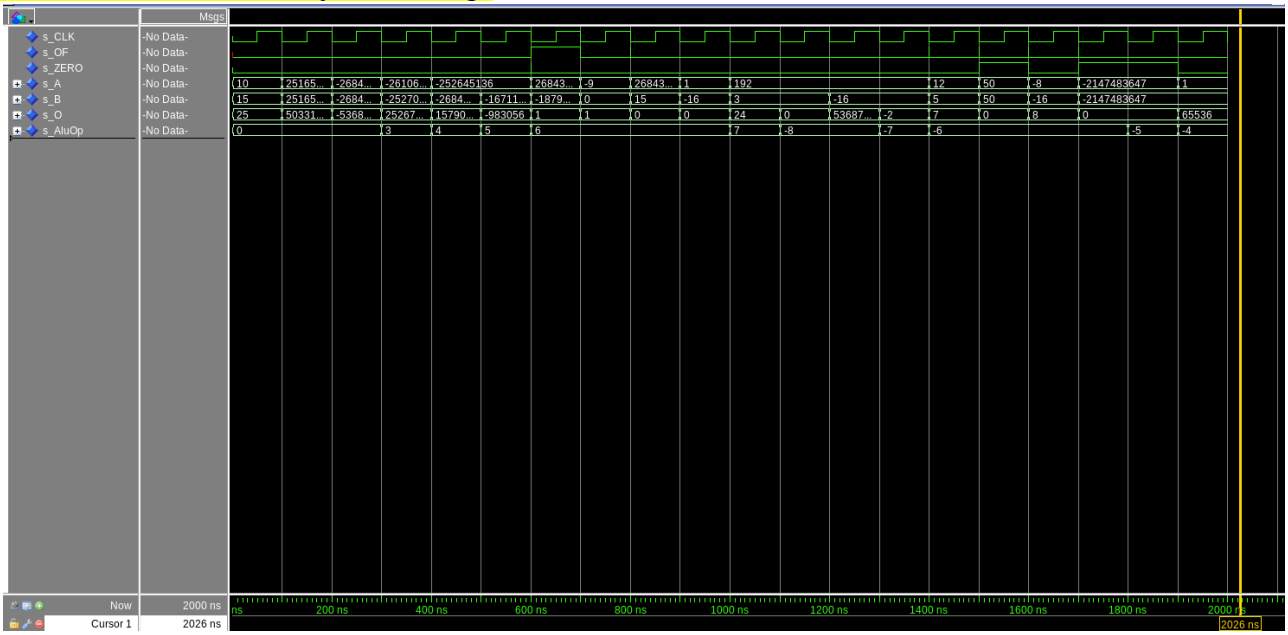
The test for the modules can be seen in the waveform for Part 2. C. V. For example, in cycle 5, you can see the operation of an XOR operation. The output (when converted to decimal) is the bitwise XOR of s_A and s_B.

[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 is calculated based on the output of the adder. It is the XOR of the carry out for FA31 and FA30 in the adder module. Zero is calculated based on a 32 input nor gate of the output of the adder. SLT is implemented using the "Comp" component. It takes the output of subtraction from the AddSub module, and makes the output 0x00000001 if the output is negative, and 0x00000000 if it is positive.

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



The waveforms seen above correspond to the different instructions needed to be carried out by the ALU. For example the fii_RSSt clock cycle performs a basic ADD operation. This is the operation performed by add, addi, lw, and sw. As you can see, it adds 10 + 15

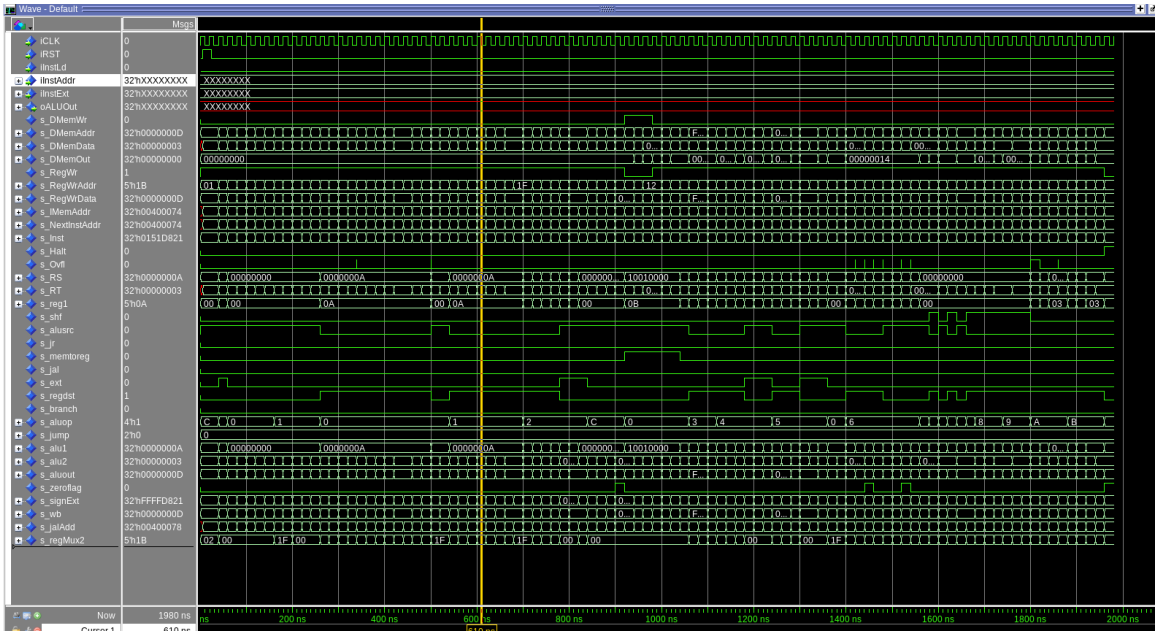
= 25. The next 19 instructions each correspond to a case seen in the ALUTB file. It tests the 13 different operations seen on the control excel sheet.

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

The waveform seen in the previous part is comprehensive as it thoroughly tests all possible ALU operations. It also tests certain instructions multiple times, such as multiple subtract operations to see if the zero flag and overflow flag output properly.

[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.

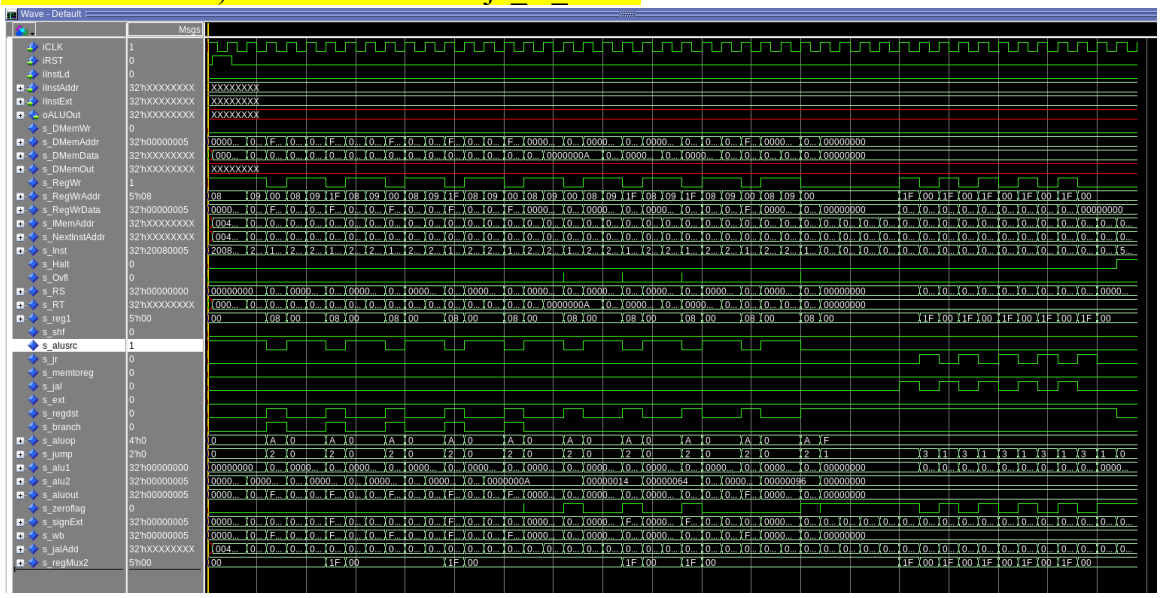
[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially,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



In this every logical/arithmetic instruction is tested starting with addi and ending with subu. Also every register inside the registerfile is tested by loading a number into them and then another instruction is used on every register too. It starts with 5 addi, then 5

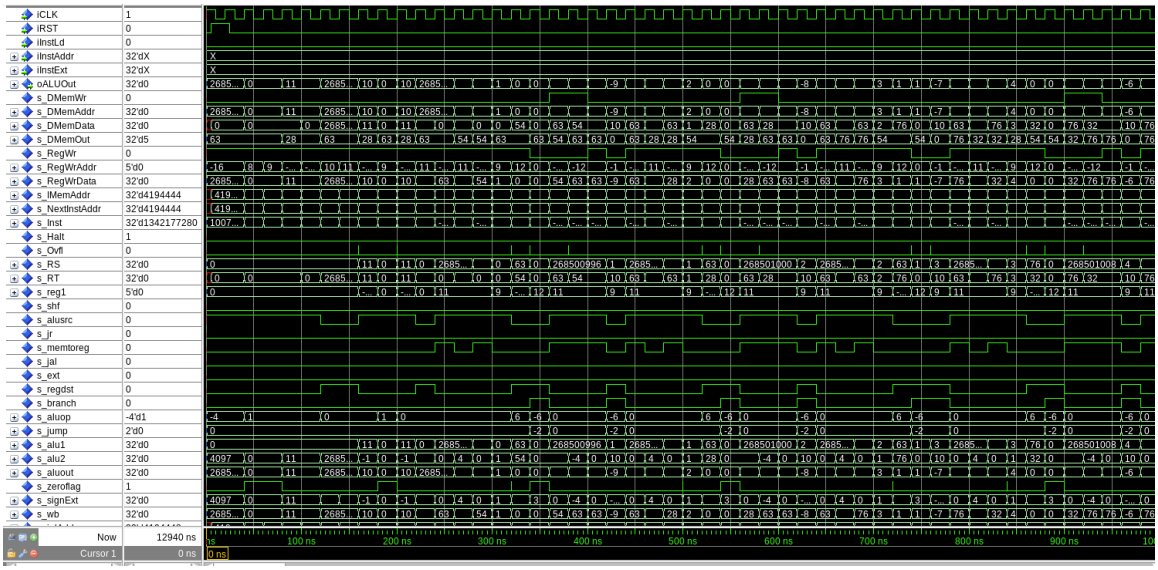
addiu, after that 11 adds another 2 addi and to wrap up the adds 8 addu are called. then the next is 4 and and then 3 andi instructions then the last 11 blocks of instructions are 4 lui, 3 sw, 3 lw, an addi, 3 nor, 3 xor, 3 xori, 3 or, 3 ori, 2 addi and 4 slt, and finally 4 slti.

[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.



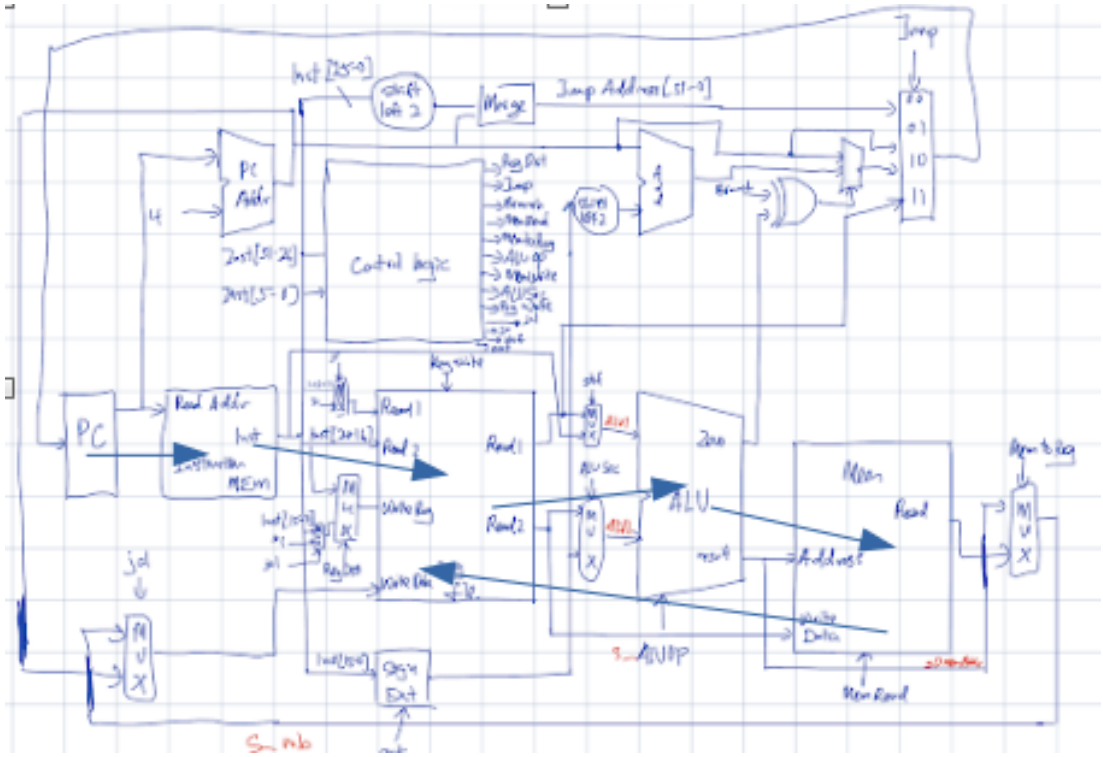
In this file it tests the commands bne, beq, j, jal, and jr which all get run 5 times each. Fii_RSSt, it runs bne 5 times all which run correctly. Second it runs beq all run correctly. Third, the fii_RSSt 4 jump instructions are called and run correctly. Then, Finally, the jal, jr, and the final jump are all called where it would call jal the jr where it would return to the next jal then finally returning to the last jump where it jumps to the end and finishes the program.

[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.



This program takes in an array of integei_RSS in memory and a value N, and sorts N elements of the array using a bubble sort algorithm. You can see when loops are iterated using the zero flag. You can see the outer loop begin iterating in cycle 18. It loads an immediate value of 0 into $t1, as can be seen by RegWrData being 0, it then performs, addi $2, $2, -1, moves $s0 into $t3, and then starts the 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 schematics. What components would you focus on to improve the frequency?



The maximum frequency that our processor can run at is 22.81 mhz. Our critical path starts with fetch logic to Imem to the register file to the ALU to Dmem back to the register file. To improve the frequency we would focus on changing our adder from a ripple carry adder to a fast adder to improve the ALU speed. Currently the biggest slow down to our processor is the two memories with Imem and Dmem so decreasing the time to get information from them would also speed up the frequency.