CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 1 Report

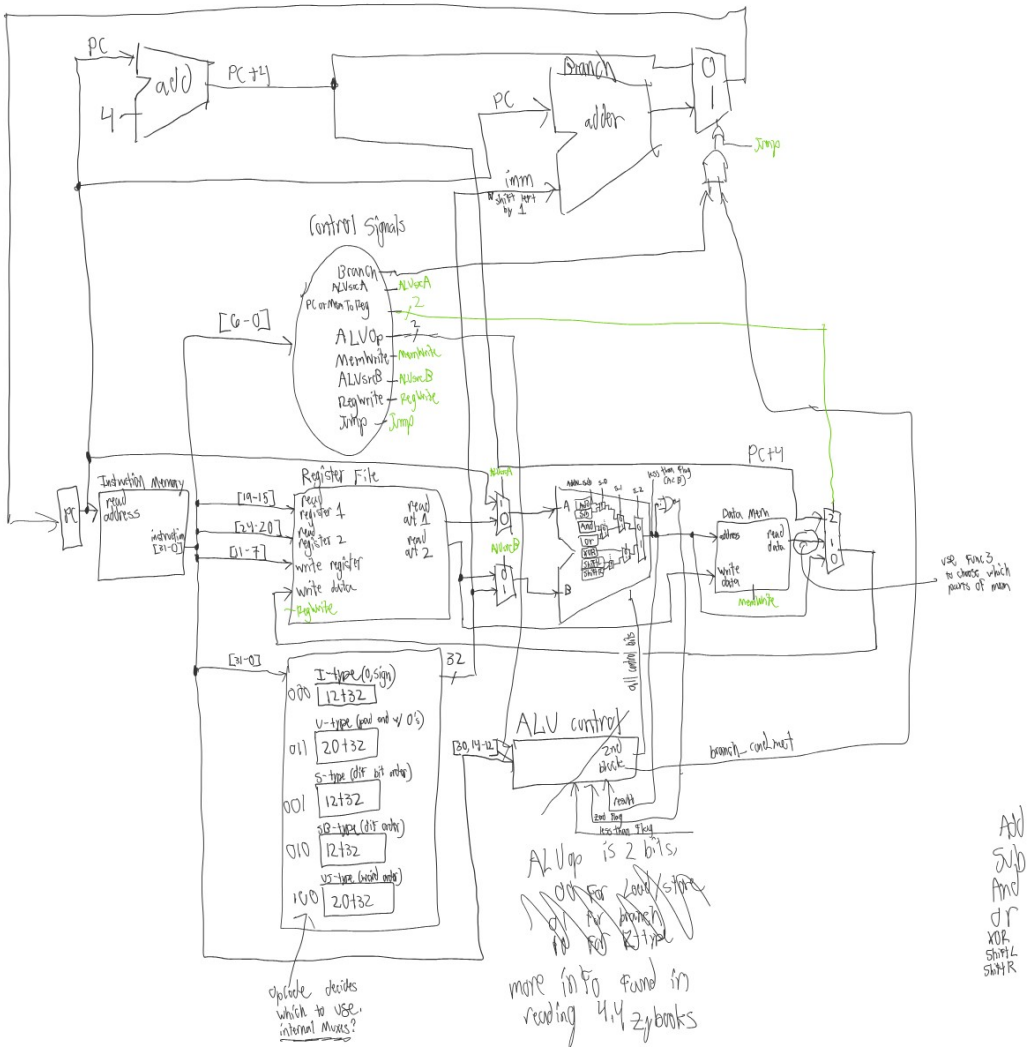
Team Members: Gabe Mankowski

Blane Fullmer

Project Teams Group #: C_07

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

[Part 2 (d)] Include your final RISC-V processor schematic in your lab report.

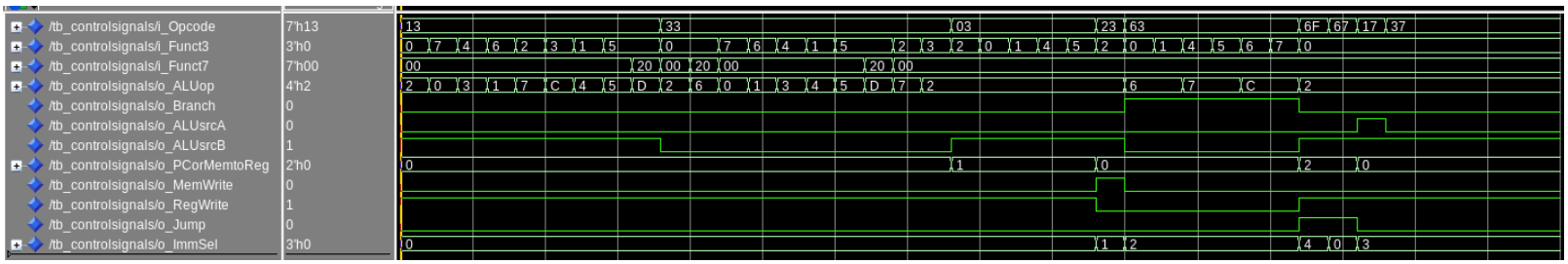


[Part 3.1.a.] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your 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.

A screenshot is included here:

	E	F	G	H	I	J	K	L	M
1						Control Signals			
2	ALUscrA	ALUsrcB	ALUop (4 bits)	MemWrite	RegWrite	PcorMemtoReg (2 bits)	Branch	Jump	ImmSel (3bits)
3	0	1	“0010”	0	1	“00”	0	0	“000”
4	0	0	“0010”	0	1	“00”	0	0	“---”
5	0	0	“0000”	0	1	“00”	0	0	“---”
6	0	1	“0000”	0	1	“00”	0	0	“000”
7	0	1	“0010”	0	1	“00”	0	0	“011”
8	0	1	“0010”	0	1	“01”	0	0	“000”
9	0	0	“0011”	0	1	“00”	0	0	“---”
10	0	1	“0011”	0	1	“00”	0	0	“000”
11	0	0	“0001”	0	1	“00”	0	0	“---”
12	0	1	“0001”	0	1	“00”	0	0	“000”
13	0	0	“0111”	0	1	“00”	0	0	“---”
14	0	1	“0111”	0	1	“00”	0	0	“000”
15	0	1	“1000”	0	1	“00”	0	0	“000”
16	0	0	“0100”	0	1	“00”	0	0	“---”
17	0	0	“0101”	0	1	“00”	0	0	“---”
18	0	0	“1101”	0	1	“00”	0	0	“---”
19	0	0	“0010”	1	0	“-”	0	0	“001”
20	0	0	“0110”	0	1	“00”	0	0	“---”
21	0	0	“0110”	0	0	“-”	1	0	“010”
22	0	0	“0110”	0	0	“-”	1	0	“010”
23	0	0	“0111”	0	0	“-”	1	0	“010”
24	0	0	“0111”	0	0	“-”	1	0	“010”
25	0	0	“1000”	0	0	“-”	1	0	“010”
26	0	0	“1000”	0	0	“-”	1	0	“010”
27	0	1	“----”	0	1	“10”	0	1	“100”
28	0	1	“0010”	0	1	“10”	0	1	“000”
29	0	1	“0010”	0	1	“01”	0	0	“000”
30	0	1	“0010”	0	1	“01”	0	0	“000”
31	0	1	“0010”	0	1	“01”	0	0	“000”
32	0	1	“0010”	0	1	“01”	0	0	“000”
33	0	1	“0100”	0	1	“00”	0	0	“000”
34	0	1	“0101”	0	1	“00”	0	0	“000”
35	0	1	“1101”	0	1	“00”	0	0	“000”
36	1	1	“0010”	0	1	“00”	0	0	“011”

[Part 3.1.(b)] 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 3.2. (a)] 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.

There are two behaviors that the fetch module needs to implement based on a set of three control signals. First, it needs to take the value of the program counter, increase it by 4, representing a word, and then write that larger value back to the program counter. Secondly, it needs to be able to jump to an address in memory based on a calculated offset, which is supplied externally by the "extender". In practice, this is done by wiring the output of the PC into the inputs of two different adder units. One adder is wired to always add 4, and the other is given that immediate offset value. The two modules are muxed, and selected between via a small assortment of logic gates operating on the control signals. This controlls which adder propagates back to the PC.

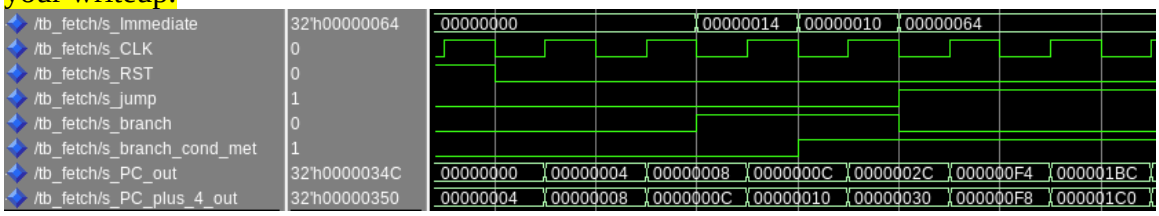
[Part 3.2. (b)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

The Fetch functionality needs three control signals, all of which get sent to the select of the adder mux through a set of logic gates. This Boolean logic as written is

(C_Branch and C_Branch_Cond_Met) or C_Jump

This implements the functionality flawlessly. If there is a branch, and the condition to take it is met, then the mux propagates the imm-adder. If it is a jump instruction, the mux propagates the imm-adder. If neither are true, the mux propagates the +4 adder, and the program reads the next instruction in the linear sequence.

[Part 3.2.(c)] 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.



Only when the two branch controls were high, or when the jump control was high did the shifted value on the immediate line get added to the PC. All other times, the PC was incremented by 4, as is expected.

[Part 3.3.1.(a)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does RISC-V not have a `sla` instruction?

Srl always shifts "in" a 0, to fill the most significant bit of the register. This is not ideal when you want to use the "division" property of shifting on a negative number, because the most significant bit is the sign bit. By replacing it with a zero, we have divided by two, and made it positive. Sra preserves the most significant bit, and shifts in it's value, instead of only zero. There is no sla instruction, because this would not be a meaningful thing. It would amount to a x*2+1 macro.

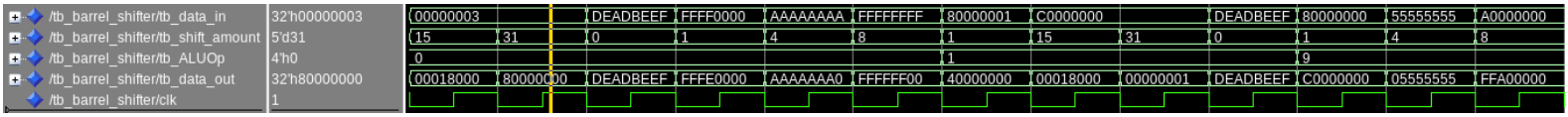
[Part 3.3.1.(b)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.
The code is looking at a control bit generated by the sra instruction, and the most significant bit. Instead of tying the empty “shift in” lines of my design to a constant source of zero, they are tied to an and gate that is evaluating that control bit and the sign bit. If the instruction is not sra, this control bit is zero, which propagates zero into the shifter.

[Part 3.3.1.(c)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.
The ability two shift in both directions was accomplished by reversing the 32 bit register via a row of muxes, then shifting the backwards string to the right. The string of bits is then unreserved at the end of the right shift.

[Part 3.3.1.(d)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.

[Part 3.3.2.(a)] 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.
My design approach for this component started by considering what it would take to make a shifter that had the left and right functionality that was smaller, acting on only two bits of data. I sketched this out and manually traced the bits flowing through it. I then tried to draw and trace a larger one that acted on four bits, and noted how the thing scaled between the two. Essentially for every doubling of bits, I needed a row of muxes that shifted in the control bit “twice as soon”. This leads to a stage that takes the regular or reversed input, and shifts it 16 over, followed by a stage that does 8, 4, 2, and then 1.

[Part 3.3.2.(b)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.
The waveform correctly shows the shifter taking the value and shifting it left when the ALUOP = 0, shifting right logical when ALUOP = 1, and shifting right arithmetic when ALUOP = 9.



[Part 3.3.3] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: How is Zero calculated? How is `slt` implemented?

Zero is being calculated manually within the ALU by comparing the result signal to a constant hex mask of all zeros. This sets o_Zero to 1 when the result is all zero, and otherwise sets that output to 0 with an else statement.

Slt sets the result to one if i_A < i_B, and 0 otherwise. When the i_Control signal is 0111, the i_Control(2) bit is fed to the i_Cin port of the add sub unit, which sets it to subtract A -B. It then XORs the sign bit of result of the subtraction with the overflow flag. This works because if there is no overflow, the sign bit of the result is the output. If A-B is a negative value, the sign bit will be one, and we can conclude that B was larger than A. If overflow did occur, the sign bit will be the opposite of what we want, requiring a reversal via XOR.

[Part 3.3.5] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.
The test bench waveform correctly executes one of every operation the ALU is meant to perform. In order, it checks the following functionalities:
1. Add two positive numbers: 5+3 = 8
2. Add two negative numbers -5 + -3 = -8
3. Sub two positive numbers with positive result 10 – 4 = 6
4. Sub two positive numbers with negative result 5 – 8 = -3
5. And two masks
6. Or two masks
7. Xor two masks
8. SLT 5 < 10: 1
9. SLT -10 < -5: 1
10. SLTU
11. Check the zero flag via sub 5 – 5 = 0
12. give an invalid ALU control combination, all 1111. triggers default case.

[Part 3.3.8] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.
The testing procedure demonstrates all instructions the ALU is meant to implement.

[Part 4] 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 4.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 4.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 4.c] Create and test an application that sorts an array with *N* elements using the MergeSort algorithm (link). Name this file Proj1_mergesort.s.

[Part 5] 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?

Our maximum frequency reported after synthesis was 21.82mhz. The specific instruction being executed during this path was lw. The value is being propagated through our ALU, and because all operations are muxed from, the operations have to be done and propagate even if they aren't being selected through to the output. We can re design the ALU, and likely see a sharp increase in performance.