‘CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: Jack Sebahar

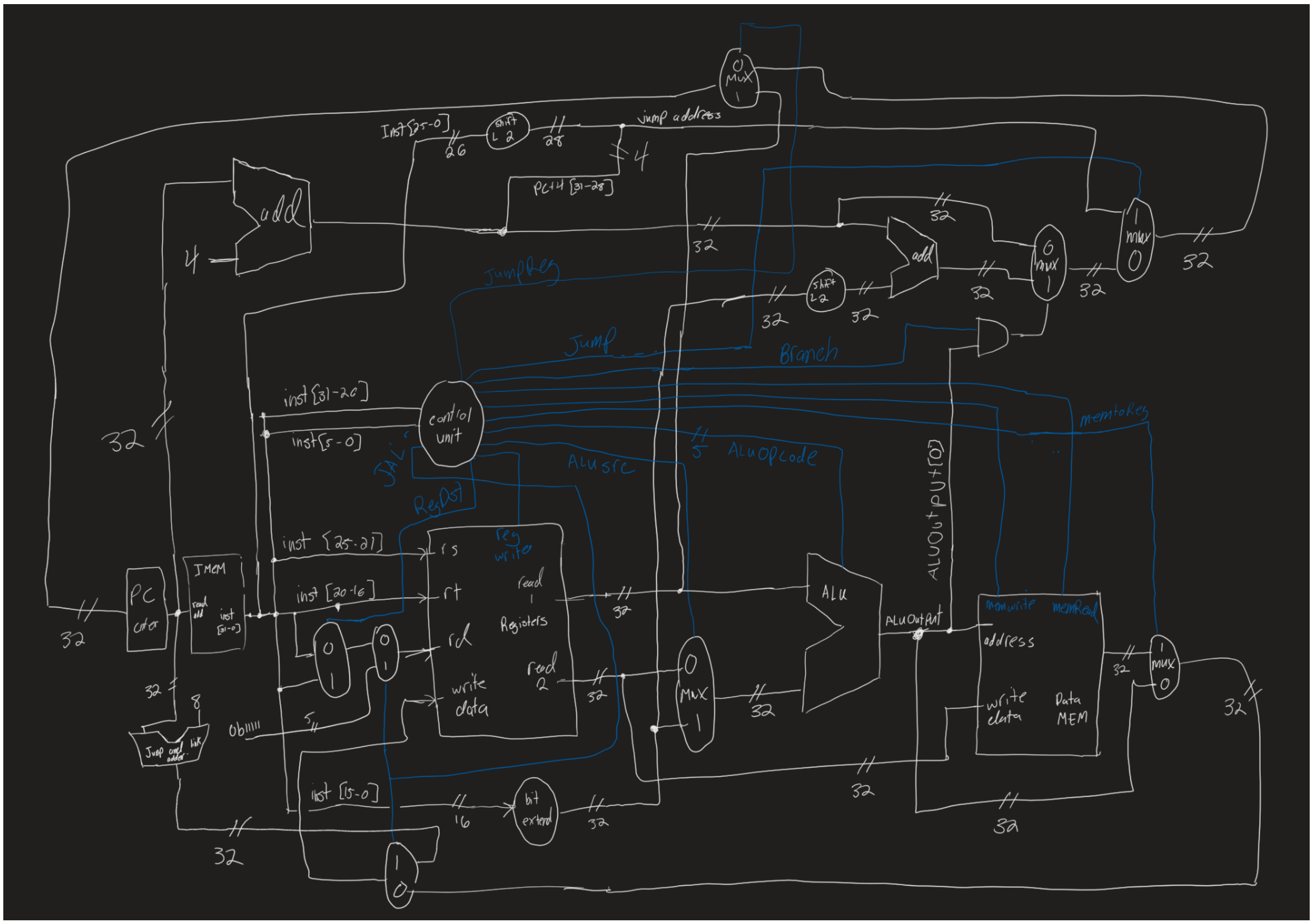
Peter Wissman

John Lavigne

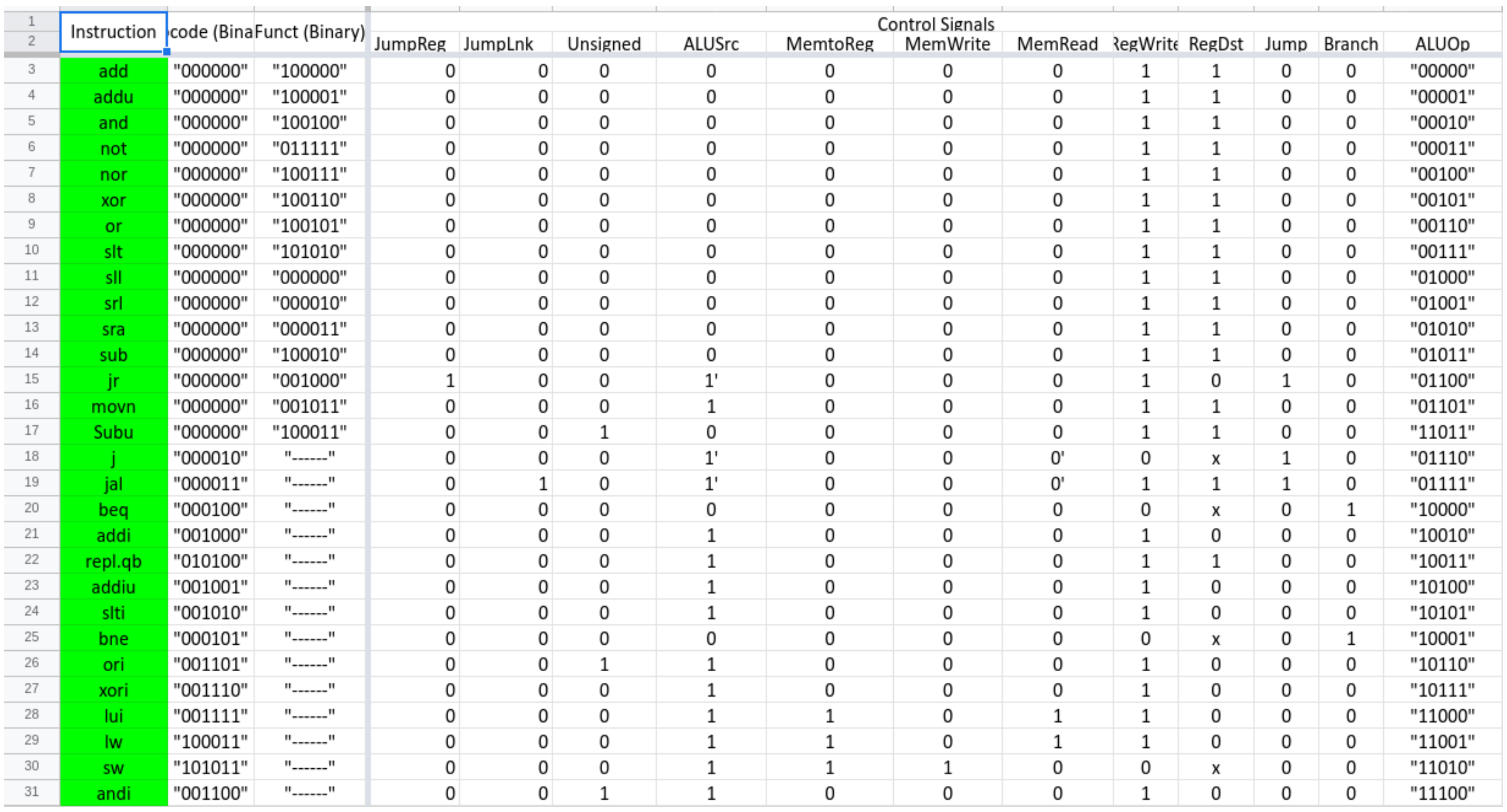
Project Teams Group #: 1

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.
all the blue lines are control input lines

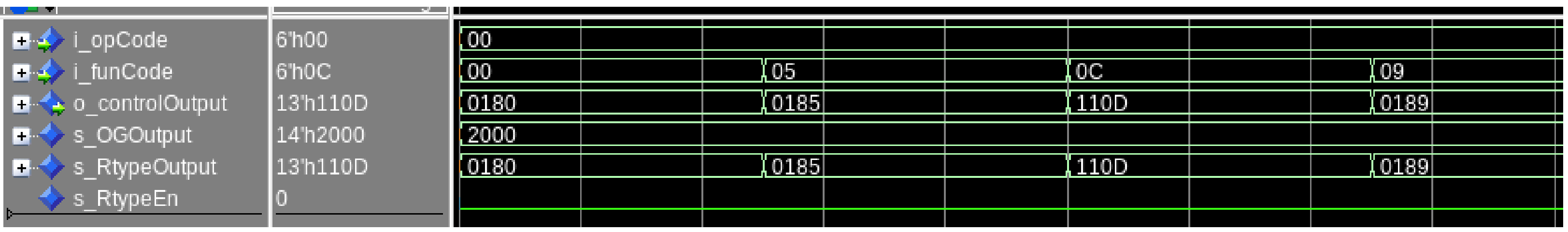


[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your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

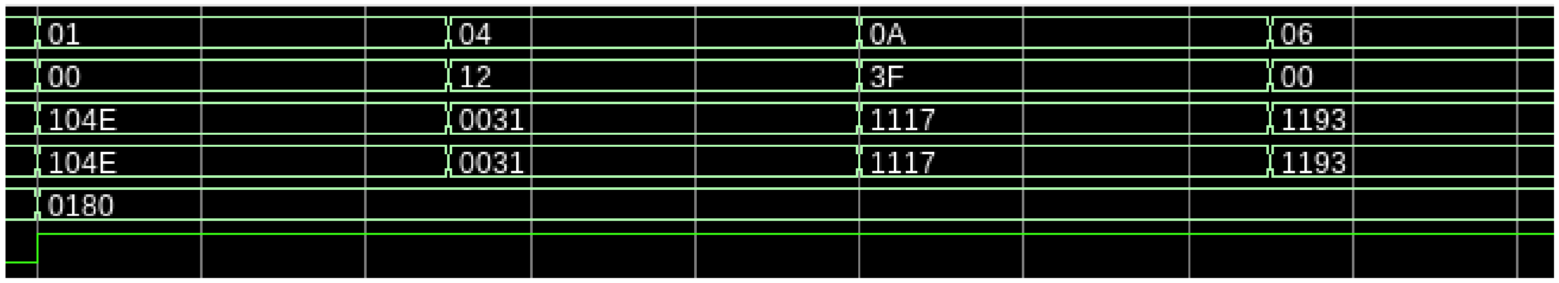


1	Instruction	code (Bina	Funct (Binary)	Control Signals											
2				JumpReg	JumpLnk	Unsigned	ALUSrc	MemtoReg	MemWrite	MemRead	RegWrite	RegDst	Jump	Branch	ALUOp
3	add	"000000"	"100000"	0	0	0	0	0	0	0	1	1	0	0	"00000"
4	addu	"000000"	"100001"	0	0	0	0	0	0	0	1	1	0	0	"00001"
5	and	"000000"	"100100"	0	0	0	0	0	0	0	1	1	0	0	"00010"
6	not	"000000"	"011111"	0	0	0	0	0	0	0	1	1	0	0	"00011"
7	nor	"000000"	"100111"	0	0	0	0	0	0	0	1	1	0	0	"00100"
8	xor	"000000"	"100110"	0	0	0	0	0	0	0	1	1	0	0	"00101"
9	or	"000000"	"100101"	0	0	0	0	0	0	0	1	1	0	0	"00110"
10	slt	"000000"	"101010"	0	0	0	0	0	0	0	1	1	0	0	"00111"
11	sll	"000000"	"000000"	0	0	0	0	0	0	0	1	1	0	0	"01000"
12	srl	"000000"	"000010"	0	0	0	0	0	0	0	1	1	0	0	"01001"
13	sra	"000000"	"000011"	0	0	0	0	0	0	0	1	1	0	0	"01010"
14	sub	"000000"	"100010"	0	0	0	0	0	0	0	1	1	0	0	"01011"
15	jr	"000000"	"001000"	1	0	0	1'	0	0	0	1	0	1	0	"01100"
16	movn	"000000"	"001011"	0	0	0	1	0	0	0	1	1	0	0	"01101"
17	Subu	"000000"	"100011"	0	0	1	0	0	0	0	1	1	0	0	"11011"
18	j	"000010"	"------"	0	0	0	1'	0	0	0'	0	x	1	0	"01110"
19	jal	"000011"	"------"	0	1	0	1'	0	0	0'	1	1	1	0	"01111"
20	beq	"000100"	"------"	0	0	0	0	0	0	0	0	x	0	1	"10000"
21	addi	"001000"	"------"	0	0	0	1	0	0	0	1	0	0	0	"10010"
22	repl.qb	"010100"	"------"	0	0	0	1	0	0	0	1	1	0	0	"10011"
23	addiu	"001001"	"------"	0	0	0	1	0	0	0	1	0	0	0	"10100"
24	slti	"001010"	"------"	0	0	0	1	0	0	0	1	0	0	0	"10101"
25	bne	"000101"	"------"	0	0	0	0	0	0	0	0	x	0	1	"10001"
26	ori	"001101"	"------"	0	0	1	1	0	0	0	1	0	0	0	"10110"
27	xori	"001110"	"------"	0	0	0	1	0	0	0	1	0	0	0	"10111"
28	lui	"001111"	"------"	0	0	0	1	1	0	1	1	0	0	0	"11000"
29	lw	"100011"	"------"	0	0	0	1	1	0	1	1	0	0	0	"11001"
30	sw	"101011"	"------"	0	0	0	1	1	1	0	0	x	0	0	"11010"
31	andi	"001100"	"------"	0	0	1	1	0	0	0	1	0	0	0	"11100"

[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).



```
i_opCode          6'h00     00
i_funCode         6'h0C     00          05          0C          09
o_controlOutput   13'h110D  0180        0185        110D        0189
s_OGOutput        14'h2000  2000
s_RtypeOutput     13'h110D  0180        0185        110D        0189
s_RtypeEn         0
```



```
01          04          0A          06
00          12          3F          00
104E        0031        1117        1193
104E        0031        1117        1193
0180
```

Testing these opcodes comes out with the right values, as shown later, testing with the full processor shows the rest of the values not tested here work correctly

[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.

BNE: must take a branch not equal input from the control unit and and gate it with NOT ZERO in order to follow with the normal incremented address or change to the branch address

BEQ: must take a branch is equal input from the control unit and and gate with ZERO to choose whether to branch to branch address or stay on normal incremented address

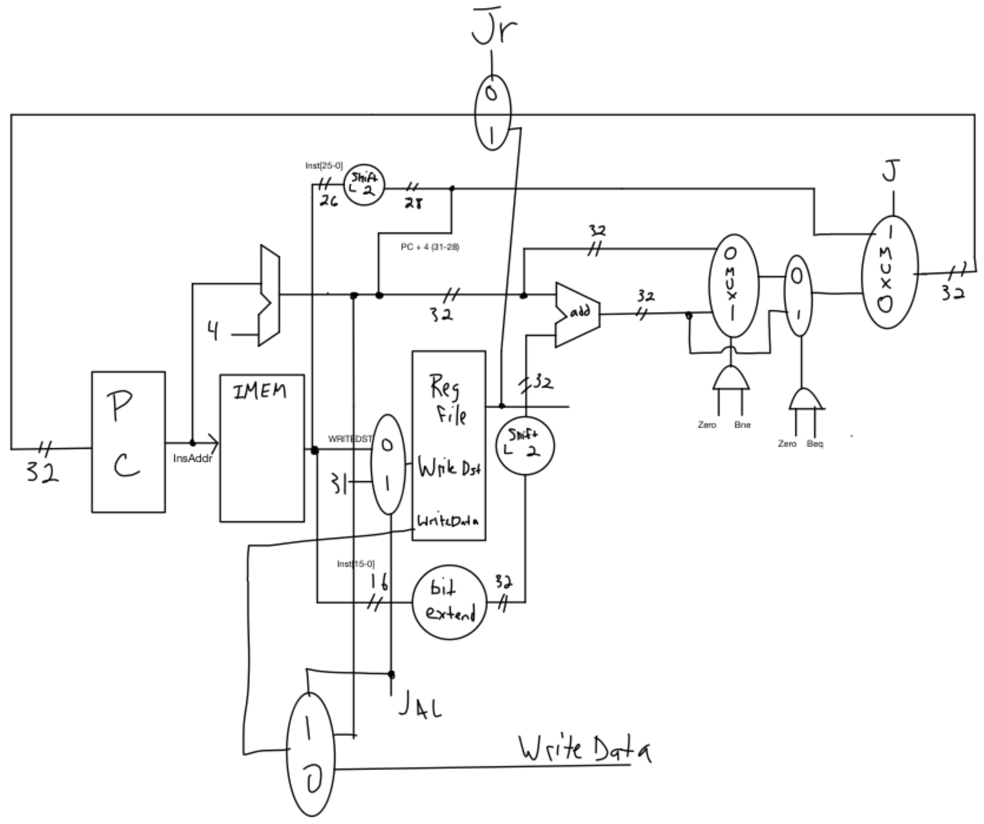
Jal: Jump and link must set register 31 to the PC address incremented by 4. This can be done with a mux on the write address of the register file and porting the pc address +4 into a mux to choose 1 output from mem or pc address when Jal = 1

J: Jump runs to a mux to choose between the jump address and or keep the normal address

Jr: Jump register jumps to the saved register address in RS (31) or the normal address + 4

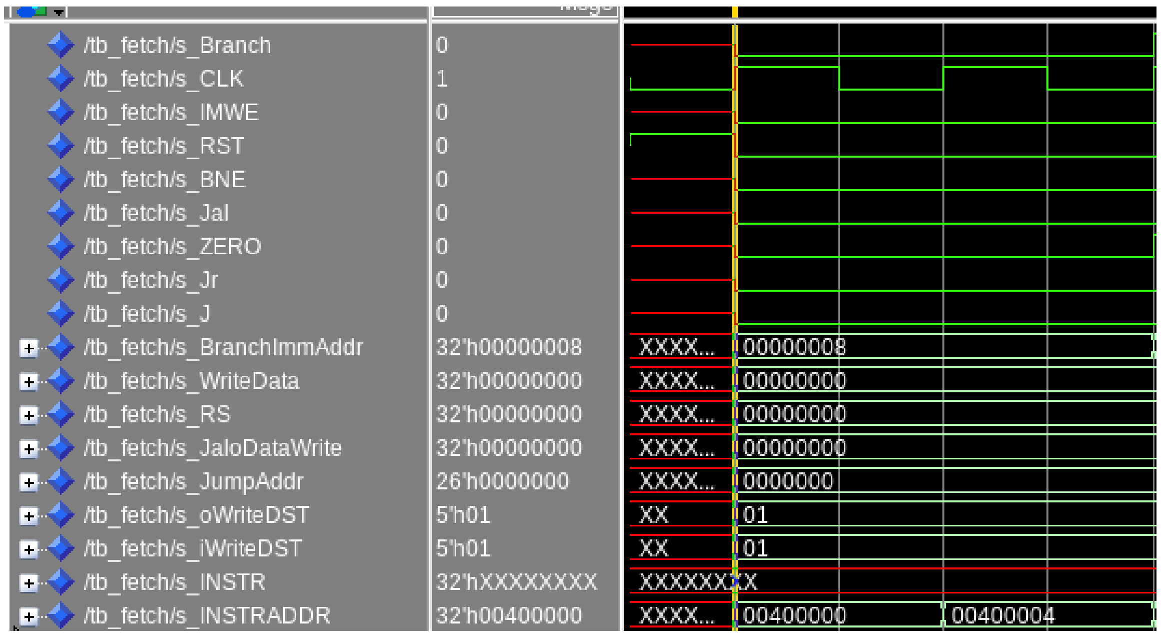
[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

We needed J, JAL, JR, BNE, and BEQ implementation.



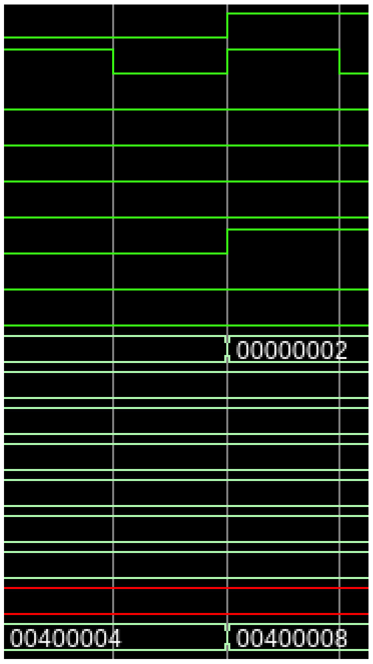
[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use Modelsim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the Modelsim waveforms in your writeup.

Normal Increment: address of 0 + 4 is expected or address 1



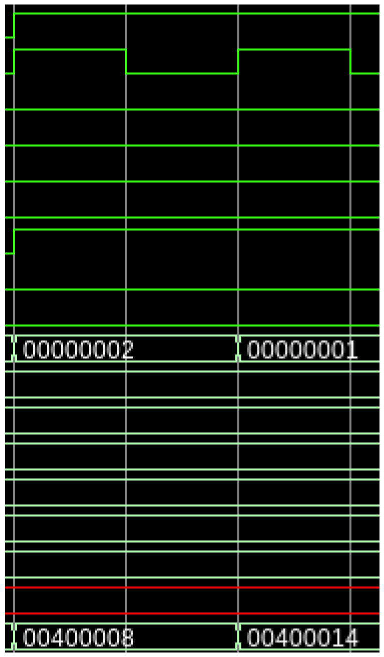
```
/tb_fetch/s_Branch          0              
/tb_fetch/s_CLK             1              
/tb_fetch/s_IMWE            0              
/tb_fetch/s_RST             0              
/tb_fetch/s_BNE             0              
/tb_fetch/s_Jal             0              
/tb_fetch/s_ZERO            0              
/tb_fetch/s_Jr              0              
/tb_fetch/s_J               0              
/tb_fetch/s_BranchImmAddr   32'h00000008   XXXX...  00000008
/tb_fetch/s_WriteData       32'h00000000   XXXX...  00000000
/tb_fetch/s_RS              32'h00000000   XXXX...  00000000
/tb_fetch/s_JaloDataWrite   32'h00000000   XXXX...  00000000
/tb_fetch/s_JumpAddr        26'h0000000    XXXX...  0000000
/tb_fetch/s_oWriteDST       5'h01          XX       01
/tb_fetch/s_iWriteDST       5'h01          XX       01
/tb_fetch/s_INSTR           32'hXXXXXXXX   XXXXXXXX
/tb_fetch/s_INSTRADDR       32'h00400000   XXXX...  00400000         00400004
```

Normal Increment address of last +4 is expected or address 2

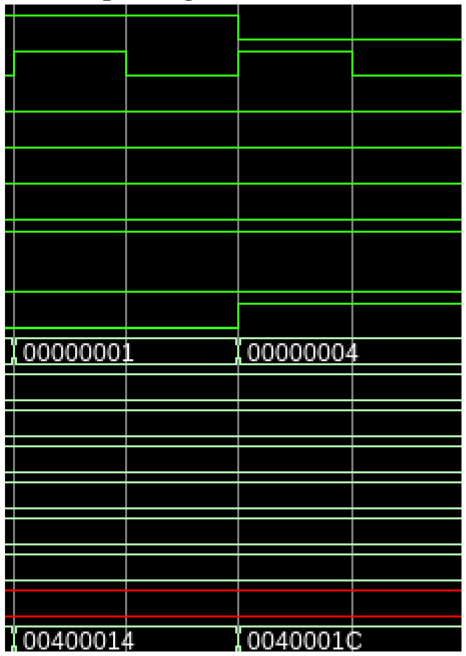


```
00000002
00400004    00400008
```

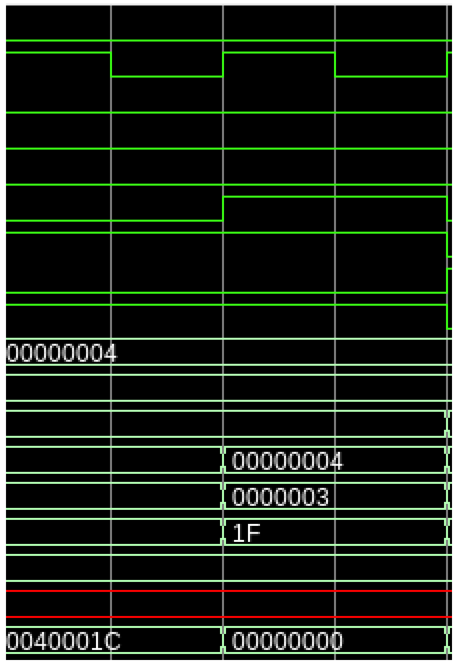
BEQ expecting instruction address 5 or hx08 + hx0C = hx14



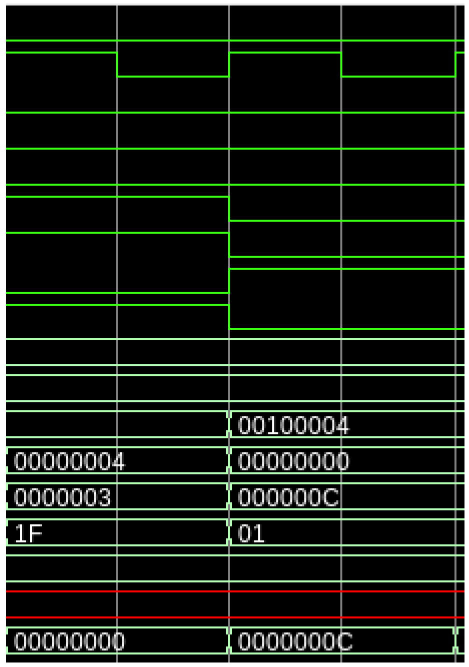
BNE expecting address 7 or 1C



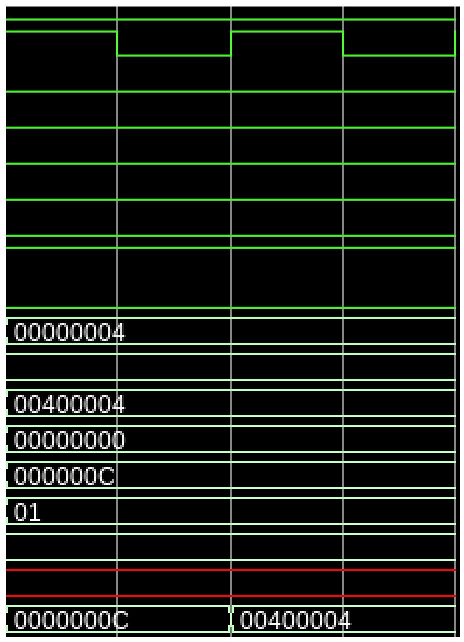
J expecting address output 0000



Jal expecting address 3 with writeDST 31



Jr expecting address 1 again



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

A right shift logical simply shifts each bit one position to the right, with the least significant bit being discarded and the most significant bit filled with 0. A left shift logical shifts each bit one to the left, with the least significant bit filled with 0 and the most significant bit being discarded. A right arithmetic shift still shifts each bit one to the right, with the least significant bit being discarded but the most significant bit stays the same, preserving the sign of the number. MIPS does not have an sla option because arithmetic and non-arithmetic operations can already be done with sll, so it would be the same thing.

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

For logical shifts, it just pads the shifted bits with 0s, not caring about what the sign bit actually is. For arithmetic operations, it checks whether the sign bit is 1 or 0 (bit 31), and saves it for when it needs to pad the shifted bits at the end.

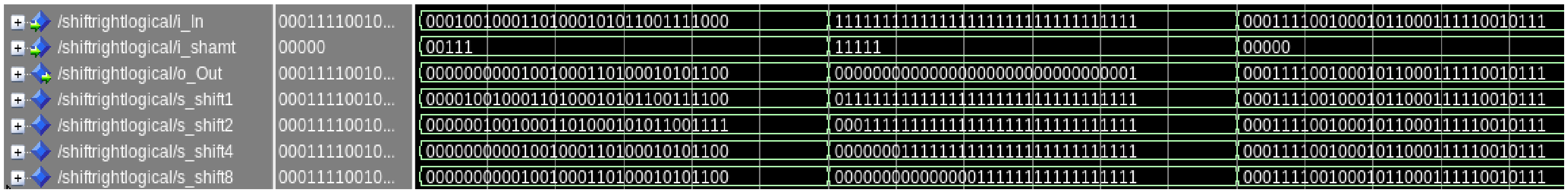
[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

If you reverse the input, shift right, and then reverse the output, it's effectively the same thing as shifting left by the same amount. So, instead of coming up with a completely new shifter design, you can just reverse the input and output when a left shift is selected.

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

Instead of using one barrel shifter file for all of the shifting instructions, we decided to split it up into each shifting operation having a different file. This made more sense to us when implementing the ALU op code and differentiating which shift is done. It also simplifies some of the control signals. The three files are very similar but with slight differences depending on if it was for a left shift or arithmetic shift.

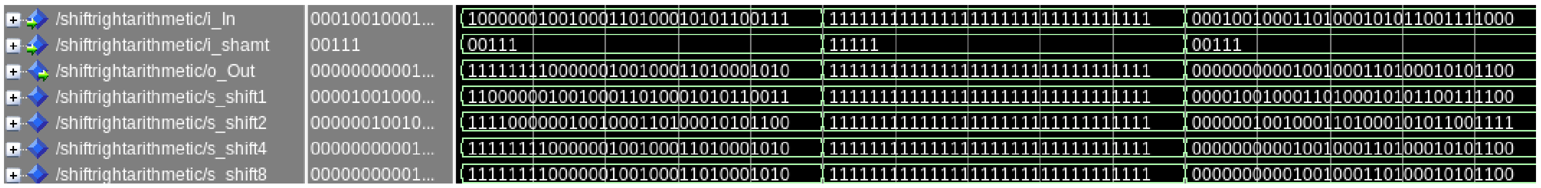
srl:



```
/shiftrightlogical/i_In      00011110010...  00010010001101000101011001111000  11111111111111111111111111111111  00011110010001011000111110010111
/shiftrightlogical/i_shamt   00000           00111                             11111                             00000
/shiftrightlogical/o_Out     00011110010...  00000000000100100011010001010110  00000000000000000000000000000001  00011110010001011000111110010111
/shiftrightlogical/s_shift1  00011110010...  00001001000110100010101100111100  01111111111111111111111111111111  00011110010001011000111110010111
/shiftrightlogical/s_shift2  00011110010...  00000010010001101000101011001111  00011111111111111111111111111111  00011110010001011000111110010111
/shiftrightlogical/s_shift4  00011110010...  00000000000100100011010001010110  00000001111111111111111111111111  00011110010001011000111110010111
/shiftrightlogical/s_shift8  00011110010...  00000000000100100011010001010110  00000000000000011111111111111111  00011110010001011000111110010111
```

This waveform shows the srl instruction. The first test was shifting by 7 bits, which it does correctly. The second test was shifting the max of 31 times, leaving just one bit at the end. The last test was performing no shift. In all cases, the sign bit is never preserved, and the shifted bits are filled with 0, as expected of srl.

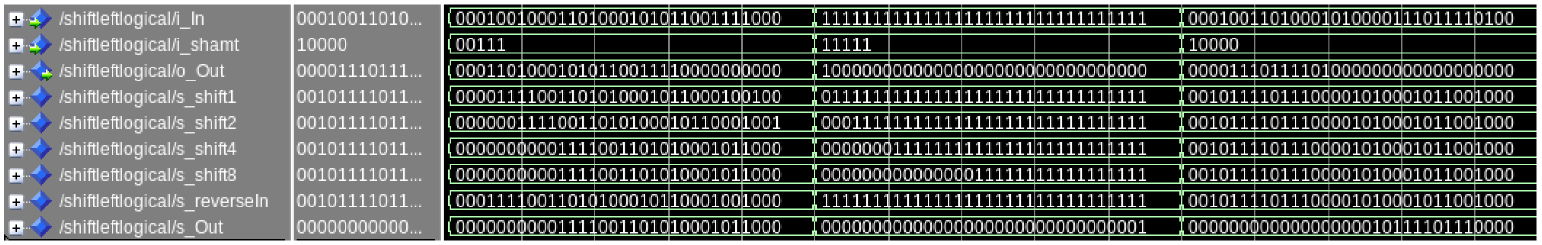
sra:



```
/shiftrightarithmetic/i_In      00010010001...  10000001001000110100010101100111  11111111111111111111111111111111  00010010001101000101011001111000
/shiftrightarithmetic/i_shamt   00111           00111                             11111                             00111
/shiftrightarithmetic/o_Out     00000000001...  11111111000000100100011010001010  11111111111111111111111111111111  00000000000100100011010001010110
/shiftrightarithmetic/s_shift1  00001001000...  11000000100100011010001010110011  11111111111111111111111111111111  00001001000110100010101100111100
/shiftrightarithmetic/s_shift2  00000010010...  11110000001001000110100010101100  11111111111111111111111111111111  00000010010001101000101011001111
/shiftrightarithmetic/s_shift4  00000000001...  11111111000000100100011010001010  11111111111111111111111111111111  00000000000100100011010001010110
/shiftrightarithmetic/s_shift8  00000000001...  11111111000000100100011010001010  11111111111111111111111111111111  00000000000100100011010001010110
```

This waveform shows the sra instruction. The first test was shifting a number with a sign by 7. As you can see, the sign is preserved, effectively dividing the number by 7. The second test shifts 31 times but preserves the sign, so no change is made. Lastly, a number is shifted by 7 that does not have a sign, so no sign bit needs to be preserved.

sll:



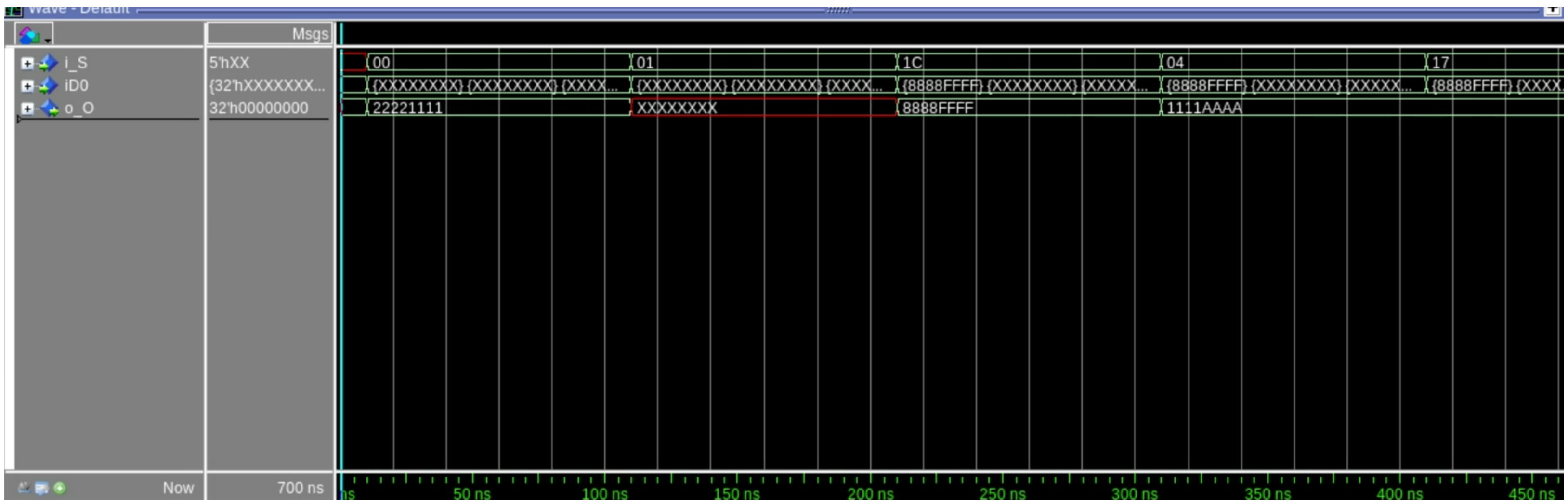
```
/shiftleftlogical/i_In          00010011010...  00010010001101000101011001111000  11111111111111111111111111111111  00010011010001010000111011110100
/shiftleftlogical/i_shamt       10000           00111                             11111                             10000
/shiftleftlogical/o_Out         00001110111...  00011010001010110011110000000000  10000000000000000000000000000000  00001110111101000000000000000000
/shiftleftlogical/s_shift1      00101111011...  00001111001101010001011000100100  01111111111111111111111111111111  00101111011100001010001011001000
/shiftleftlogical/s_shift2      00101111011...  00000011110011010100010110001001  00011111111111111111111111111111  00101111011100001010001011001000
/shiftleftlogical/s_shift4      00101111011...  00000000001111001101010001011000  00000001111111111111111111111111  00101111011100001010001011001000
/shiftleftlogical/s_shift8      00101111011...  00000000001111001101010001011000  00000000000000011111111111111111  00101111011100001010001011001000
/shiftleftlogical/s_reverseIn   00101111011...  00011110011010100010110001001000  11111111111111111111111111111111  00101111011100001010001011001000
/shiftleftlogical/s_Out         00000000000...  00000000001111001101010001011000  00000000000000000000000000000001  00000000000000000000101111011110000
```

This is the waveform for the sll instruction. The first test shifts a random number by 7. The second test shifts the number 32 times, leaving just the most significant bit. The last test shifts a random number 16 times. As you can see, all the shifted numbers are filled with 0 which is expected of sll.

[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.

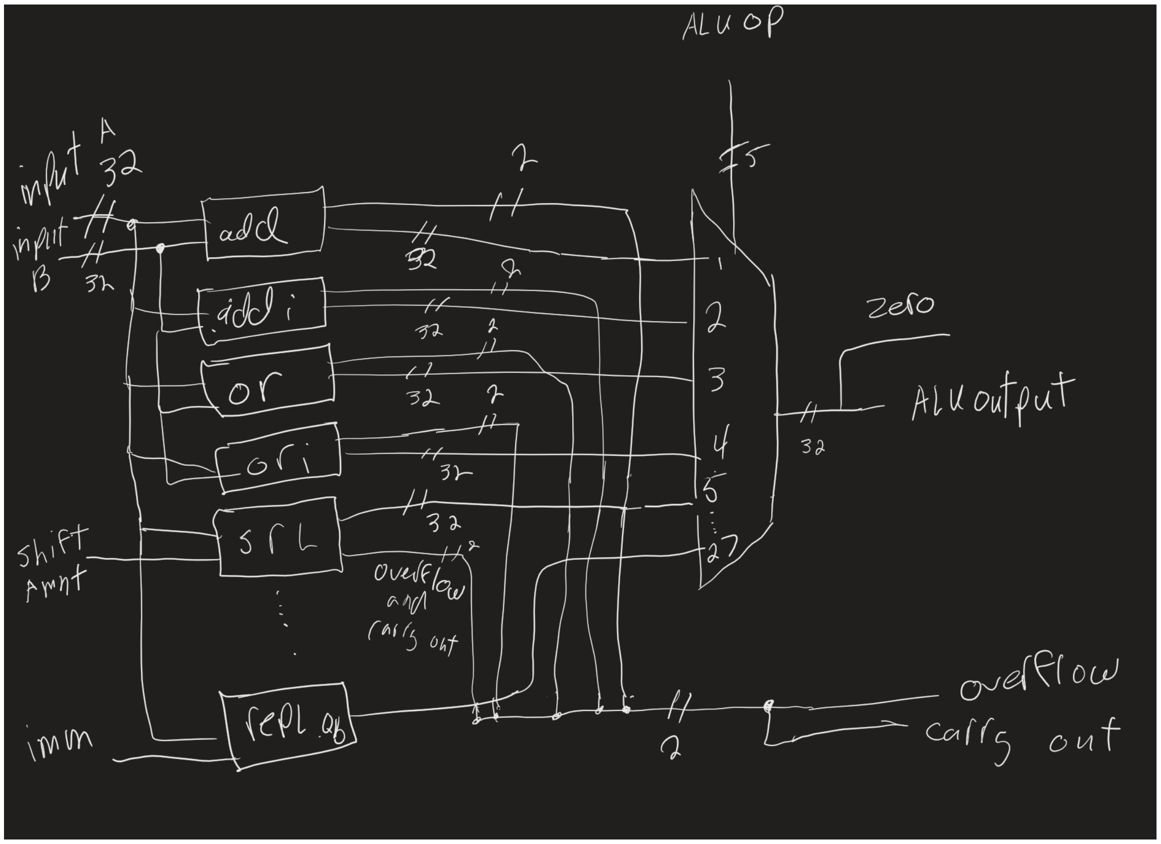
As mentioned before, instead of having our top level ALU schematic contain the shifter, adder, sign extender, and other functional units and then creating signals that choose what operation was done, we decided to make a schematic for every operation, even if they used the same functional unit. For example, srl, sll, and sra all have their own files. Obviously, they are very similar to each other but with a few differences. This was somewhat redundant since many operations are similar, but it didn't take much longer and made it a lot easier for us to understand which operation is chosen via the ALU opcode. Because of this, we also had to make a new input specifically for the repl instruction that is only active when the repl ALU op is chosen. We also had to create a 29-1 Multiplexor that the ALU Op Code chooses which instruction to take. Depending on the op code, that instruction is put through the multiplexer and only its output is outputted from the ALU

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



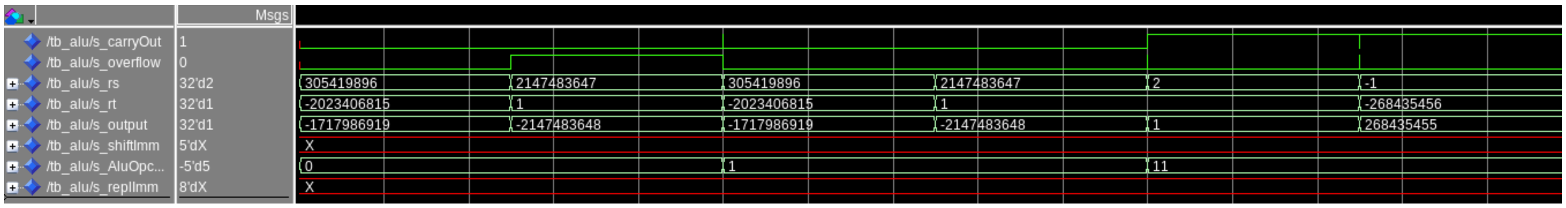
This waveform was a test for the 29 to 1 - 32 bit Multiplexor. This multiplexor is used by the ALU and takes in the ALUOpCode control bus from the control unit. This control bus will choose the output of the ALU as well as choose the overflow and carry-out bits. The above waveform is acting correctly. It is a bit difficult to explain what's going on in the waveform, but the testbench is in the ALU folder if you wanna look at it. All instructions will run ass if they were called, but only the correct instructions output will come out of the ALU due to this MUX and the ALUOpCode control signal.

[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?

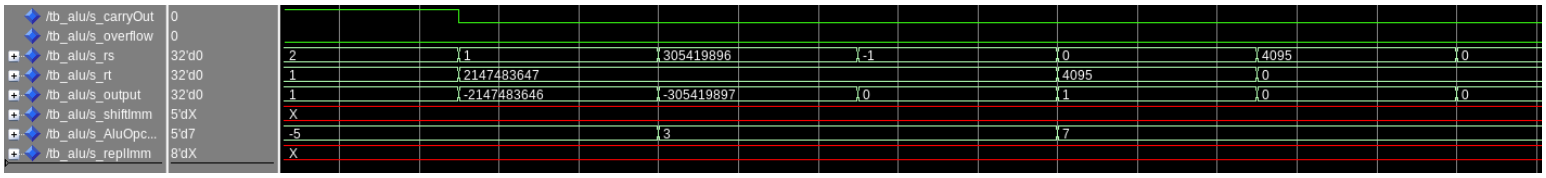


Overflow is calculated by taking the carry out bit and second to last carry out bit and XORing them. The zero is calculated by making the least significant bit of the ALU output either 1 or 0. We do this instead of making a whole different output in order to make the ALU outputs simple, all we need to do is correctly setup the bit to the right place in the top level processor. SLT is implemented by subtracting the rt register from rs and if the output is negative, then rs is smaller than rt and therefore the output of this instruction is 0x00000001. In the opposite case where the subtraction output is positive, then the output of the instruction is 0x00000000. That last bit is used by the branch instructions.

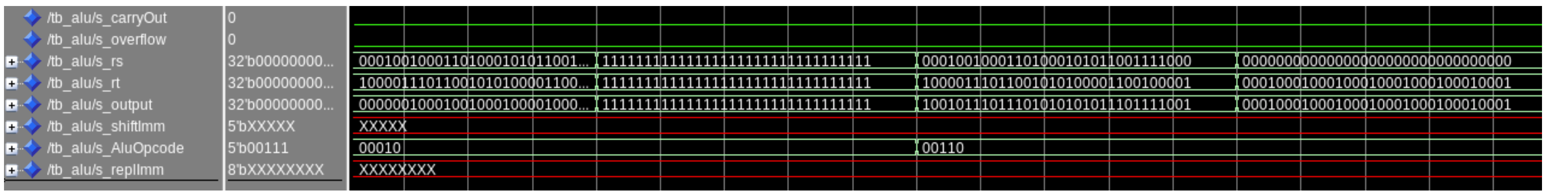
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



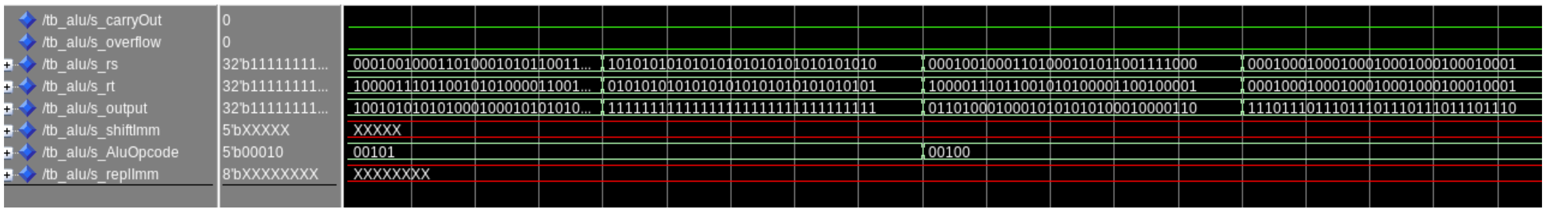
The first two tests are for **add (ALU op 0)** instruction. They both work as intended and calculate overflow as intended. The second two tests are for **addu (ALU op 1)**, they work the exact same as add but they don't care about overflow. The next two tests are for **sub (ALU op 11)** instruction. They both subtract the two inputs and make sure to keep the overflow. ShiftImm and replImm signals are always don't care unless we are shifting or replicating.



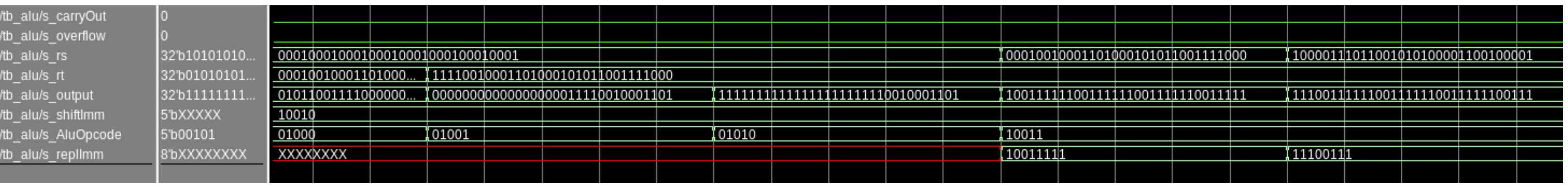
The next two tests are for **subu (ALU op -5)**, it works the same as sub but just doesn't care about overflow. The next test was for **not (ALU op 3)**, it simply inverts all of the bits, which in decimal would flip the sign and then subtract one as the output shows. The last three tests are for testing **slt (ALU op 7)**, which produces a 1 when RS is less than RT; this also includes the case in which they are both 0, then the output is still 0.



The next tests were for **and (ALU op 2)** instruction and **or (ALU op 6)** instruction, these cases are trivial but work perfectly.



The next tests here are for **xor (ALU op 5)** and **nor (ALU op 4)**, these are also trivial and work as expected even with edge cases such as all 0s or all 1s.


/tb_alu/s_carryOut 0
/tb_alu/s_overflow 0
/tb_alu/s_rs 32'b10101010... 00010001000100010001000100010001 00010010001101000101011001111000 10000111011001010100001100100001
/tb_alu/s_rt 32'b01010101... 0001001000110100... 11110010001101000101011001111000
/tb_alu/s_output 32'b11111111... 0101100111100000... 000000000000000000011110010001101 11111111111111111111110010001101 10011111100111111001111110011111 11100111111001111110011111100111
/tb_alu/s_shiftImm 5'bXXXXX 10010
/tb_alu/s_AluOpcode 5'b00101 01000 01001 01010 10011
/tb_alu/s_replImm 8'bXXXXXXXX XXXXXXXX 10011111 11100111

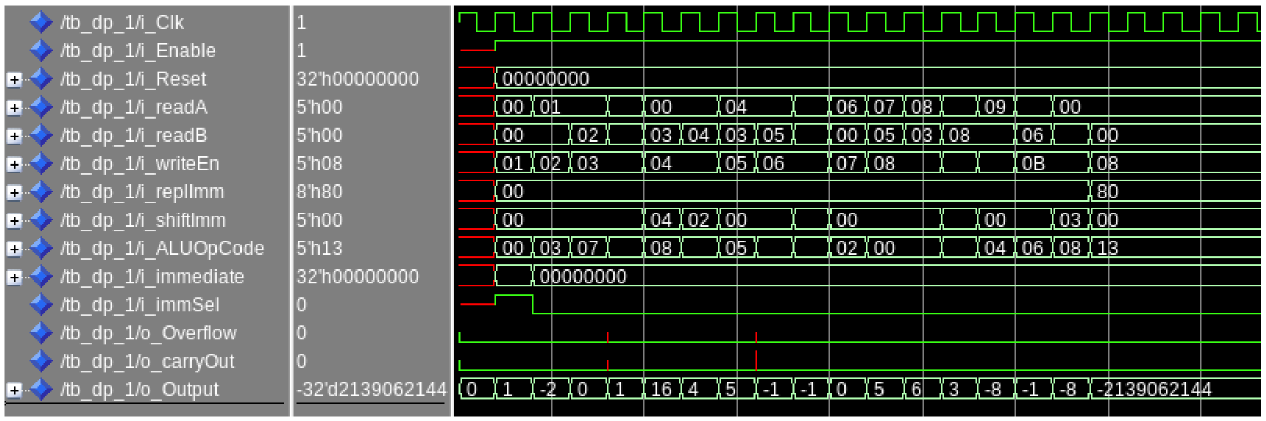
The next three tests are for the **three shifting (ALU op 8, 9, 10)** operations. As you can see, now the shiftImm signal is active. This was tested thoroughly in part 2 (c.i.4) so we did not include as many tests here, but they all work as intended. Lastly, we tested **repl.qb (ALU op 19)**. As you can see, now the replImm signal is active, and regardless of what the input is, the output replicates those 8 bits four different times.

Note that in this step, we did not test the immediate instructions because the bit extender is not technically part of the ALU, and they function the exact same was their non immediate counterparts once extended.

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

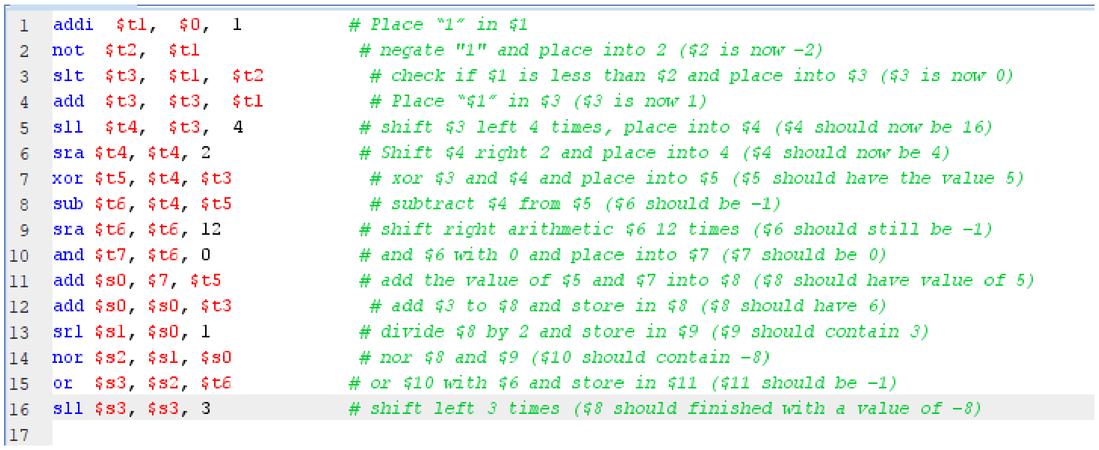
```
---ALU dataflow using lab 2 datapath---


addi  $1,  $0,  1          # Place “1” in $1
not  $2,  $1               # negate "1" and place into 2 ($2 is now -2)
slt  $3,  $1,  $2          # check if $1 is less than $2 and place into $3 ($3 is now 0)
add  $3,  $3,  $1          # Place “$1” in $3 ($3 is now 1)
sll  $4,  $3,  4           # shift $3 left 4 times, place into $4 ($4 should now be 16)
sra $4, $4, 2              # Shift $4 right 2 and place into 4 ($4 should now be 4)
xor $5, $4, $3             # xor $3 and $4 and place into $5 ($5 should have the value 5)
sub $6, $4, $5             # subtract $4 from $5 ($6 should be -1)
sra $6, $6, 12             # shift right arithmetic $6 12 times ($6 should still be -1)
and $7, $6, 0              # and $6 with 0 and place into $7 ($7 should be 0)
add $8, $7, $5             # add the value of $5 and $7 into $8 ($8 should have value of 5)
add $8, $8, $3             # add $3 to $8 and store in $8 ($8 should have 6)
srl $9, $8, 1              # divide $8 by 2 and store in $9 ($9 should contain 3)
nor $10, $9, $8            # nor $8 and $9 ($10 should contain -8)
or  $11, $10, $6           # or $10 with $6 and store in $11 ($11 should be -1)
sll $11, $11, 3             # shift left 3 times ($8 should finished with a value of -8)
repl $12, 01000000          #replicate 010000000 and place into $12.

final value of $12 should be 0x80808080 (-2139062144 in decimal) and the value of 11 should be -8 after everything
```

This is the test plan that we created. We believe it is comprehensive because it uses each of the major ALU operations at least once. This test did not include immediate values because the ALU does not include the bit extender so we had to load the first value in manually. Other than that, you can follow through to see how each register gets updated each instruction. $11 ends with a value of -8 and $12 ends with a value of 0x80808080 after the repl instruction. We did not test unsigned instructions to reduce redundancy and because they produce the same outputs aside from the overflow which we already verified in the previous step.

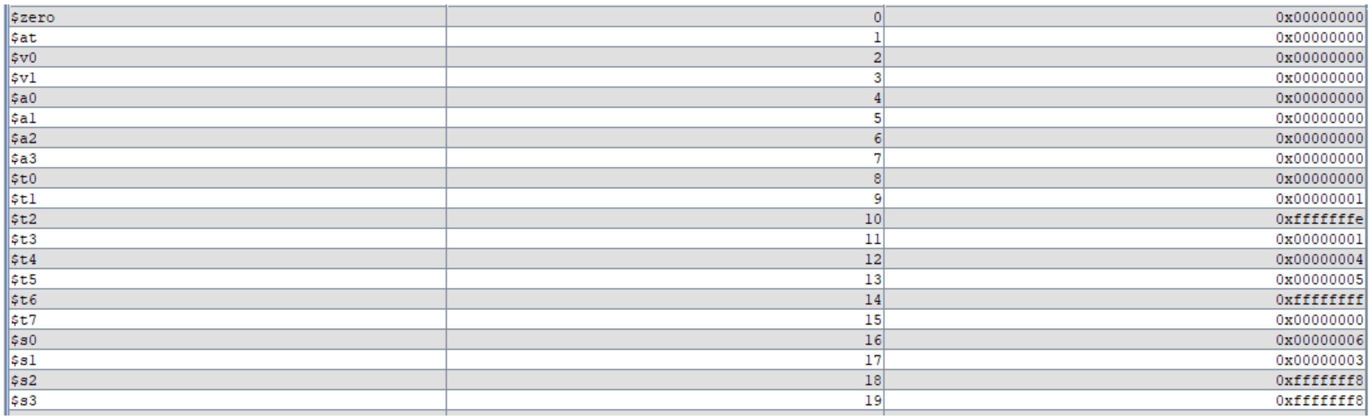


```
/tb_dp_1/i_Clk          1
/tb_dp_1/i_Enable       1
/tb_dp_1/i_Reset        32'h00000000
/tb_dp_1/i_readA        5'h00
/tb_dp_1/i_readB        5'h00
/tb_dp_1/i_writeEn      5'h08
/tb_dp_1/i_replImm      8'h80
/tb_dp_1/i_shiftImm     5'h00
/tb_dp_1/i_ALUOpCode    5'h13
/tb_dp_1/i_immediate    32'h00000000
/tb_dp_1/i_immSel       0
/tb_dp_1/o_Overflow     0
/tb_dp_1/o_carryOut     0
/tb_dp_1/o_Output       -32'd2139062144
```

Here is the waveform for our test. The signal o_Output contains the output after each instruction so you can follow it along to see how it changes. This also contains all of the other intermediate signals used to perform the operations.



```
addi  $t1,  $0,  1          # Place "1" in $1
not  $t2,  $t1               # negate "1" and place into 2 ($2 is now -2)
slt  $t3,  $t1,  $t2          # check if $1 is less than $2 and place into $3 ($3 is now 0)
add  $t3,  $t3,  $t1          # Place "$1" in $3 ($3 is now 1)
sll  $t4,  $t3,  4           # shift $3 left 4 times, place into $4 ($4 should now be 16)
sra $t4, $t4, 2              # Shift $4 right 2 and place into 4 ($4 should now be 4)
xor $t5, $t4, $t3             # xor $3 and $4 and place into $5 ($5 should have the value 5)
sub $t6, $t4, $t5             # subtract $4 from $5 ($6 should be -1)
sra $t6, $t6, 12             # shift right arithmetic $6 12 times ($6 should still be -1)
and $t7, $t6, 0              # and $6 with 0 and place into $7 ($7 should be 0)
add $s0, $7, $t5             # add the value of $5 and $7 into $8 ($8 should have value of 5)
add $s0, $s0, $t3             # add $3 to $8 and store in $8 ($8 should have 6)
srl $s1, $s0, 1              # divide $8 by 2 and store in $9 ($9 should contain 3)
nor $s2, $s1, $s0            # nor $8 and $9 ($10 should contain -8)
or  $s3, $s2, $t6           # or $10 with $6 and store in $11 ($11 should be -1)
sll $s3, $s3, 3             # shift left 3 times ($8 should finished with a value of -8)
```

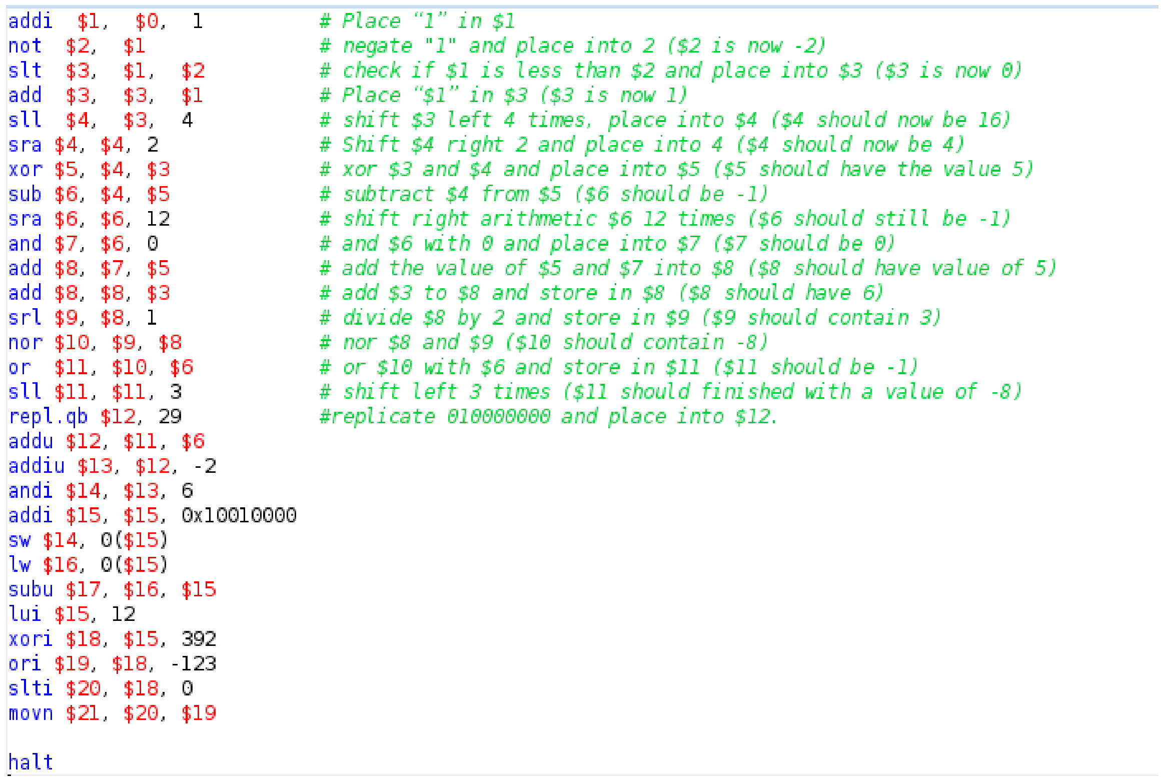


$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000001
$t2	10	0xfffffffe
$t3	11	0x00000001
$t4	12	0x00000004
$t5	13	0x00000005
$t6	14	0xffffffff
$t7	15	0x00000000
$s0	16	0x00000006
$s1	17	0x00000003
$s2	18	0xfffffff8
$s3	19	0xfffffff8

Here is our program ran in MARS. $S3 contains the output of $11 and finishes with a value -8 as expected. All of the other registers contain the correct values throughout the program (did not test repl on MARS).

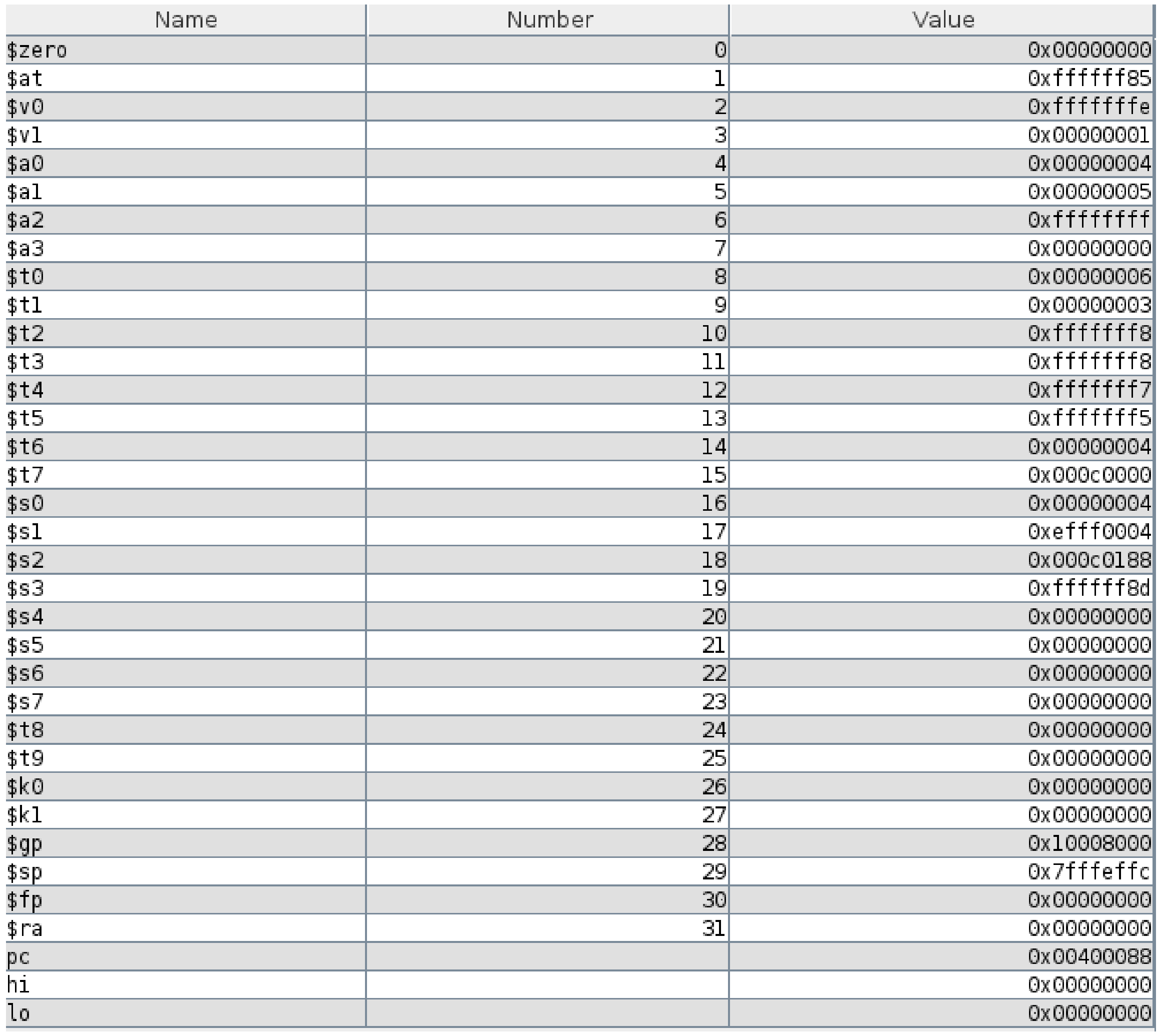
[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[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.



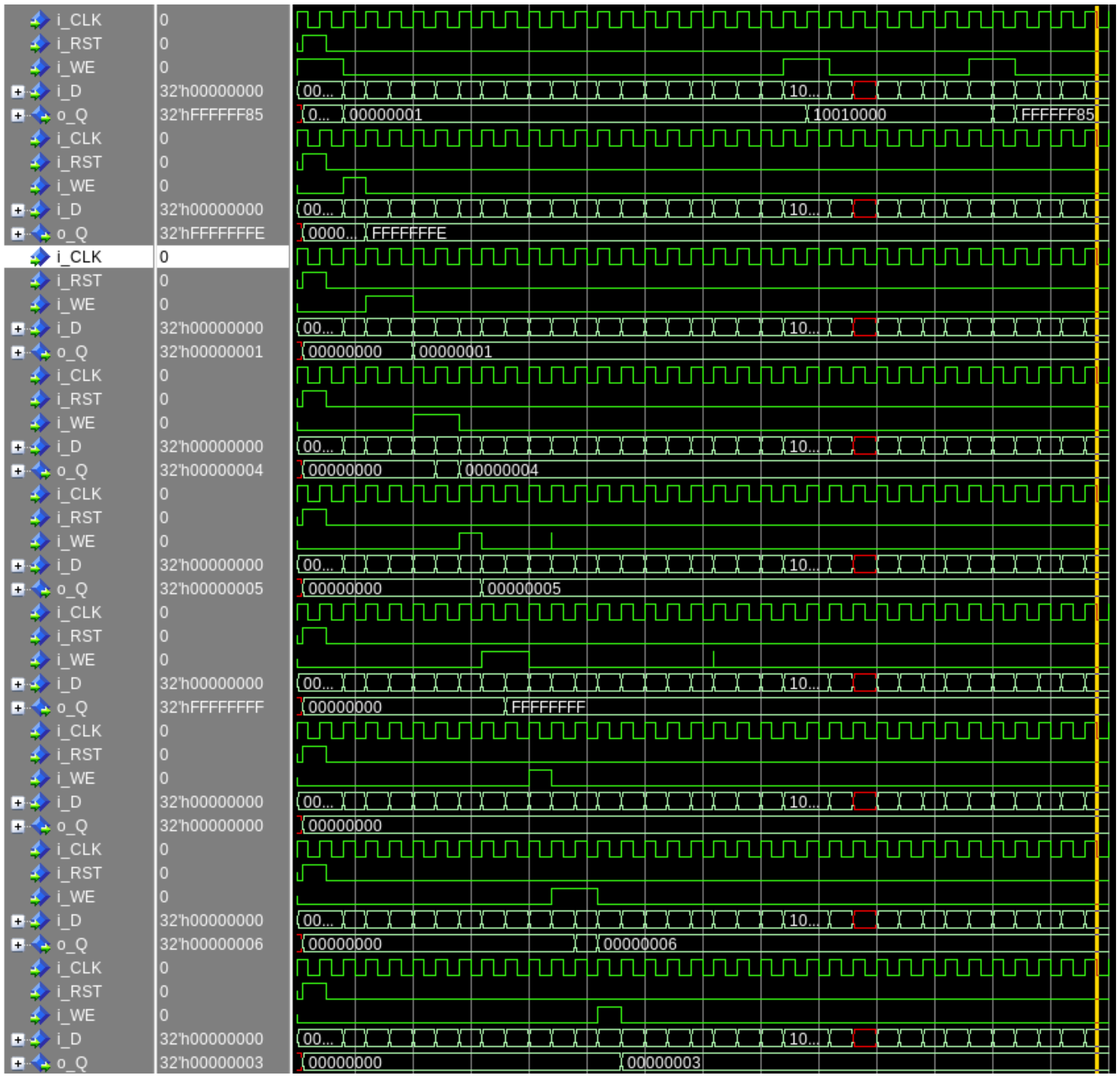
```
addi  $1,  $0,  1           # Place "1" in $1
not  $2,  $1                # negate "1" and place into 2 ($2 is now -2)
slt  $3,  $1,  $2           # check if $1 is less than $2 and place into $3 ($3 is now 0)
add  $3,  $3,  $1           # Place "$1" in $3 ($3 is now 1)
sll  $4,  $3,  4            # shift $3 left 4 times, place into $4 ($4 should now be 16)
sra $4, $4, 2               # Shift $4 right 2 and place into 4 ($4 should now be 4)
xor $5, $4, $3              # xor $3 and $4 and place into $5 ($5 should have the value 5)
sub $6, $4, $5              # subtract $4 from $5 ($6 should be -1)
sra $6, $6, 12              # shift right arithmetic $6 12 times ($6 should still be -1)
and $7, $6, 0               # and $6 with 0 and place into $7 ($7 should be 0)
add $8, $7, $5              # add the value of $5 and $7 into $8 ($8 should have value of 5)
add $8, $8, $3              # add $3 to $8 and store in $8 ($8 should have 6)
srl $9, $8, 1               # divide $8 by 2 and store in $9 ($9 should contain 3)
nor $10, $9, $8             # nor $8 and $9 ($10 should contain -8)
or  $11, $10, $6            # or $10 with $6 and store in $11 ($11 should be -1)
sll $11, $11, 3             # shift left 3 times ($11 should finished with a value of -8)
repl.qb $12, 29             #replicate 010000000 and place into $12.
addu $12, $11, $6
addiu $13, $12, -2
andi $14, $13, 6
addi $15, $15, 0x10010000
sw $14, 0($15)
lw $16, 0($15)
subu $17, $16, $15
lui $15, 12
xori $18, $15, 392
ori $19, $18, -123
slti $20, $18, 0
movn $21, $20, $19

halt
```

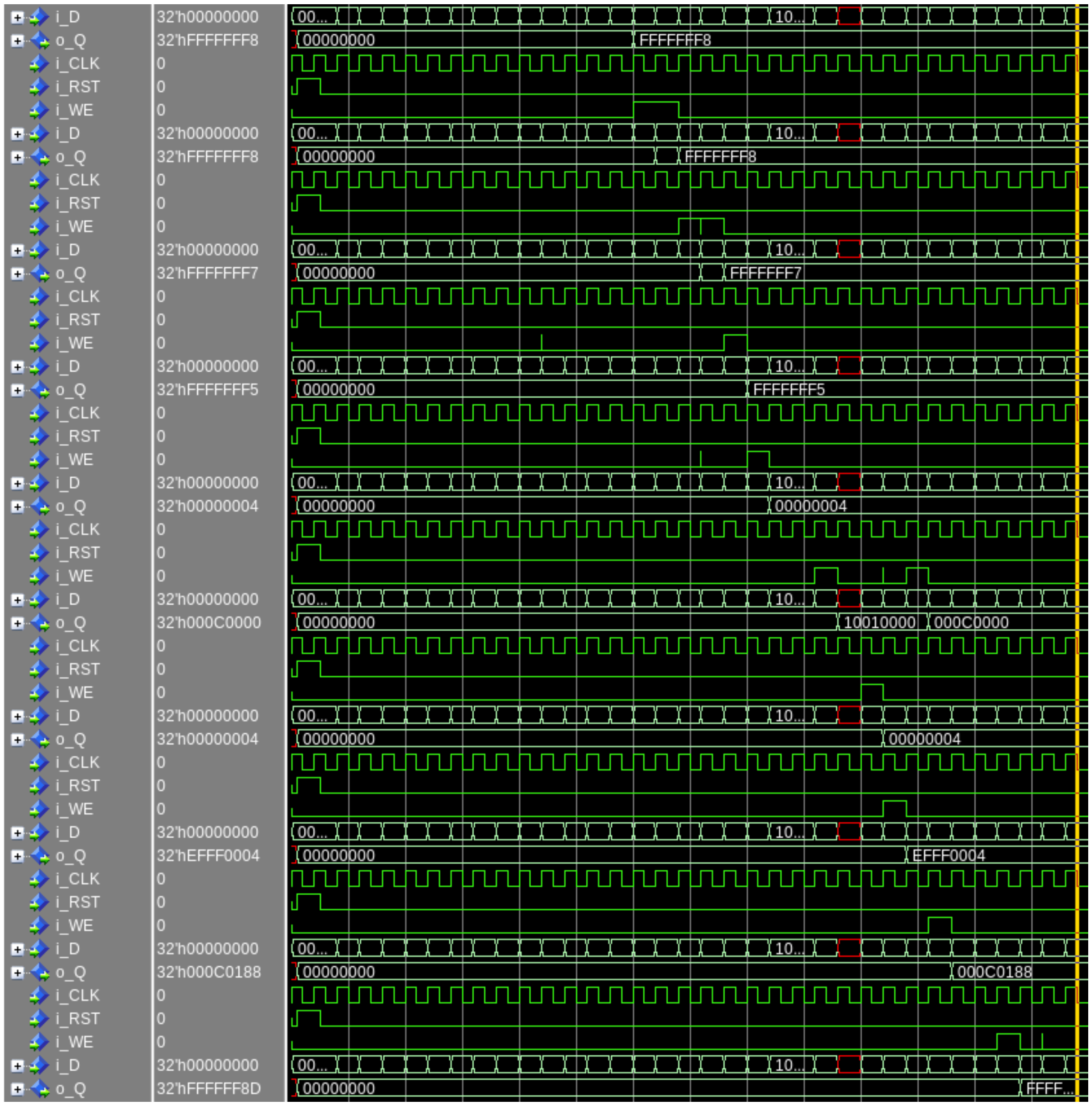
This is the Proj1_base_test.s file in Mars. It uses every required arithmetic/logical instruction at least once. It does not do anything important, but demonstrates the ability of the processor to perform instructions sequentially.

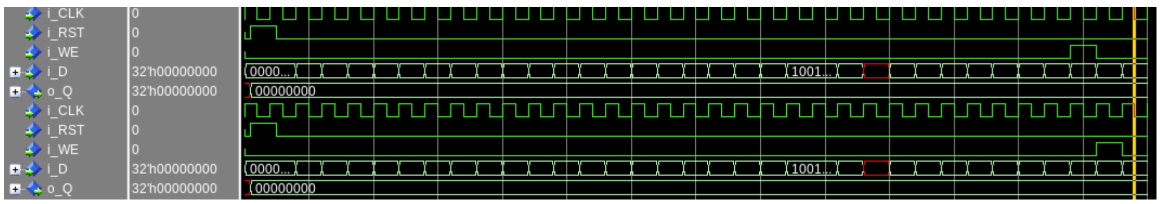


Name	Number	Value
$zero	0	0x00000000
$at	1	0xffffff85
$v0	2	0xfffffffe
$v1	3	0x00000001
$a0	4	0x00000004
$a1	5	0x00000005
$a2	6	0xffffffff
$a3	7	0x00000000
$t0	8	0x00000006
$t1	9	0x00000003
$t2	10	0xfffffff8
$t3	11	0xfffffff8
$t4	12	0xfffffff7
$t5	13	0xfffffff5
$t6	14	0x00000004
$t7	15	0x000c0000
$s0	16	0x00000004
$s1	17	0xefff0004
$s2	18	0x000c0188
$s3	19	0xffffff8d
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400088
hi		0x00000000
lo		0x00000000

These are the register outputs in Mars after running the previous program. Every register has its intended value.


i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'hFFFFFF85
i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'hFFFFFFFE
i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'h00000001
i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'h00000004
i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'h00000005
i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'hFFFFFFFF
i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'h00000000
i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'h00000006
i_CLK 0
i_RST 0
i_WE 0
i_D 32'h00000000
o_Q 32'h00000003
00...
10...
0...
00000001
10010000
FFFFFF85
0000...
FFFFFFFE
00000000
00000001
00000004
00000005
FFFFFFFF
00000006
00000003



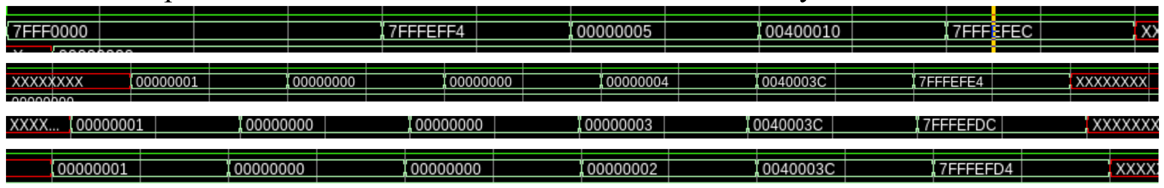


Here are the output waveforms generated by the tool flow tests. As you can see, each register contains the same values as it does in the Mars simulation ranging from register $1 to $21, indicating the processor works correctly.

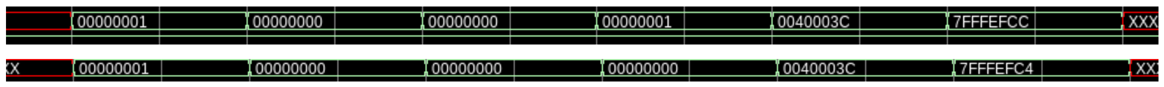
[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and .data (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

```
main:
        lui $sp, 0x7FFF
        ori $sp, 0xEFF4
        addi $a0, $0, 5 # set n = 5
        jal recursive
        addi $t9, $0, 55
        bne $v0, $t9, failure
        j exit

recursive:
        addi $sp, $sp, -8       # space for two words
        sw $ra, 4($sp)          # save return address
        sw $a0, 0($sp)          # temporary variable to hold n
        li $v0, 1
        slti $t0, $a0, 1
        bne $t0, $0, recexit
        addi $a0, $a0, -1
        jal recursive
        lw $a0, 0($sp)          # retrieve original n
        add $v0, $v0, $a0       # n + ((n - 1) + (n - 2) + ... (n - n))

recexit:
        lw $ra 4($sp)           # restore $ra
        addi $sp, $sp, 8        # restore $sp
        jr $ra                  # back to caller

failure:
        addi $t8, $0, 1
        j exit

exit:
        li $v0, 5
        halt
```

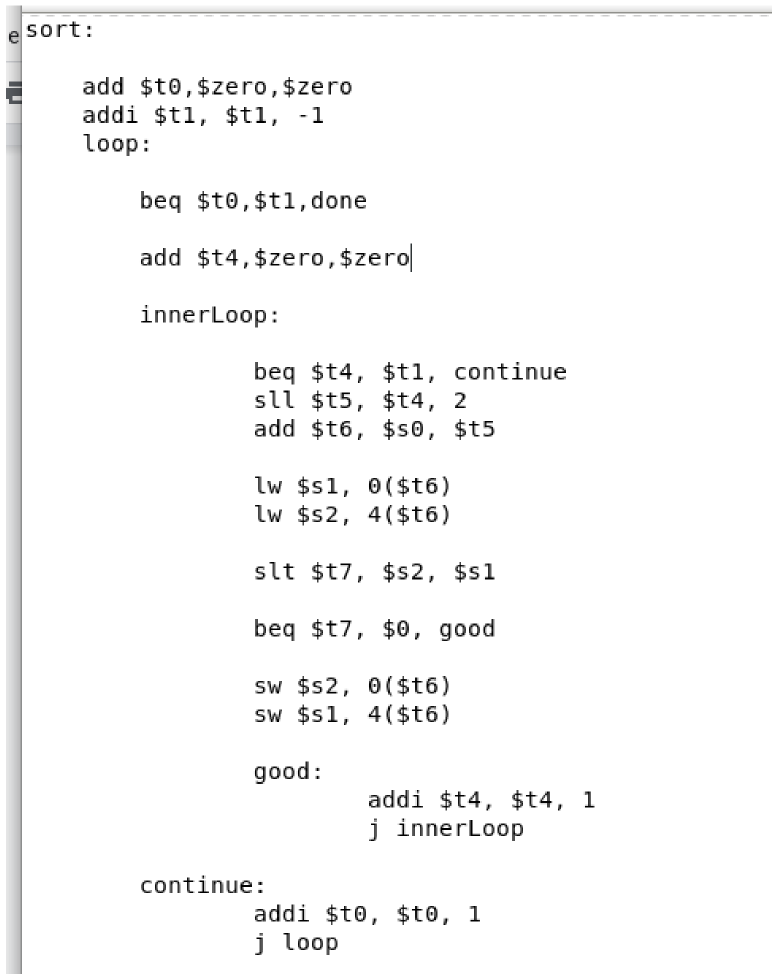
This is a script that takes a value n and calculates n! recursively.



```
7FFF0000   7FFFEFF4   00000005   00400010   7FFFEFEC   XX
XXXXXXXX   00000001   00000000   00000000   00000004   0040003C   7FFFEFE4   XXXXXXXX
XXXX...    00000001   00000000   00000000   00000003   0040003C   7FFFEFDC   XXXXXXX
           00000001   00000000   00000000   00000002   0040003C   7FFFEFD4   XXXX
```

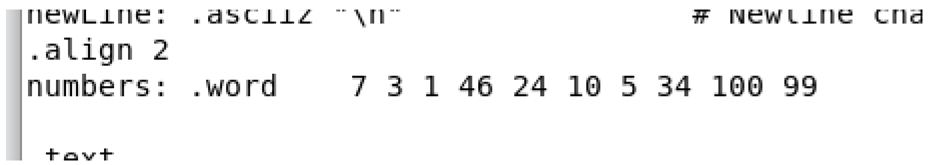


[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.



```
sort:

    add $t0,$zero,$zero
    addi $t1, $t1, -1
    loop:

        beq $t0,$t1,done

        add $t4,$zero,$zero

        innerLoop:

                beq $t4, $t1, continue
                sll $t5, $t4, 2
                add $t6, $s0, $t5

                lw $s1, 0($t6)
                lw $s2, 4($t6)

                slt $t7, $s2, $s1

                beq $t7, $0, good

                sw $s2, 0($t6)
                sw $s1, 4($t6)

                good:
                        addi $t4, $t4, 1
                        j innerLoop

        continue:
                addi $t0, $t0, 1
                j loop
```

This is the sorting algorithm for bubble sort, accessing the numbers in the array from memory and switching them if the first number is larger then the second number.

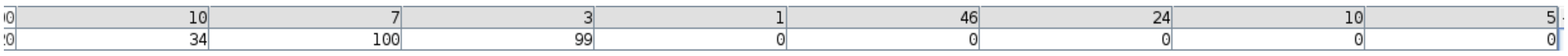


```
newLine: .asciiz "\n"                    # Newline cha
.align 2
numbers: .word    7 3 1 46 24 10 5 34 100 99

 .text
```

This is the starting array of numbers for bubble sort. These numbers can be changed to anything and make any size of array and the bubble sort code we wrote will still work.

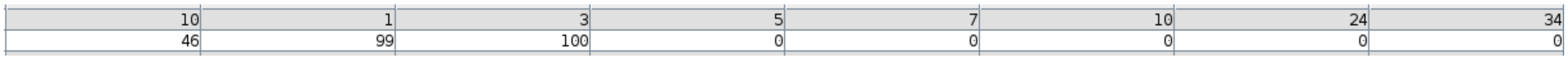
The waveforms below show the order the sorted numbers in the array which is the correct order found in the mars simulator

before:
The array starts at 7

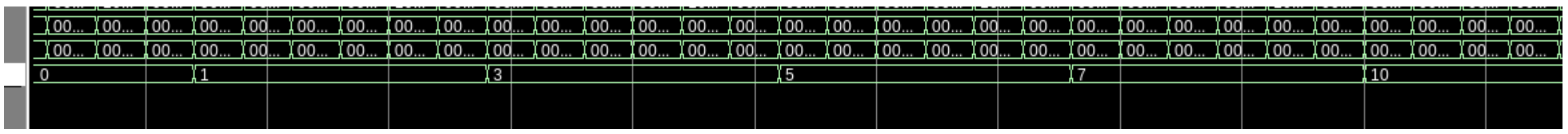


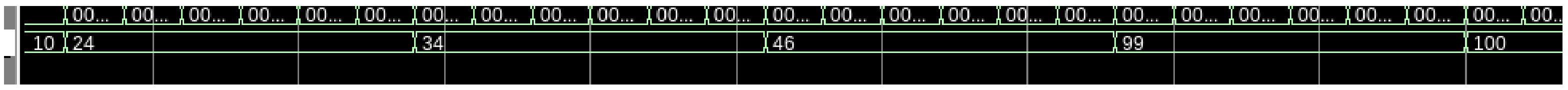
10	7	3	1	46	24	10	5
34	100	99	0	0	0	0	0

after:
the array starts at 1



10	1	3	5	7	10	24	34
46	99	100	0	0	0	0	0



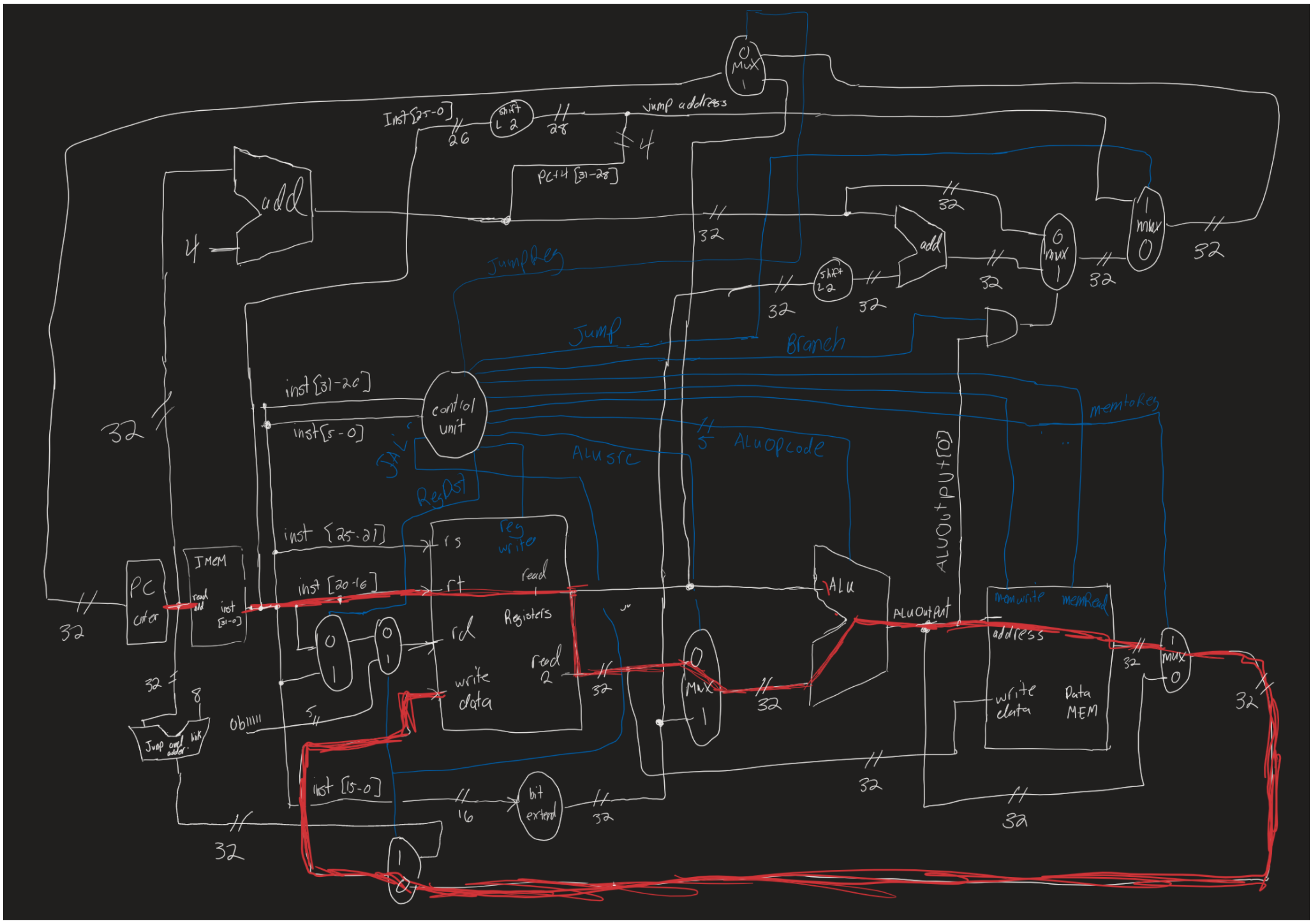


[Part 4] report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematics. What components would you focus on to improve the frequency?

Max frequency is 20.39mhz

We would probably need to optimize the ALU because no matter what instruction is called, all instructions run in the ALU and only the correct instruction is outputted out of the ALU. So changing that fact would greatly increase our frequency

The red line below is the critical path, the critical path goes through the reg file, the ALU, both memory files, and a couple mux's


0 Mux 1
jump address
Inst[25-0]
26
shift L2
28
4
PC+4 [31-28]
add
4
32
32
add
32
0 mux 1
32
1 mux 0
32
JumpReg
shift L2
32
32
Jump
Branch
inst[31-26]
inst[5-0]
control unit
JAL
RegDst
ALUsrc
5
ALUopcode
memtoReg
ALUOutput[0]
32
PC cter
IMEM
read add
inst [31-0]
inst [25-21]
rs
reg write
inst [20-16]
rt
read
Registers
rd
write data
read 2
32
0 1
0 1
0b11111
5
32
8
Jump and link adder
32
inst [15-0]
16
bit extend
32
0 Mux 1
32
ALU
ALUOutput
memwrite
memRead
address
write data
Data MEM
32
1 mux 0
32
32
32
1 0