CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 1 Report

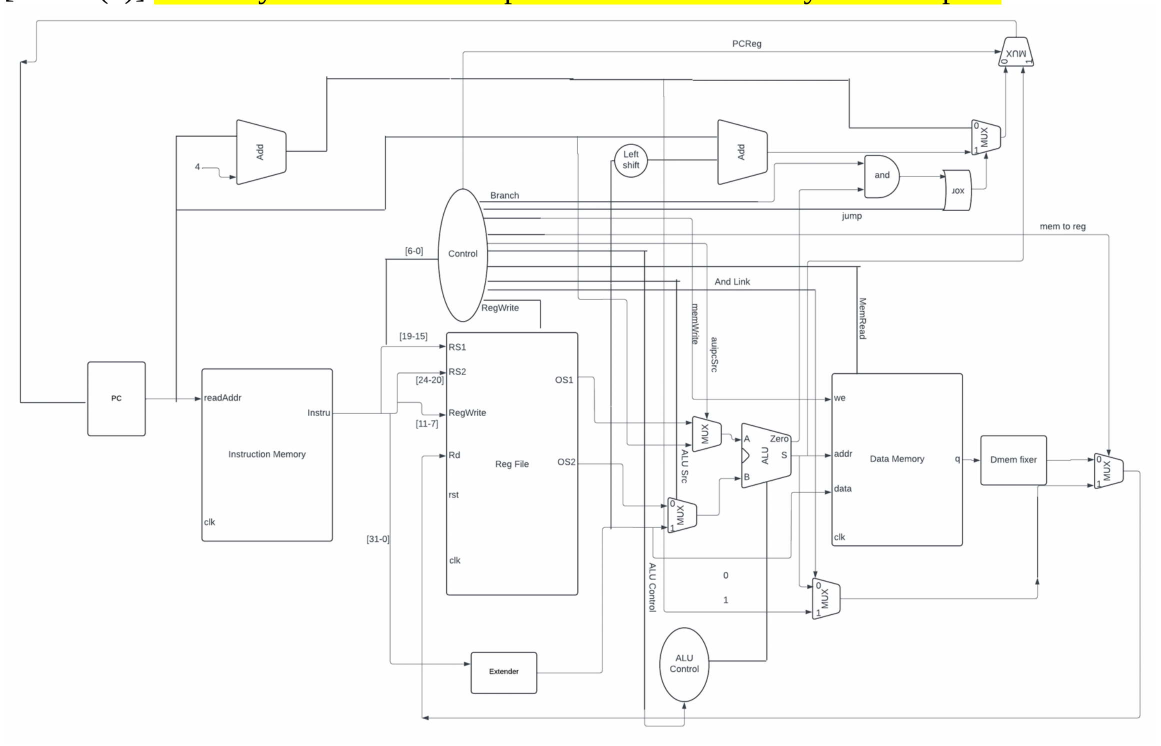
Team Members: __Shane Nebraska_________

___Zevan Gustafon________

Project Teams Group #:_____D_4____________________

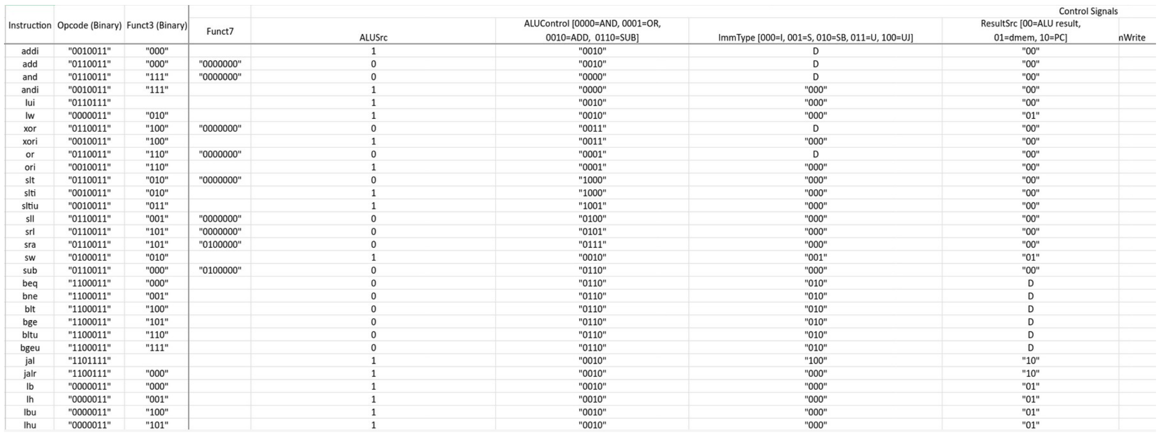
***Refer to the highlighted language in the project 1 instruction for the context of the following questions*.**

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

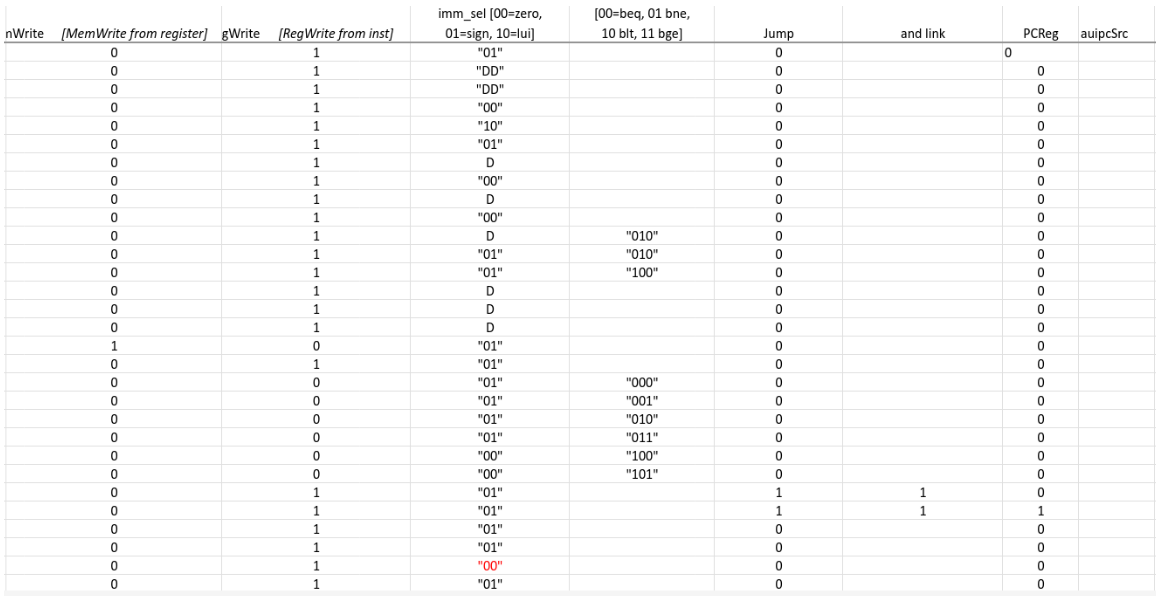


[Part 3.1.a.] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your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

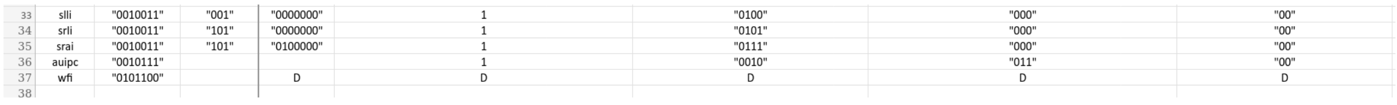
*N*M* table where each row corresponds to the output of the control logic module for a given instruction.



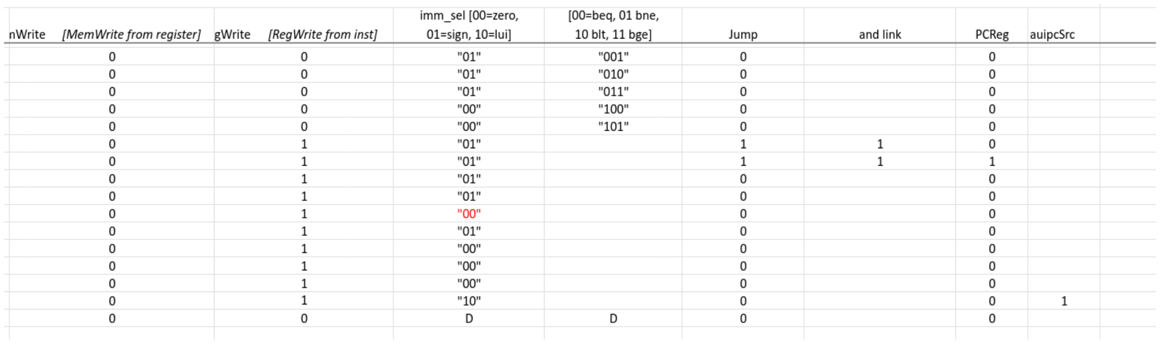
							Control Signals	
Instruction	Opcode (Binary)	Funct3 (Binary)	Funct7	ALUSrc	ALUControl [0000=AND, 0001=OR, 0010=ADD, 0110=SUB]	ImmType [000=I, 001=S, 010=SB, 011=U, 100=UJ]	ResultSrc [00=ALU result, 01=dmem, 10=PC]	nWrite
addi	"0010011"	"000"		1	"0010"	D	"00"	
add	"0110011"	"000"	"0000000"	0	"0010"	D	"00"	
and	"0110011"	"111"	"0000000"	0	"0000"	D	"00"	
andi	"0010011"	"111"		1	"0000"	"000"	"00"	
lui	"0110111"			1	"0010"	"000"	"00"	
lw	"0000011"	"010"		1	"0010"	"000"	"01"	
xor	"0110011"	"100"	"0000000"	0	"0011"	D	"00"	
xori	"0010011"	"100"		1	"0011"	"000"	"00"	
or	"0110011"	"110"	"0000000"	0	"0001"	D	"00"	
ori	"0010011"	"110"		1	"0001"	"000"	"00"	
slt	"0110011"	"010"	"0000000"	0	"1000"	"000"	"00"	
slti	"0010011"	"010"		1	"1000"	"000"	"00"	
sltiu	"0010011"	"011"		1	"1001"	"000"	"00"	
sll	"0110011"	"001"	"0000000"	0	"0100"	"000"	"00"	
srl	"0110011"	"101"	"0000000"	0	"0101"	"000"	"00"	
sra	"0110011"	"101"	"0100000"	0	"0111"	"000"	"00"	
sw	"0100011"	"010"		1	"0010"	"001"	"01"	
sub	"0110011"	"000"	"0100000"	0	"0110"	"000"	"00"	
beq	"1100011"	"000"		0	"0110"	"010"	D	
bne	"1100011"	"001"		0	"0110"	"010"	D	
blt	"1100011"	"100"		0	"0110"	"010"	D	
bge	"1100011"	"101"		0	"0110"	"010"	D	
bltu	"1100011"	"110"		0	"0110"	"010"	D	
bgeu	"1100011"	"111"		0	"0110"	"010"	D	
jal	"1101111"			1	"0010"	"100"	"10"	
jalr	"1100111"	"000"		1	"0010"	"000"	"10"	
lb	"0000011"	"000"		1	"0010"	"000"	"01"	
lh	"0000011"	"001"		1	"0010"	"000"	"01"	
lbu	"0000011"	"100"		1	"0010"	"000"	"01"	
lhu	"0000011"	"101"		1	"0010"	"000"	"01"	



nWrite *[MemWrite from register]*	gWrite *[RegWrite from inst]*	imm_sel [00=zero, 01=sign, 10=lui]	[00=beq, 01 bne, 10 blt, 11 bge]	Jump	and link	PCReg	auipcSrc
0	1	"01"		0		0	
0	1	"DD"		0		0	
0	1	"DD"		0		0	
0	1	"00"		0		0	
0	1	"10"		0		0	
0	1	"01"		0		0	
0	1	D		0		0	
0	1	"00"		0		0	
0	1	D		0		0	
0	1	"00"		0		0	
0	1	D	"010"	0		0	
0	1	"01"	"010"	0		0	
0	1	"01"	"100"	0		0	
0	1	D		0		0	
0	1	D		0		0	
0	1	D		0		0	
1	0	"01"		0		0	
0	1	"01"		0		0	
0	0	"01"	"000"	0		0	
0	0	"01"	"001"	0		0	
0	0	"01"	"010"	0		0	
0	0	"01"	"011"	0		0	
0	0	"00"	"100"	0		0	
0	0	"00"	"101"	0		0	
0	1	"01"		1	1	0	
0	1	"01"		1	1	1	
0	1	"01"		0		0	
0	1	"01"		0		0	
0	1	"00"		0		0	
0	1	"01"		0		0	

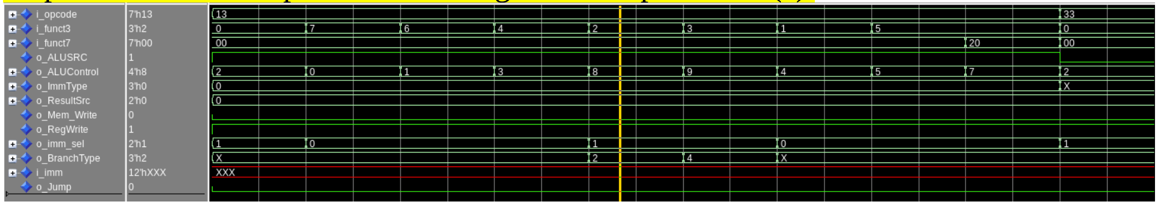


33	slli	"0010011"	"001"	"0000000"	1	"0100"	"000"	"00"
34	srli	"0010011"	"101"	"0000000"	1	"0101"	"000"	"00"
35	srai	"0010011"	"101"	"0100000"	1	"0111"	"000"	"00"
36	auipc	"0010111"			1	"0010"	"011"	"00"
37	wfi	"0101100"		D	D	D	D	D
38								



nWrite *[MemWrite from register]*	gWrite *[RegWrite from inst]*	imm_sel [00=zero, 01=sign, 10=lui]	[00=beq, 01 bne, 10 blt, 11 bge]	Jump	and link	PCReg	auipcSrc
0	0	"01"	"001"	0		0	
0	0	"01"	"010"	0		0	
0	0	"01"	"011"	0		0	
0	0	"00"	"100"	0		0	
0	0	"00"	"101"	0		0	
0	1	"01"		1	1	0	
0	1	"01"		1	1	1	
0	1	"01"		0		0	
0	1	"01"		0		0	
0	1	"00"		0		0	
0	1	"01"		0		0	
0	1	"00"		0		0	
0	1	"00"		0		0	
0	1	"00"		0		0	
0	1	"10"		0		0	1
0	0	D	D	0		0	

[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).

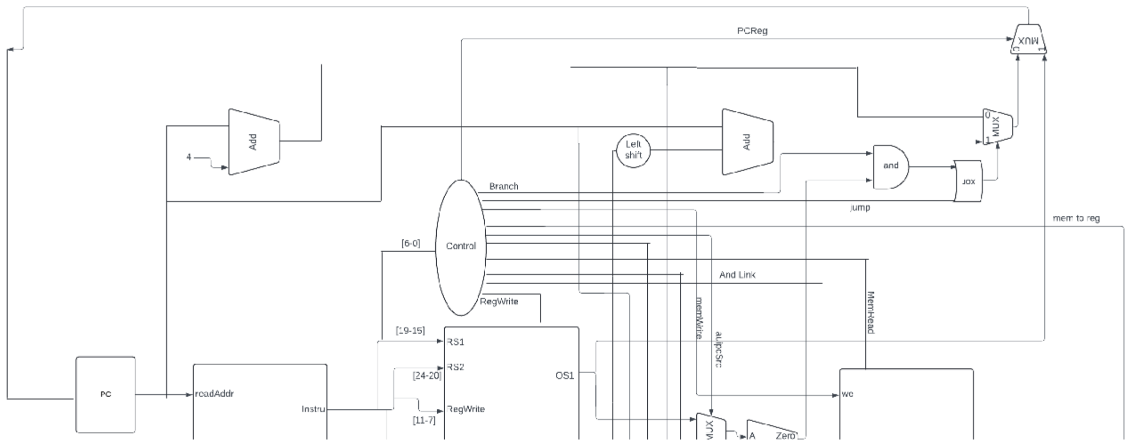


```
i_opcode       7'h13
i_funct3       3'h2
i_funct7       7'h00
o_ALUSRC       1
o_ALUControl   4'h8
o_ImmType      3'h0
o_ResultSrc    2'h0
o_Mem_Write    0
o_RegWrite     1
o_imm_sel      2'h1
o_BranchType   3'h2
i_imm          12'hXXX
o_Jump         0
```

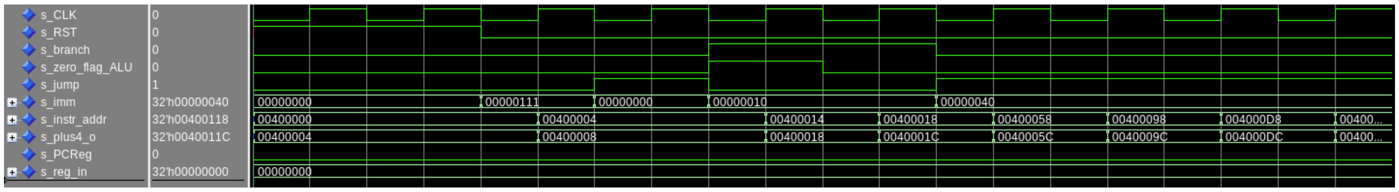
The testbench goes through most of the instructions we are required to implement. The control signals match what the instruction does. For example sw writes to memory so o_Mem_Write is a 1. All the outputs match to what we included in the control signal excel sheet.

[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.
The possible control flow possibilities for the fetch include branching, zero flag for comparisons, jump, and PCReg. Branching checks if the instruction requires branching. Jump is used to allow jump related commands to jump to the given address. Finally PCReg sends the value from what is stored in the register 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 new flags that were included were branching, zero flag, jump, and PCReg.
[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.



```
s_CLK           0
s_RST           0
s_branch        0
s_zero_flag_ALU 0
s_jump          1
s_imm           32'h00000040   00000000  00000111  00000000  00000010  00000040
s_instr_addr    32'h00400118   00400000  00400004  00400014  00400018  00400058  00400098  004000D8  00400...
s_plus4_o       32'h0040011C   00400004  00400008  00400018  0040001C  0040005C  0040009C  004000DC  00400...
s_PCReg         0
s_reg_in        32'h00000000   00000000
```

The testbench includes instructions that require these new control signal and fetch. The testbench includes add and addi to ensure that the pc gets increased by four after executing the instructions. Finally there are some jump and brach instructions to test the other parts of fetch. All behavior matches what is expected in the excel sheet and from how the instructions work.

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

Logical shifts include zero in the new bits created from the shift. Arithmetic creates new bits with Fs or zeros depending on the sign instead. Left shifts do not need the new bits to be anything other than zero since left shifts are the same regardless of sign.

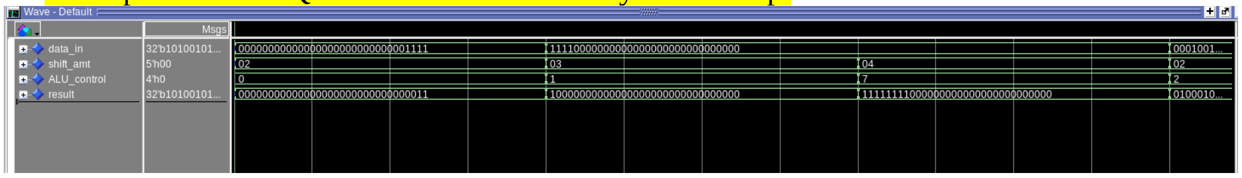
[Part 3.3.1.(b)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

There is a control signal that goes into the shifter the states what type of shift it will be doing. The control uses the instruction's opcode to decide this. Then with the control signal the shifter will know what the new bits should be and if it should be based off if the input is signed or not.

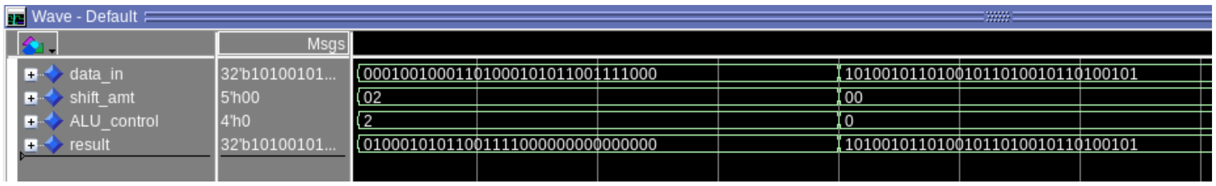
[Part 3.3.1.(c)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

To do this a new internal value will need to be made to decide the direction of the shift which could be decided from the same control signal used above. Once it is found that a left shift needs to be done then the input data is reversed and a right shift is done.

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



```
Wave - Default
                Msgs
data_in     32'b10100101...   00000000000000000000000000001111  11110000000000000000000000000000                                    0001001...
shift_amt   5'h00             02                                03                                04                                02
ALU_control 4'h0              0                                 1                                 7                                 2
result      32'b10100101...   00000000000000000000000000000011  10000000000000000000000000000000  11111111000000000000000000000000  0100010...
```



```
Wave - Default
                Msgs
data_in     32'b10100101...   00010010001101000101011001111000  10100101101001011010010110100101
shift_amt   5'h00             02                                00
ALU_control 4'h0              2                                 0
result      32'b10100101...   01000101011001111000000000000000  10100101101001011010010110100101
```

The testbench verifies that each type of shift works. In addition, it checks if it shifts the proper amount. It also verifies that the proper alu_control works as intended.

[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.

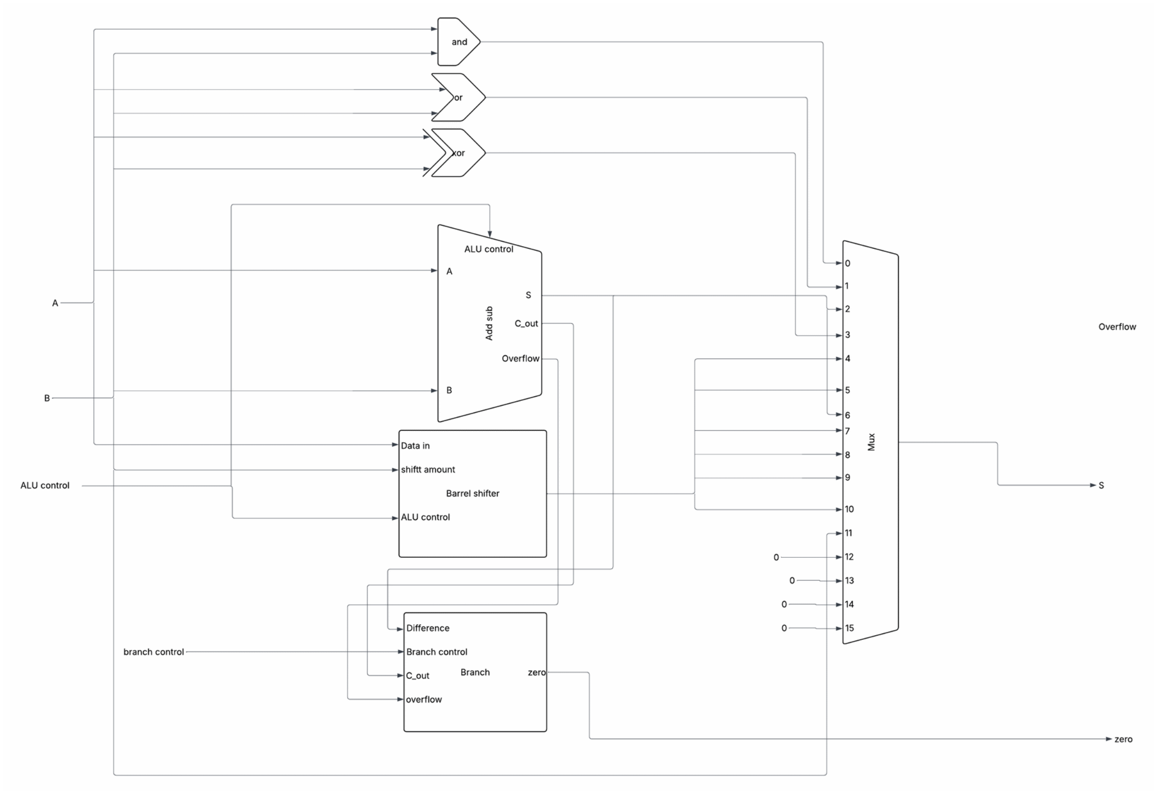
For the design of the processor we took inspiration from the fetch processor example from lecture. The complete design of the of the whole processor did not require any new components that the lab manual did not have us create already.

[Part 3.3.2.(b)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

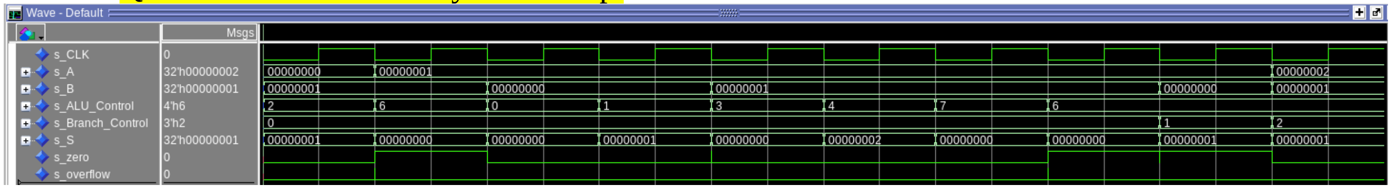
Not available since we did not create any new components not listed in the lab manual.

[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 calculated by subtracting the two inputs if the ALU. This allows for comparisons to be done since for example if they are equal the sum should be zero. Slt was implemented by doing the same comparison and depending on the result setting the C-out to be one if that statement is true.



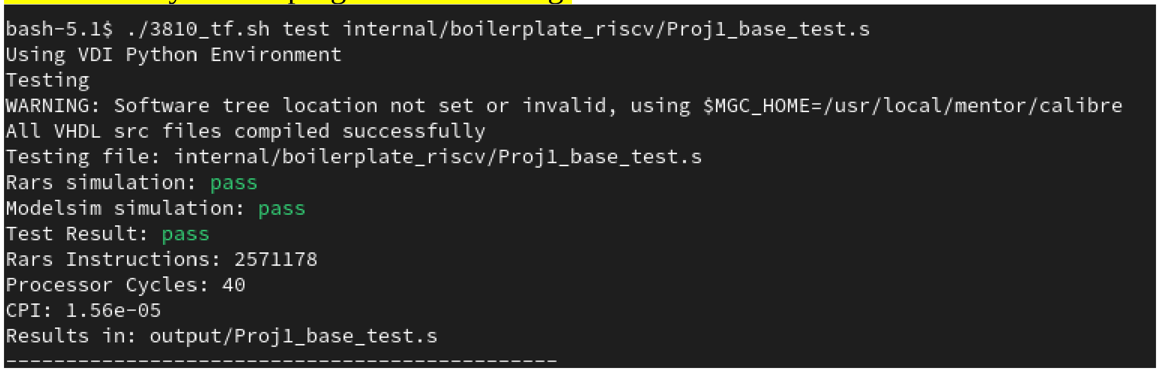
[Part 3.3.5] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.



```
s_CLK             0
s_A               32'h00000002  00000000  00000001                                                                                                      00000002
s_B               32'h00000001  00000001            00000000            00000001                                                      00000000  00000001
s_ALU_Control     4'h6          2         6         0         1         3         4         7         6
s_Branch_Control  3'h2          0                                                                                                     1         2
s_S               32'h00000001  00000001  00000000  00000000  00000001  00000000  00000002  00000000  00000000  00000001  00000001
s_zero            0
s_overflow        0
```

The testbench verifies each unique type of instruction. The outputs match the expected outcome for the instructions used. The control signals are also verified to match what was present in control signals excel sheet.

[Part 3.3.8] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

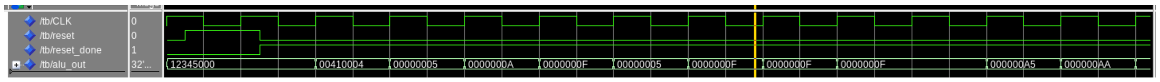


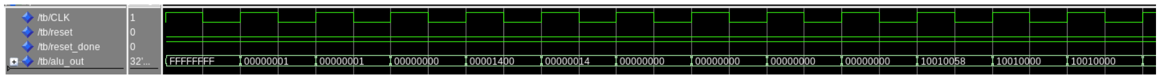
```
bash-5.1$ ./3810_tf.sh test internal/boilerplate_riscv/Proj1_base_test.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using $MGC_HOME=/usr/local/mentor/calibre
All VHDL src files compiled successfully
Testing file: internal/boilerplate_riscv/Proj1_base_test.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 2571178
Processor Cycles: 40
CPI: 1.56e-05
Results in: output/Proj1_base_test.s
----------------------------------------------
```

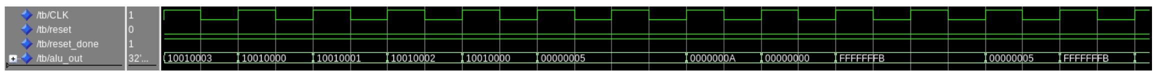
We were able to get all the tests to pass including the one that use all of the required instructions. This test proves the integrated ALU works.

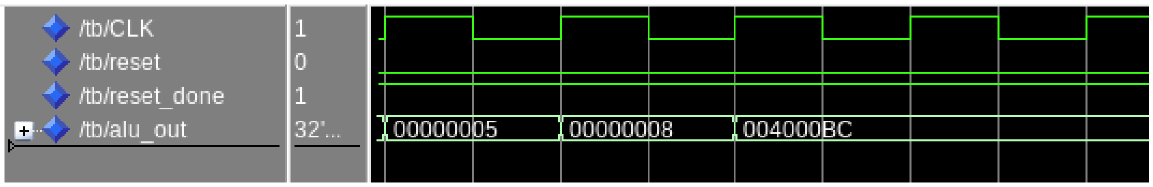
[Part 4] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[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.



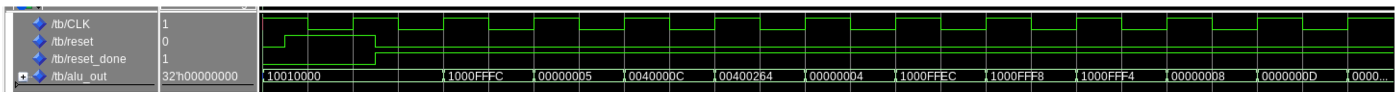


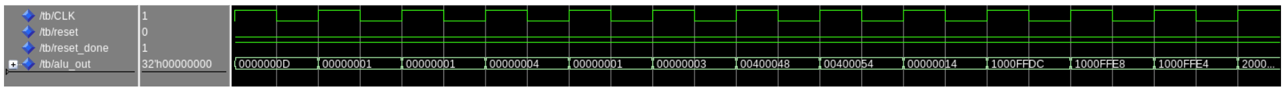


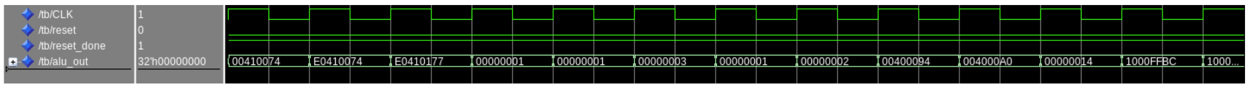


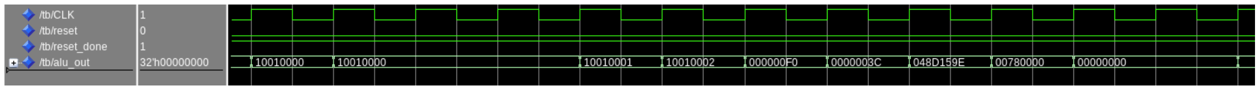
These waveform screenshots show correct behavior in addition the processor passes the test.

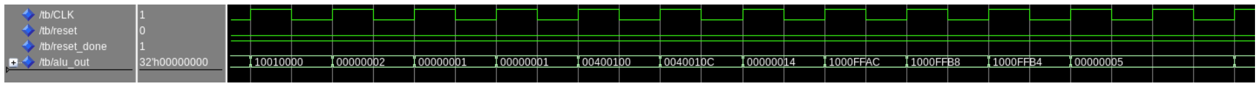
[Part 4.b] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

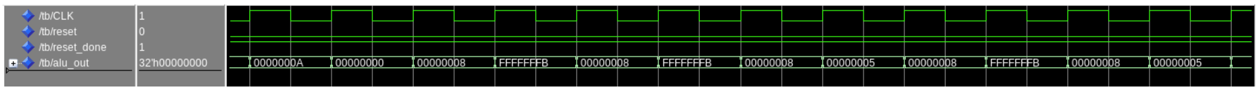

/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
10010000 1000FFFC 00000005 0040000C 00400264 00000004 1000FFEC 1000FFF8 1000FFF4 00000008 0000000D 0000...

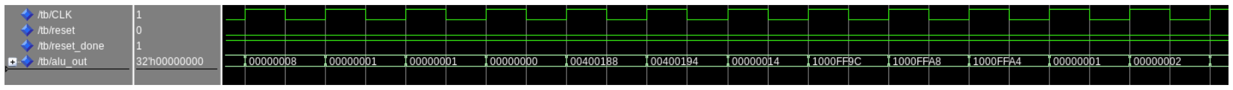

/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
0000000D 00000001 00000001 00000004 00000001 00000003 00400048 00400054 00000014 1000FFDC 1000FFE8 1000FFE4 2000...

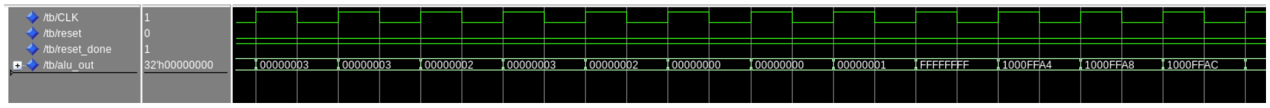

/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
00410074 E0410074 E0410177 00000001 00000001 00000003 00000001 00000002 00400094 004000A0 00000014 1000FFBC 1000...

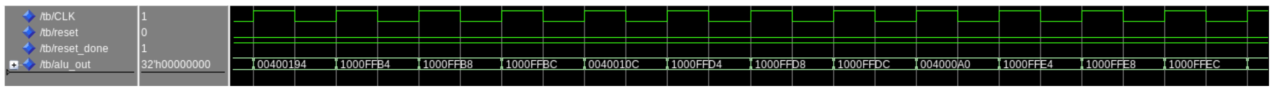

/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
10010000 10010000 10010001 10010002 000000F0 0000003C 048D159E 00780000 00000000

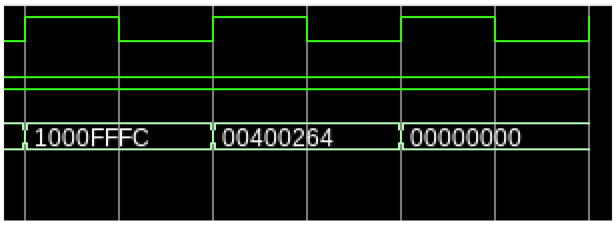

/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
10010000 00000002 00000001 00000001 00400100 0040010C 00000014 1000FFAC 1000FFB8 1000FFB4 00000005


/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
0000000A 00000000 00000008 FFFFFFFB 00000008 FFFFFFFB 00000008 00000005 00000008 FFFFFFFB 00000008 00000005


/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
00000008 00000001 00000001 00000000 00400188 00400194 00000014 1000FF9C 1000FFA8 1000FFA4 00000001 00000002

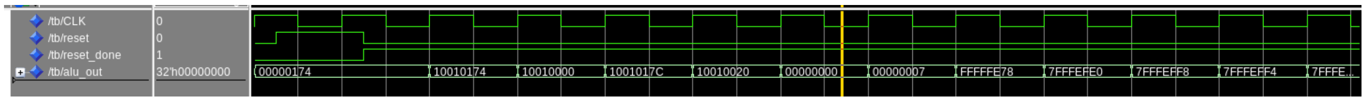

/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
00000003 00000003 00000002 00000003 00000002 00000000 00000000 00000001 FFFFFFFF 1000FFA4 1000FFA8 1000FFAC


/tb/CLK 1
/tb/reset 0
/tb/reset_done 1
/tb/alu_out 32'h00000000
00400194 1000FFB4 1000FFB8 1000FFBC 0040010C 1000FFD4 1000FFD8 1000FFDC 004000A0 1000FFE4 1000FFE8 1000FFEC

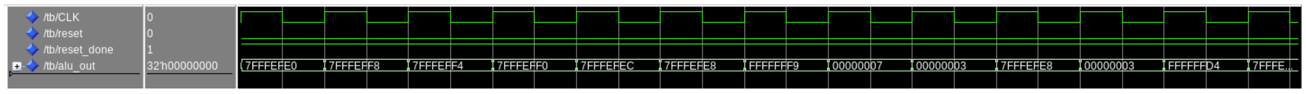


These waveform screenshots show correct behavior in addition the processor passes the test.

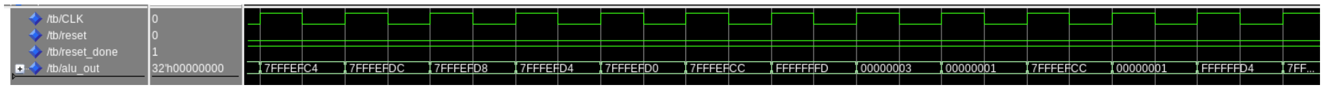
[Part 4.c] Create and test an application that sorts an array with *N* elements using the MergeSort algorithm (link). Name this file Proj1_mergesort.s.



```
/tb/CLK         0
/tb/reset       0
/tb/reset_done  1
/tb/alu_out     32'h00000000   00000174 10010174 10010000 1001017C 10010020 00000000 00000007 FFFFFE78 7FFFEFE0 7FFFEFF8 7FFFEFF4 7FFFE...
```



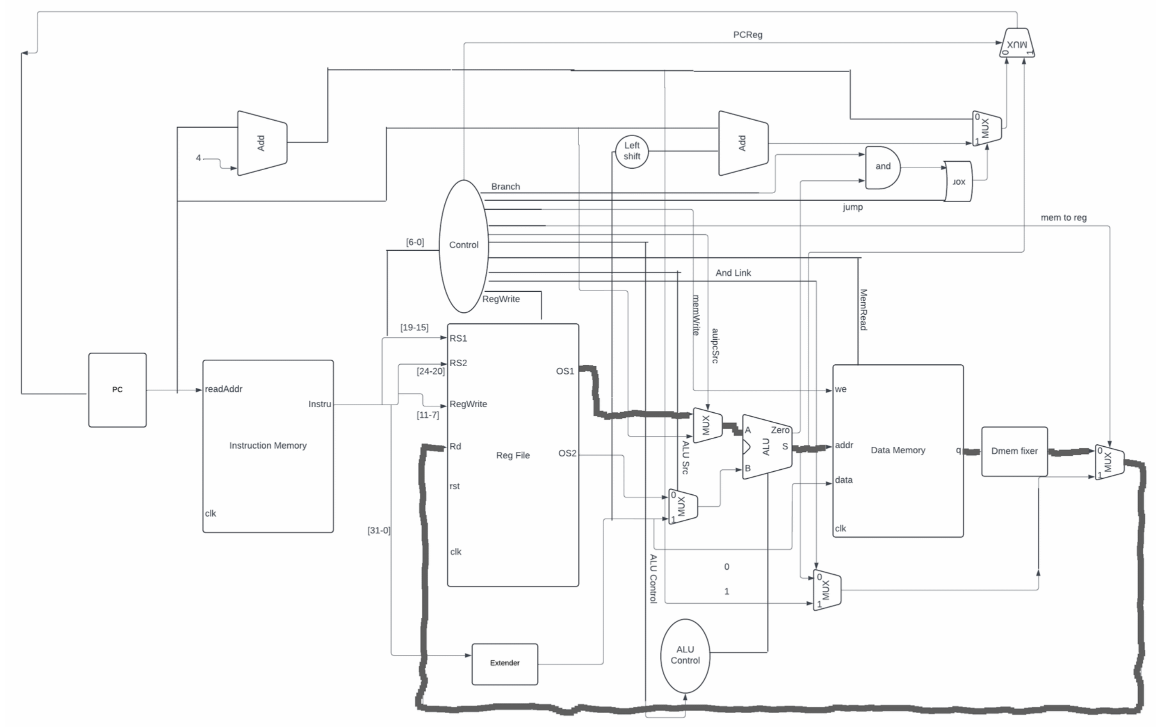
```
/tb/CLK         0
/tb/reset       0
/tb/reset_done  1
/tb/alu_out     32'h00000000   7FFFEFE0 7FFFEFF8 7FFFEFF4 7FFFEFF0 7FFFEFEC 7FFFEFE8 FFFFFFF9 00000007 00000003 7FFFEFE8 00000003 FFFFFFD4 7FFFE...
```



```
/tb/CLK         0
/tb/reset       0
/tb/reset_done  1
/tb/alu_out     32'h00000000   7FFFEFC4 7FFFEFDC 7FFFEFD8 7FFFEFD4 7FFFEFD0 7FFFEFCC FFFFFFFD 00000003 00000001 7FFFEFCC 00000001 FFFFFFD4 7FF...
```

These waveform screenshots show correct behavior in addition the processor passes the test.

[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?
The maximum frequency that the processor can run at is 21.65 mhz. The critical path of the processor is illustrated below. To improve the frequency we can improve the latency of any big component on the critical path. For example if the dmem has a lot of latency we can split it into two components to lower the latency.



PCReg
MUX
0
1
Add
4
Left shift
Add
and
xor
Branch
jump
mem to reg
Control
[6-0]
And Link
MemRead
memWrite
auipcSrc
RegWrite
[19-15]
RS1
RS2
[24-20]
RegWrite
[11-7]
Rd
rst
clk
Reg File
OS1
OS2
PC
readAddr
Instru
Instruction Memory
clk
[31-0]
MUX
ALU Src
A
B
Zero
S
ALU
we
addr
data
clk
Data Memory
q
Dmem fixer
ALU Control
Extender