CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: ____Owen Jewell_____

____Corey Heithoff_____

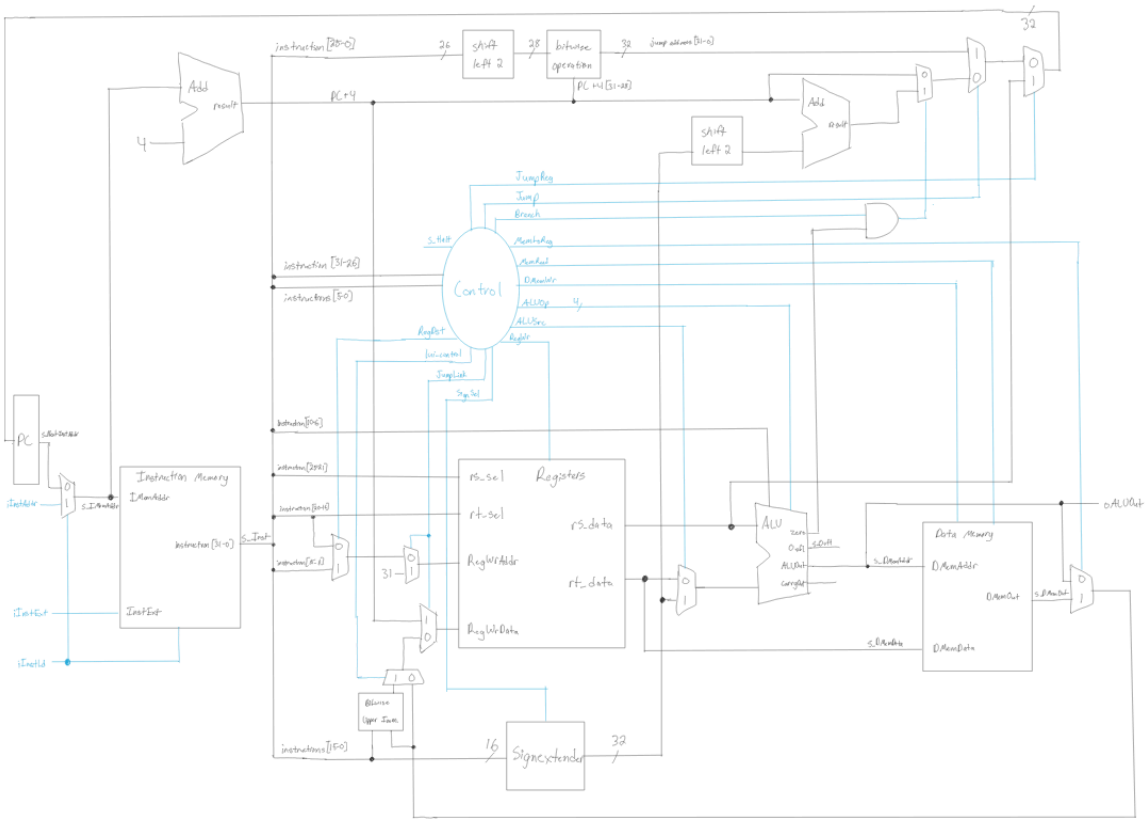
____Jason Di Giovanni_____

____Luca Cano_____

Project Teams Group #:_____Section C, Group 4_____

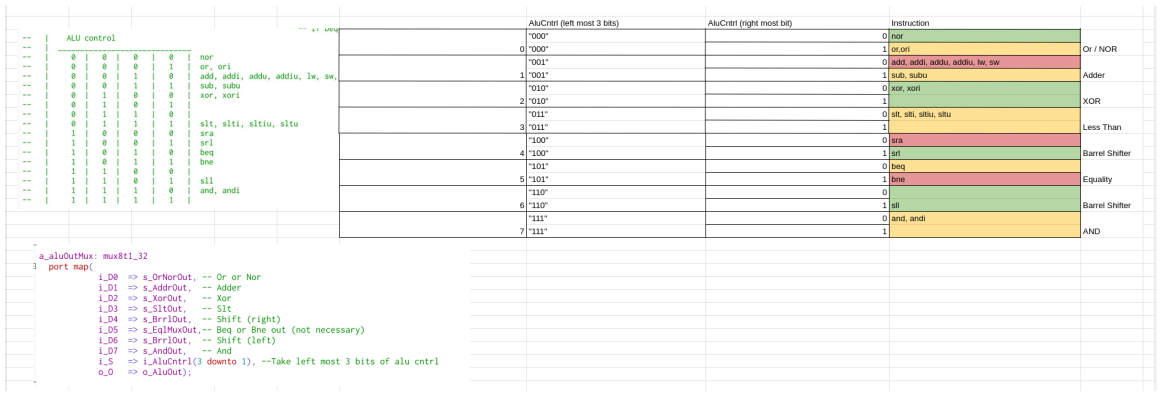
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

[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	Funct (If applicable)	ALUSrc	ALUControl	MemtoReg (Does function read from memory)	s_DMemWr (Does function write to memory)	s_RegWr (Does function write to a register)	RegDst (Function uses R-type instead of I-type)	Unsigned
addi	"001000"	NA	1	0010	0	0	1	0	0
add	"000000"	"100000"	0	0010	0	0	1	1	0
addiu	"000000"	NA	1	0010	0	0	1	0	1
addu	"000000"	"100001"	0	0010	0	0	1	1	1
and	"000000"	"100100"	0	1110	0	0	1	1	0
andi	"001100"	NA	1	1110	0	0	1	0	0
lui	"001111"	NA	1	NA	0	0	1	0	0
lw	"100011"	NA	1	0010	1	0	1	0	0
nor	"000000"	"100111"	0	0000	0	0	1	1	0
xor	"000000"	"100110"	0	0100	0	0	1	1	0
xori	"001110"	NA	1	0100	0	0	1	0	0
or	"000000"	"100101"	0	0001	0	0	1	1	0
ori	"001101"	NA	1	0001	0	0	1	0	0
slt	"000000"	"101010"	0	0111	0	0	1	1	0
slti	"001010"	NA	1	0111	0	0	1	0	0
sltiu	"001011"	NA	1	0111	0	0	1	0	1
sltu	"000000"	"101011"	0	0111	0	0	1	1	1
sll	"000000"	"000000"	0	1101	0	0	1	1	0
srl	"000000"	"000010"	0	1001	0	0	1	1	0
sra	"000000"	"000011"	0	1000	0	0	1	1	0
sllv	"000000"	"000100"	0	NA	0	0	1	1	0
srlv	"000000"	"000110"	0	NA	0	0	1	1	0
srav	"000000"	"000111"	0	NA	0	0	1	1	0
sw	"101011"	NA	1	0010	0	1	0	0	0
sub	"000000"	"100010"	0	0011	0	0	1	1	0
subu	"000000"	"100011"	0	0011	0	0	1	1	1
beq	"000100"	NA	0	1010	0	0	0	0	0
bne	"000101"	NA	0	1011	0	0	0	0	0
j	"000010"	NA	0	NA	0	0	0	0	0
jal	"000011"	NA	0	NA	0	0	1	0	0
jr	"000000"	"001000"	0	NA	0	0	0	1	0
halt	"010001"	NA	0	NA	0	0	0	0	0
noop	"111111"	NA	NA	NA	NA	NA	NA	NA	NA

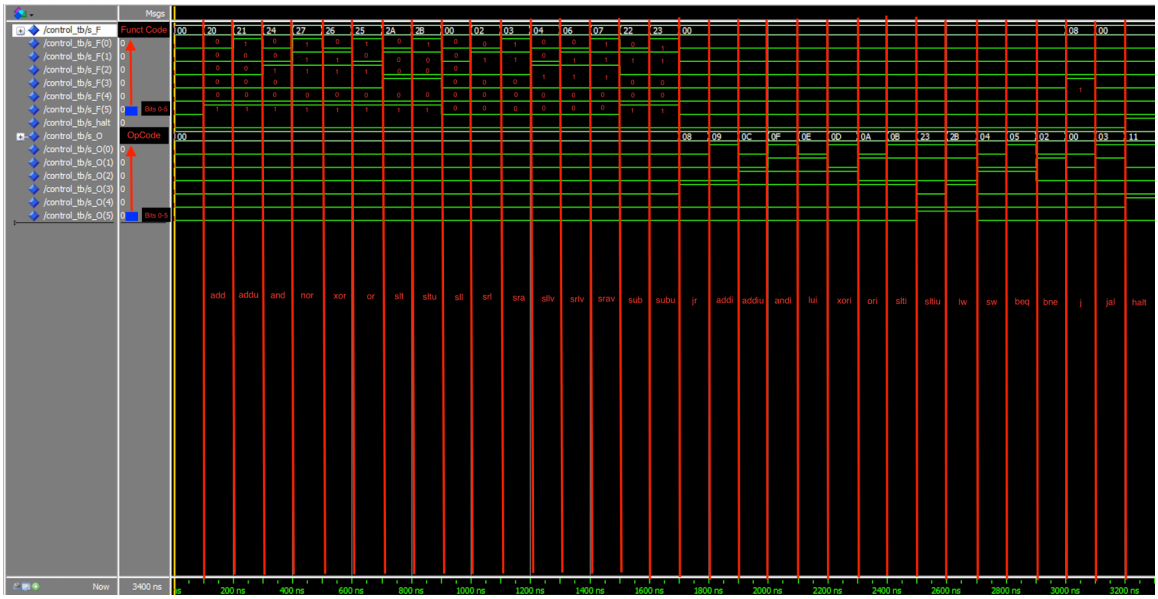


```
--  |   ALU control
--  |   ____________________
--  |    0  |  0  |  0  |  0  |  nor
--  |    0  |  0  |  0  |  1  |  or, ori
--  |    0  |  0  |  1  |  0  |  add, addi, addu, addiu, lw, sw,
--  |    0  |  0  |  1  |  1  |  sub, subu
--  |    0  |  1  |  0  |  0  |  xor, xori
--  |    0  |  1  |  0  |  1  |
--  |    0  |  1  |  1  |  0  |
--  |    0  |  1  |  1  |  1  |  slt, slti, sltiu, sltu
--  |    1  |  0  |  0  |  0  |  sra
--  |    1  |  0  |  0  |  1  |  srl
--  |    1  |  0  |  1  |  0  |  beq
--  |    1  |  0  |  1  |  1  |  bne
--  |    1  |  1  |  0  |  0  |
--  |    1  |  1  |  0  |  1  |  sll
--  |    1  |  1  |  1  |  0  |  and, andi
--  |    1  |  1  |  1  |  1  |
```

	AluCntrl (left most 3 bits)	AluCntrl (right most bit)	Instruction	
	"000"	0	nor	
0	"000"	1	or,ori	Or / NOR
	"001"	0	add, addi, addu, addiu, lw, sw	
1	"001"	1	sub, subu	Adder
	"010"	0	xor, xori	
2	"010"	1		XOR
	"011"	0	slt, slti, sltiu, sltu	
3	"011"	1		Less Than
	"100"	0	sra	
4	"100"	1	srl	Barrel Shifter
	"101"	0	beq	
5	"101"	1	bne	Equality
	"110"	0		
6	"110"	1	sll	Barrel Shifter
	"111"	0	and, andi	
7	"111"	1		AND

```
a_aluOutMux: mux8t1_32
  port map(
            i_D0  => s_OrNorOut, -- Or or Nor
            i_D1  => s_AddrOut,  -- Adder
            i_D2  => s_XorOut,   -- Xor
            i_D3  => s_SltOut,   -- Slt
            i_D4  => s_BrrlOut,  -- Shift (right)
            i_D5  => s_EqlMuxOut,-- Beq or Bne out (not necessary)
            i_D6  => s_BrrlOut,  -- Shift (left)
            i_D7  => s_AndOut,   -- And
            i_S   => i_AluCntrl(3 downto 1), --Take left most 3 bits of alu cntrl
            o_O   => o_AluOut);
```

[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).



Each instruction matches the expected Opcode or Function code depending on the case of whether it is an R-type or an I-Type instruction. Refer to the sheet for control logic in part 2 a.i .

[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 fetch logic is responsible for updating the program counter. For most cycles, it increments the program counter (PC) by 4, which is the very next instruction in memory since MIPS memory is normally byte-addressable and instructions are 4 bytes. For every instruction, PC = PC + 4, except for instructions beq, bne, j, jr, and jal. For beq, if the values in two registers are equal (determined in ALU), PC = PC + 4 + BranchAddr*4, where BranchAddr is a sign-extended immediate value given by the instruction. For bne, if the values in two registers are not equal (determined in ALU), PC = PC + 4 + BranchAddr*4, where BranchAddr is a sign-extended immediate value given by the instruction.

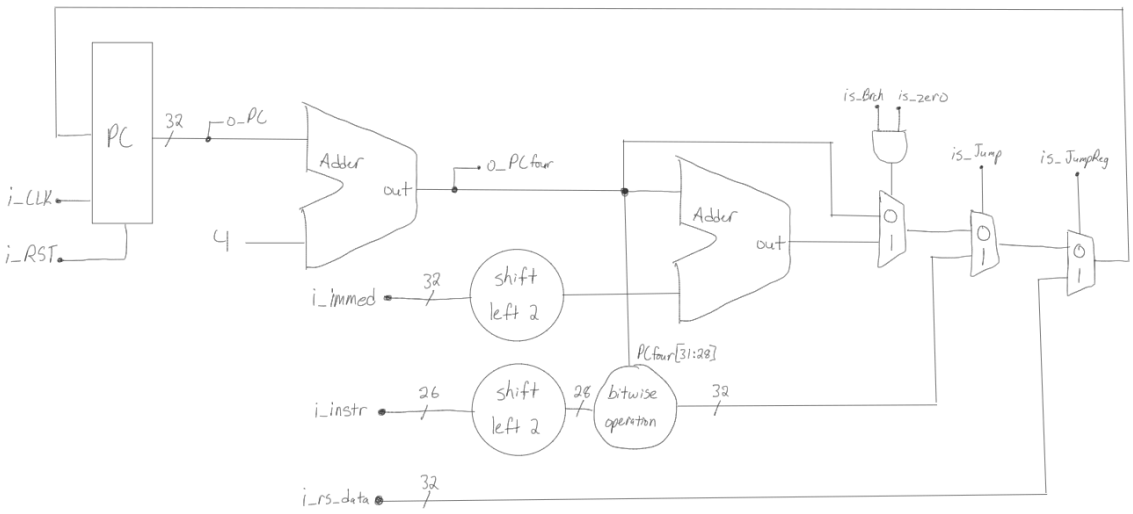
For j, PC = JumpAddr, where JumpAddr is the first 4 bits of the current PC value, and the rest of the bits are an address value given by the instruction times 4. For jr, PC = value in the register given by the instruction. For jal, two actions happen, first, the return address register ($ra) is set to the value of PC + 4, however, this action is completed outside of the fetch logic. Second, PC = JumpAddr, where JumpAddr is the first 4 bits of the current PC value, and the rest of the bits are an address value given by the instruction times 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?

The fetch logic system has nine inputs and two outputs. The outputs include the current values of the PC and PC + 4. The inputs include two functional inputs, the clock and the reset for the PC. Inputs include three data inputs, the sign-extended immediate value (i_immed) for beq/bne, the last 26 bits of the current instruction (i_instr) for j/jal, and the data output of the rs register file read port (i_rs_data) for jr.

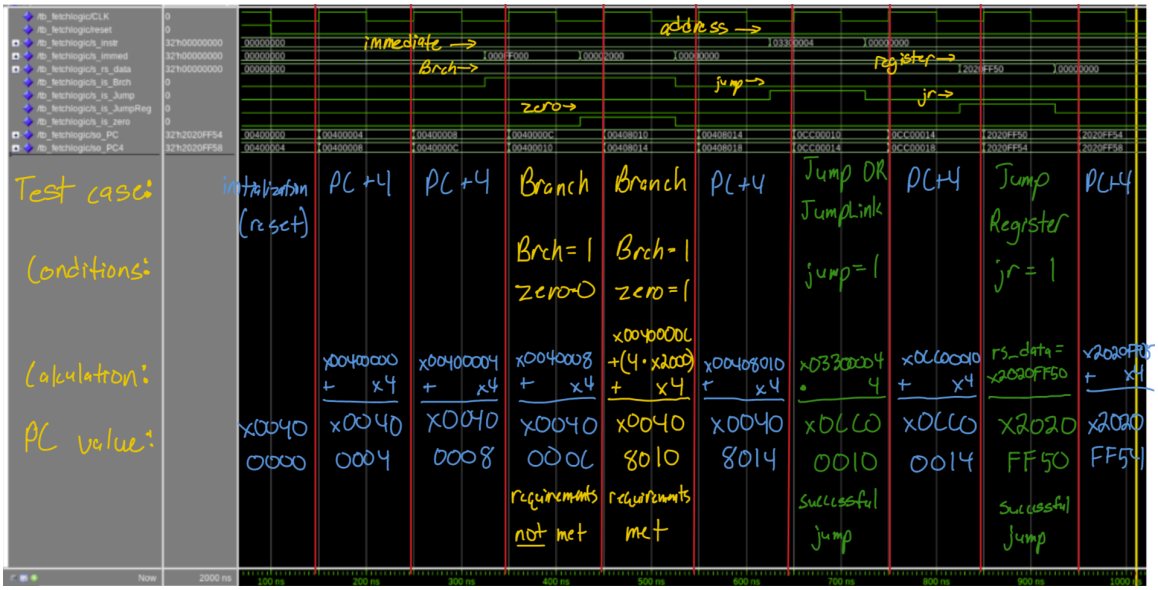
The inputs also include four needed control signals, the branch control from the Control Unit (is_Brch), which is "1" for beq/bne, the 1-bit Zero comparison output from the ALU (is_zero), which is "1" if the comparison is true, the jump control from the Control Unit (is_Jump), which is "1" for j/jal, and the jump register control from the Control Unit (is_JumpReg), which is "1" for jr. Control signals is_Brch and is_zero are ANDed together for final determination if a branch is happening.

Fetch Logic Schematic:



[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.

Fetch Logic Test Bench:



The fetch logic test bench performed 10 test cases. The 1st case was reset, where the PC value was initialized to x00400000. The next two test cases are simple PC = PC + 4 operations where all branching and jumping control flow signals are zero. The 4th test case simulated an unsuccessful branch instruction where the branch control flow signal is equal to 1, but the zero control input from the ALU is 0, so the conditions to branch were not met. The 5th test case simulated a successful branch instruction where the branch control flow signal is equal to 1 and the zero control input from the ALU is 1, so the PC is updated with the given immediate value multiplied by 4 and then added to PC + 4. The 6th test case is a simple PC = PC + 4 operation. The 7th test case simulated a jump or jal instruction where the jump control flow signal is equal to 1, so the PC is updated with the given address value multiplied by 4. The 8th test case is another simple PC = PC + 4 operation. The 9th test case simulated a jump register instruction where the jr control flow signal is equal to 1, so the PC is updated with the given register value. The last test case is another PC = PC + 4 operation where all branching and jumping control flow signals are zero.

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

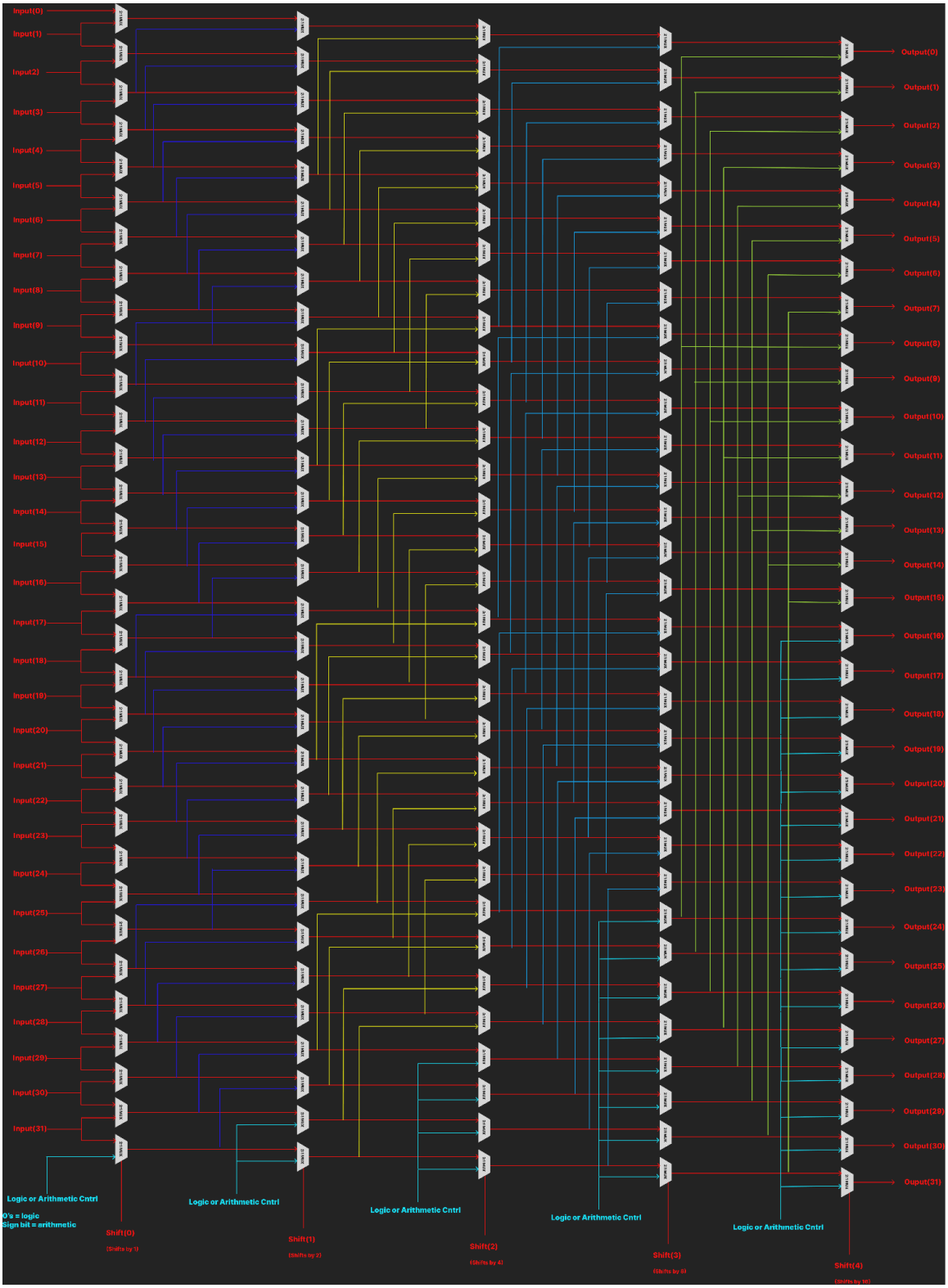
Shift right logical (srl) is a right shift that always adds 0 to the left side to replace bits that fell off the right side when being shifted. Shift right arithmetic (sra) is a right shift that always adds the original sign bit to the left side to replace bits that fell off the right side when being shifted, this preserves the sign of the number after the shift. MIPS has no need for a sla arithmetic instruction because you only ever need to add 0's to the right side for a left shift because the sign won't be affected. This left logical shift is the only needed left shift and can be handled by shift left logical (sll).

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

My barrel shifter starts with a mux to determine the shift type if the shift type bit is a 0 that is a logical shift that extends the number after the shift with 0's. If the shift type is a 1 that is an arithmetic shift that extends the number with the 31st bit of the original number. If the shift direction bit is a 0, that is a left shift, and if the shift direction bit is a 1, then that is a right shift. In order to handle a left shift after implementing a right shift, all I do is reverse the vector before and after the shift if it is a left shift.

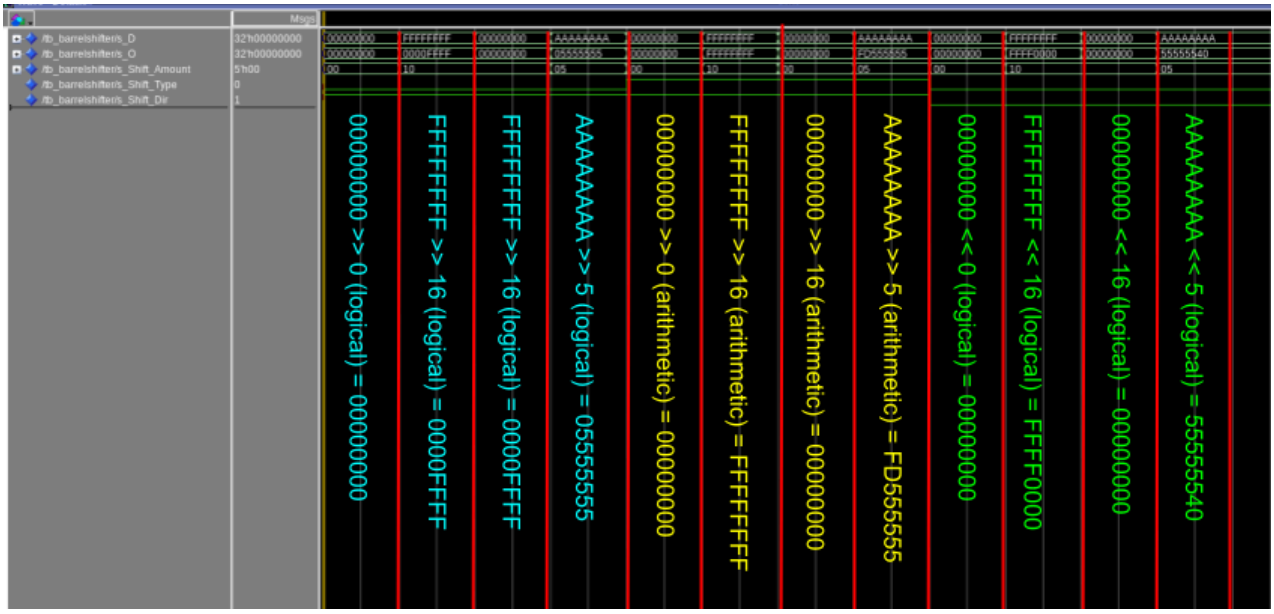
[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

My barrel shifter has a mux to determine the shift direction. If the shift direction bit is a 0 that is a left shift, and if the shift direction bit is a 1, then that is a right shift. In order to handle a left shift after implementing a right shift, all I do is reverse the vector before and after the shift if it is a left shift. This means the bits being moved forward for the shift are actually being moved backward, turning it into a left shift instead of right.


Input(0)
Input(1)
Input(2)
Input(3)
Input(4)
Input(5)
Input(6)
Input(7)
Input(8)
Input(9)
Input(10)
Input(11)
Input(12)
Input(13)
Input(14)
Input(15)
Input(16)
Input(17)
Input(18)
Input(19)
Input(20)
Input(21)
Input(22)
Input(23)
Input(24)
Input(25)
Input(26)
Input(27)
Input(28)
Input(29)
Input(30)
Input(31)
Output(0)
Output(1)
Output(2)
Output(3)
Output(4)
Output(5)
Output(6)
Output(7)
Output(8)
Output(9)
Output(10)
Output(11)
Output(12)
Output(13)
Output(14)
Output(15)
Output(16)
Output(17)
Output(18)
Output(19)
Output(20)
Output(21)
Output(22)
Output(23)
Output(24)
Output(25)
Output(26)
Output(27)
Output(28)
Output(29)
Output(30)
Ouput(31)
Logic or Arithmetic Cntrl
0's = logic
Sign bit = arithmetic
Shift(0)
(Shifts by 1)
Logic or Arithmetic Cntrl
Shift(1)
(Shifts by 2)
Logic or Arithmetic Cntrl
Shift(2)
(Shifts by 4)
Logic or Arithmetic Cntrl
Shift(3)
(Shifts by 8)
Logic or Arithmetic Cntrl
Shift(4)
(Shifts by 16)

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

Barrel Shifter



I performed 4 tests for each type of shift that the barrel shift is capable of doing (shift right logical, shift right arithmetic, and shift left logical). As can be seen in the waveform, the output of the barrel shift matches the expected output based on the input from the 12 test cases. This means that when a logical shift is done, the number is padded with 0's, and when an arithmetic shift is done, the number is padded with th original sign bit. Also, the numbers are correctly shifted in the appropriate direction based on the signal s_Shift_Dir.

[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.

We chose to implement a carry-lookahead adder to replace the ripple carry adder we made in the labs. This is because the ripple in the ripple carry adder is very slow. Our design starts with a mux that decides whether or not an addition or subtraction will take place based on the 1-bit n_AddSub bit input. If this bit is a 0, we do addition, but if the bit is a 1, we do subtraction and must first take the ones complement of the number and then add 1 (the carry-in or n_AddSub bit) to get the twos complement. From there, we generate 30 1-bit full adders. We then generate the signal assignments for the carry look-ahead logic as described in the lecture slides.

Carry-Lookahead Adder

j=0

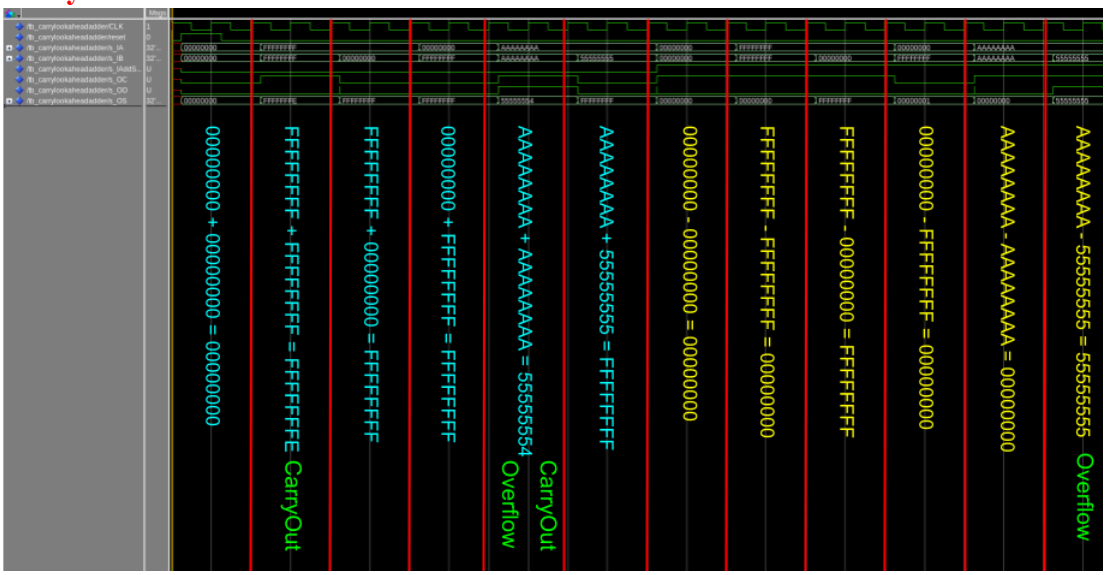
$$\begin{bmatrix} c_1 = g_0 + p_0c_0 \\ c_2 = g_1 + p_1c_1 \\ c_3 = g_2 + p_2c_2 \\ c_4 = g_3 + p_3c_3 \end{bmatrix} \quad \begin{aligned} c_2 &= g_1 + p_1g_0 + p_1p_0c_0 \\ c_3 &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \\ c_4 &= \underbrace{g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0}_{G_0} + \underbrace{p_3p_2p_1p_0}_{P_0}c_0 \end{aligned}$$

j=1

$$\begin{bmatrix} c_5 \\ c_6 \\ c_7 \\ c_8 \end{bmatrix} \quad \begin{aligned} c_8 &= G_1 + P_1c_4 \\ c_{12} &= G_2 + P_2c_8 \\ c_{16} &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0 \end{aligned}$$

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

Carry Lookahead Adder



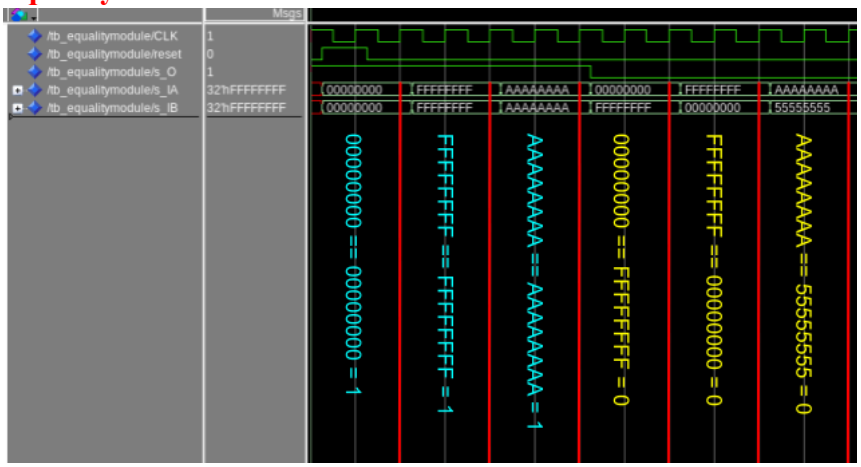
I included 12 of the tests for our Carry Lookahead Adder. The 6 in blue on the left are addition, and the 6 in yellow on the right are subtraction. For both addition and subtraction, we start with both operands as 0 and then both operands as the max value. Then for both addition and subtraction, we use 0 and the max value on either side of the operator. Then for addition, we add all A's with itself to check that the overflow flag will trigger, which it does because negative + negative yielded a positive value. Then for subtraction, we do all A's minus all 5's to check that the overflow triggers again, which it does because a negative minus a positive yielded a positive value.

[Part 2 (c.ii.3)] In your write-up, 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.

I made a simple, functional unit to determine if two values are equal. This module is used for the beq and bne instructions in the ALU. If the two input vectors are completely equal then the output is 1 else it is 0.

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

Equality Module



This model is very simple, so only 6 tests were included. The first three tests on the left in blue pass in two equal inputs, and the output is correctly 1. The last three tests on the right in yellow pass in two non-equal vectors, so the output is correctly 0.

[Part 2 (c.ii.5)] 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.

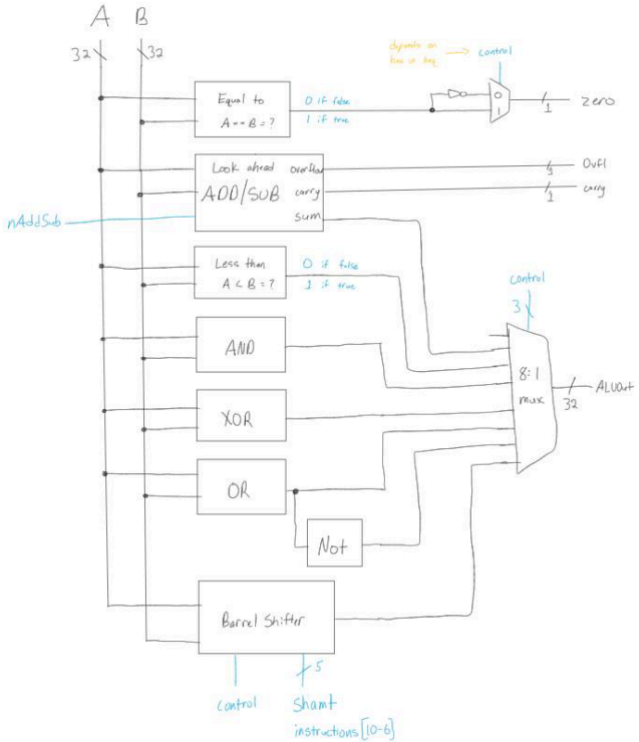
This was a very simple submodule that was created for the barrel shifter. This is because the barrel shifter needs to reverse a vector if it is going to do a left shift instead of a right shift. This functional unit takes a 32-bit input vector and outputs that vector in reverse order. I decided to use a process with a for loop that had a signal assignment statement within to simplify the task.

[Part 2 (c.ii.6)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

Vector Reverser

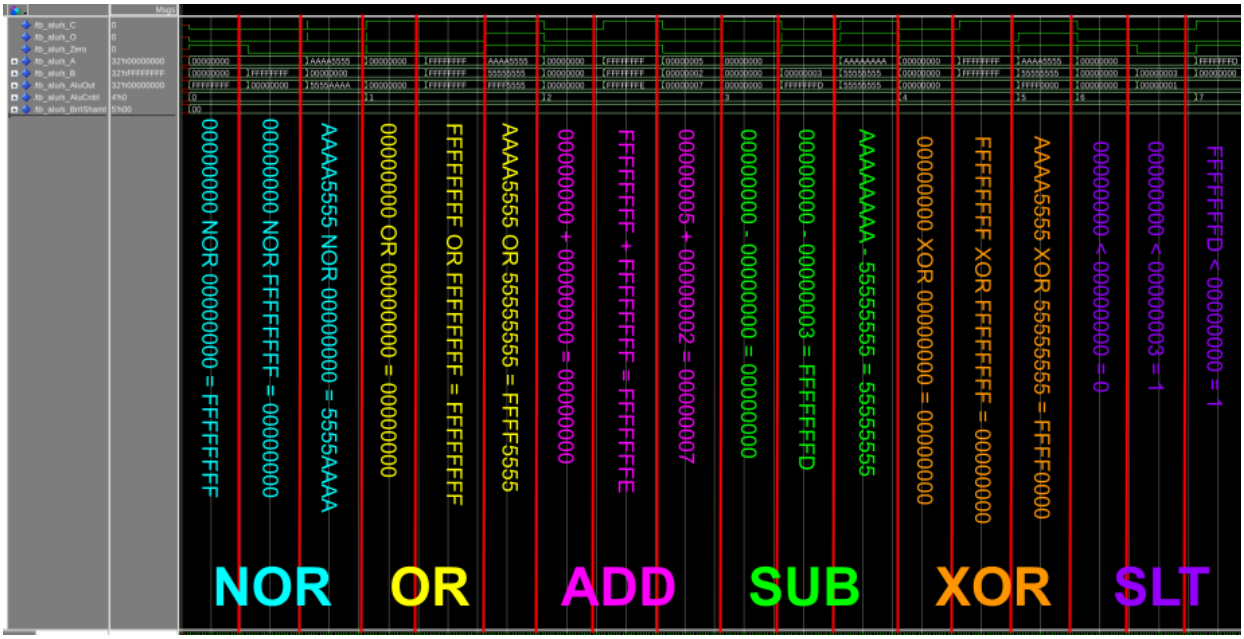
Once again this is a very simple module so there aren't many tests needed to verify the functionality of the unit. The first two tests reverse all 0's and all F's so the input and output are the same because the input is the same forwards and backwards. The next two tests show the vectors of 0000FFFF and FFFF0000 being reversed. The last test is all A's which is 10101010… which reversed is 01010101… which is 55…. These tests demonstrate that the unit works correctly.

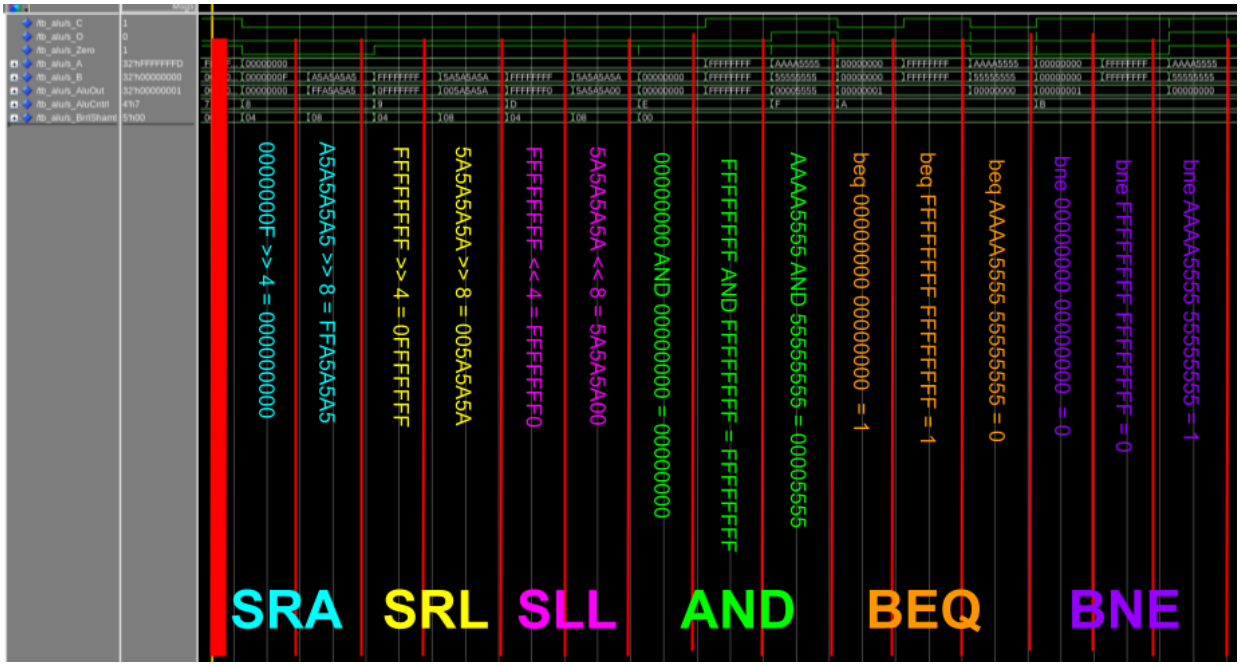
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?



Overflow is calculated by taking the xor of the last carryout and the n-1 carryout. This is because overflow only occurs when positive plus positive gives a negative, negative plus negative gives a positive, positive minus negative gives a negative, or negative minus a positive gives a positive. The zero output of the ALU is calculated for the beq and bne instructions. If the control signals indicate a beq, the zero output is set to 1 if the ALU input A and B are equal and 0 if they are not equal. The opposite is true if the control signals indicate a bne. In this case the zero output is 0 if the two values are equal and 1 if they are not equal. Slt is implemented by using the less than operator in vdhl. When A is less than B then the output of the ALU is set to "00000000000000000000000000000001" else the output is set to "00000000000000000000000000000000".

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





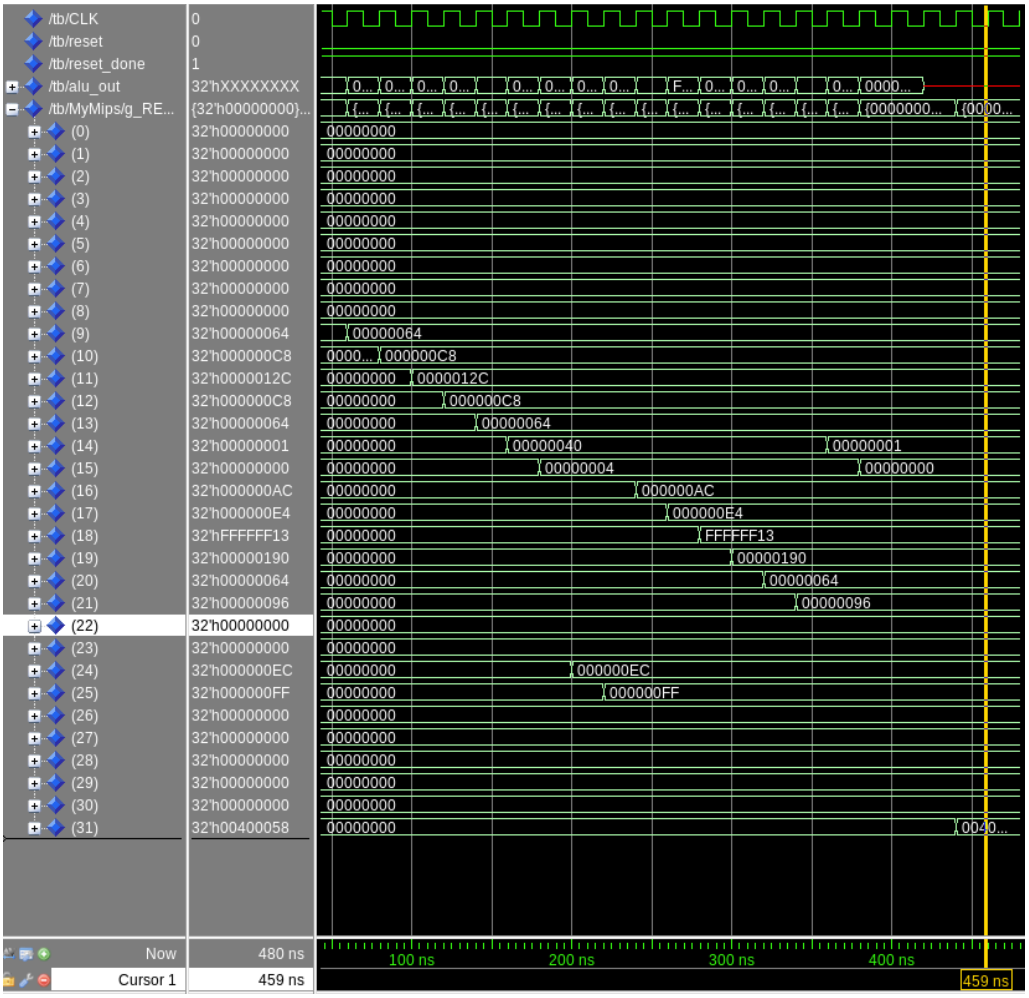
The different ALU operations are selected by the AluCntrl vector. This vector uses the last three bits to select the unit to be used for the operation (barrel shifter, adder, gates, etc.), and the 0 bit is used as a control for things like shift type (logical vs arithmetic) or branch control (beq vs bne). The s_A and s_B signals represent the two input operands. The AluOut represents the Alu output.

[Part 2 (c.viii)] justify why your test plan is comprehensive.

For the ALU test benches, I included tests for the nor, or, add, sub, xor, slt, sra, srl, sll, and, beq, and bne instructions. This was a total of 33 tests to verify that each instruction behaved as expected. For each test I used a signal to assign a relevant A and B 32-bit data input as well as a 5-bit shift amount and a 4-bit Alu control signal. The Alu control signal vector corresponds to what is listed in our control spreadsheet.

[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.

Proj1_base_test.s



Expected final register states:

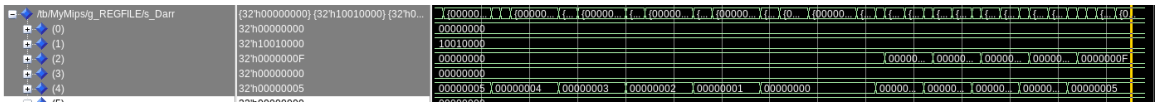
$t0=$8: 0x0
$t1=$9: 0x64
$t2=$10: 0xC8
$t3=$11: 0x12C
$t4=$12: 0xC8
$t5=$13: 0x64

$t6=$14: 0x1
$t7=$15: 0x0
$s0=$16: 0xAC
$s1=$17: 0xE4
$s2=$18: 0xFFFFFF13
$s3=$19: 0x190
$s4=$20: 0x64
$s5=$21: 0x96
$t8=$24: 0xEC
$t9=$25: 0xFF

As can be seen in the waveforms the expected final register states match the actual final register states. For a breakdown of how the actual values are calculated see the comments below the actual program. A screenshot of the program including the comments is included in section 3.A.

Proj1_cf_test.s

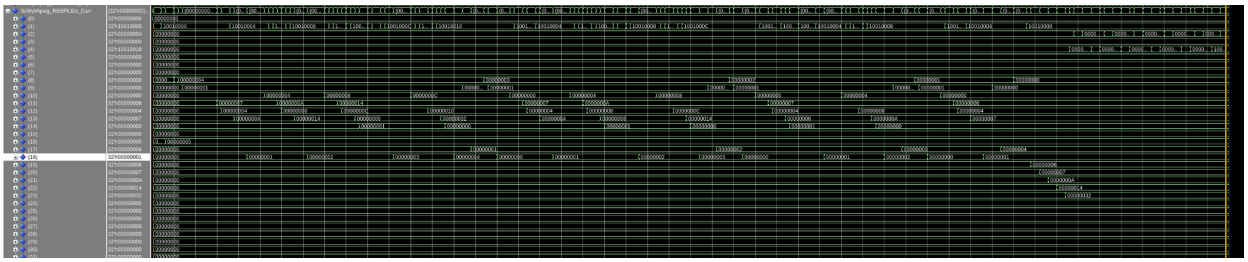
Final State



The final state waveform shows that, as expected, v0 = 15 and a0 = 5. Running the same code in Mars gives the same values, which proves our processor is working as expected.

Proj1_bubblesort.s

Full Waveform



While Difficult to show in waveform screenshots, scrolling through the waveform you can watch the addresses of the temp values get set during our swap function as expected. What a screenshot of the waveforms does show clearly, is that the initial array (7, 10, 20, 6, 50) is set in 21-25, and the final state screenshots display the sorted array (6, 7, 10, 20, 50) as expected.

[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.

```
main:
    # Arithmetic instructions
    add $t0, $t1, $t2       # t0 = t1 + t2 (Initially t1 = 0, t2 = 0, so t0 = 0)
    addi $t1, $zero, 100    # t1 = 100
    addiu $t2, $zero, 200   # t2 = 200
    addu $t3, $t1, $t2      # t3 = t1 + t2 = 100 + 200 = 300

    sub $t4, $t3, $t1       # t4 = t3 - t1 = 300 - 100 = 200
    subu $t5, $t2, $t1      # t5 = t2 - t1 = 200 - 100 = 100

    # Logical instructions
    and $t6, $t1, $t2       # t6 = t1 & t2 = 100 & 200 = 64
    andi $t7, $t1, 15       # t7 = t1 & 15 = 100 & 15 = 4

    or $t8, $t1, $t2        # t8 = t1 | t2 = 100 | 200 = 236
    ori $t9, $t1, 255       # t9 = t1 | 255 = 100 | 255 = 255

    xor $s0, $t1, $t2       # s0 = t1 ^ t2 = 100 ^ 200 = 172
    xori $s1, $t1, 128      # s1 = t1 ^ 128 = 100 ^ 128 = 228

    nor $s2, $t1, $t2       # s2 = ~(t1 | t2) = ~(100 | 200) = -237

    # Shift instructions
    sll $s3, $t1, 2         # s3 = t1 << 2 = 100 << 2 = 400
    srl $s4, $t2, 1         # s4 = t2 >> 1 = 200 >> 1 = 100
    sra $s5, $t3, 1         # s5 = t3 >> 1 (arithmetic) = 300 >> 1 = 150

    # Comparison instructions
    slt $t6, $t1, $t2       # t6 = (t1 < t2) ? 1 : 0 = (100 < 200) ? 1 : 0 = 1
    slti $t7, $t1, 50       # t7 = (t1 < 50) ? 1 : 0 = (100 < 50) ? 1 : 0 = 0

    # Branch/Jump instructions
    beq $t1, $t2, skip      # t1 != t2, so no branch
    bne $t1, $t2, continue  # t1 != t2, so branch to continue


skip:
    j end                   # Jump to end

continue:
    jal dummy_function      # Jump and link to dummy_function
    halt
    j end

dummy_function:
    # Dummy function to demonstrate jal and jr usage
    jr $ra                  # Jump to the return address (continue execution at end)

end:


# Expected final register states:
# $t0 = 0             # t0 = t1 + t2 = 0 + 0 = 0 (initial state)
# $t1 = 100           # t1 = 100 (after addi)
# $t2 = 200           # t2 = 200 (after addiu)
# $t3 = 300           # t3 = t1 + t2 = 100 + 200 = 300
# $t4 = 200           # t4 = t3 - t1 = 300 - 100 = 200
# $t5 = 100           # t5 = t2 - t1 = 200 - 100 = 100
# $t6 = 1             # t6 = (t1 < t2) ? 1 : 0 = 1
# $t7 = 0             # t7 = (t1 < 50) ? 1 : 0 = 0
# $t8 = 236           # t8 = t1 | t2 = 100 | 200 = 236
# $t9 = 255           # t9 = t1 | 255 = 100 | 255 = 255
# $s0 = 172           # s0 = t1 ^ t2 = 100 ^ 200 = 172
# $s1 = 228           # s1 = t1 ^ 128 = 100 ^ 128 = 228
# $s2 = -237          # s2 = ~(t1 | t2) = ~(100 | 200) = -237
# $s3 = 400           # s3 = t1 << 2 = 100 << 2 = 400
# $s4 = 100           # s4 = t2 >> 1 = 200 >> 1 = 100
# $s5 = 150           # s5 = t3 >> 1 (arithmetic) = 300 >> 1 = 150
# $ra = Address after jal (return to end)
```

[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.

```
#
# Test program for mips control flow instructions and has a call depth of at least 5
# beq, bne, j, jal, jr
# The program should jump (in order) init, RUN2, RUN1, RUN4, RUN3, Finish RUN4, halt
# There's a chance 3 and 4 loop forever
#

# data section
.data

.text
.globl main

main:
    li $sp, 0x10011000        # Initialize stack pointer
    addi $a0, $zero, 5        # Set initial value
    jal recursive             # Call recursive function
    addi $t1, $zero, 15       # Set expected result
    bne $v0, $t1, failure     # Check result
    j exit                    # Jump to exit

recursive:
    addi $sp, $sp, -8       # Allocate stack space
    sw $ra, 4($sp)          # Save return address
    sw $a0, 0($sp)          # Save argument n

    slti $t0, $a0, 1            # Check if n < 1
    beq $t0, $zero, decrement   # If n >= 1, recurse
    li $v0, 0                   # Base case: return 0
    j return                    # Jump to return

decrement:
    addi $a0, $a0, -1        # Decrement n
    jal recursive            # Recursive call
    lw $a0, 0($sp)           # Restore n
    add $v0, $v0, $a0        # Return n + recursive(n - 1)

return:
    lw $ra, 4($sp)           # Restore return address
    addi $sp, $sp, 8         # Deallocate stack space
    jr $ra                   # Return to caller

failure:
    li $v0, 1               # Print failure code (1)
    j exit                  # Jump to exit

exit:
    halt
```

[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.

```
.text
.globl main
main:
    # store size
    lw $t1, size

    addi $t2, $0, 0 #i = 0
    addi $t3, $0, 0 #j = 0

for1:
    beq $t2, $t1, exit # for(i = 0; i < n-1;)
    add $t3, $0, $0 # j = 0
    addi $t2, $t2, 1 # (i++)

    j for2 # jump to for2
for2:
    beq $t3, $t1, for1 # for(j = 0; j < n-1;)
    addi $t7, $t3, 1 # j + 1
    sll $s1, $t3, 2 # current j shift left (multiply by 4)
    sll $s2, $t7, 2 # current j + 1shift left (multiply by 4)
    lw $t4, arr($s1) # load content of arr[j]
    lw $t5, arr($s2) # load content of arr[j + 1]
    slt $t6, $t4, $t5 # if (arr[j] > arr[j + 1])

    add $a0, $0, $t3
    addi $v0, $0, 1
    syscall

    add $a0, $0, $t7
    addi $v0, $0, 1
    syscall

    addi $v0, $0, 4
    la $a0 , space
    syscall

    lw $s5, arr
    add $a0, $0, $s5
    addi $v0, $0, 1
    syscall

    addi $s6, $0, 4
    lw $s6, arr($s6)
```

```
    add $a0, $0, $s6
    addi $v0, $0, 1
    syscall

    addi $s7, $0, 8
    lw $s7, arr($s7)
    add $a0, $0, $s7
    addi $v0, $0, 1
    syscall

    addi $t8, $0, 12
    lw $t8, arr($t8)
    add $a0, $0, $t8
    addi $v0, $0, 1
    syscall

    addi $t9, $0, 16
    lw $t9, arr($t9)
    add $a0, $0, $t9
    addi $v0, $0, 1
    syscall

    addi $v0, $0, 4
    la $a0 , space
    syscall

    bne $t6, $0, swap # if t6 = 1, swap
    addi $t3, $t3, 1 # (j++) if no swap
swap:
    add $t7, $0, $t4 # temp = i
    add $t4, $0, $t5 # i = j
    add $t5, $0, $t7 # j = temp
    sw $t4, arr($s1) # store i
    sw $t5, arr($s2) # store j

    #addi $t3, $t3, 1 # (j++) after swap
    j for2 # return to for2
exit:

    halt
```

[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?

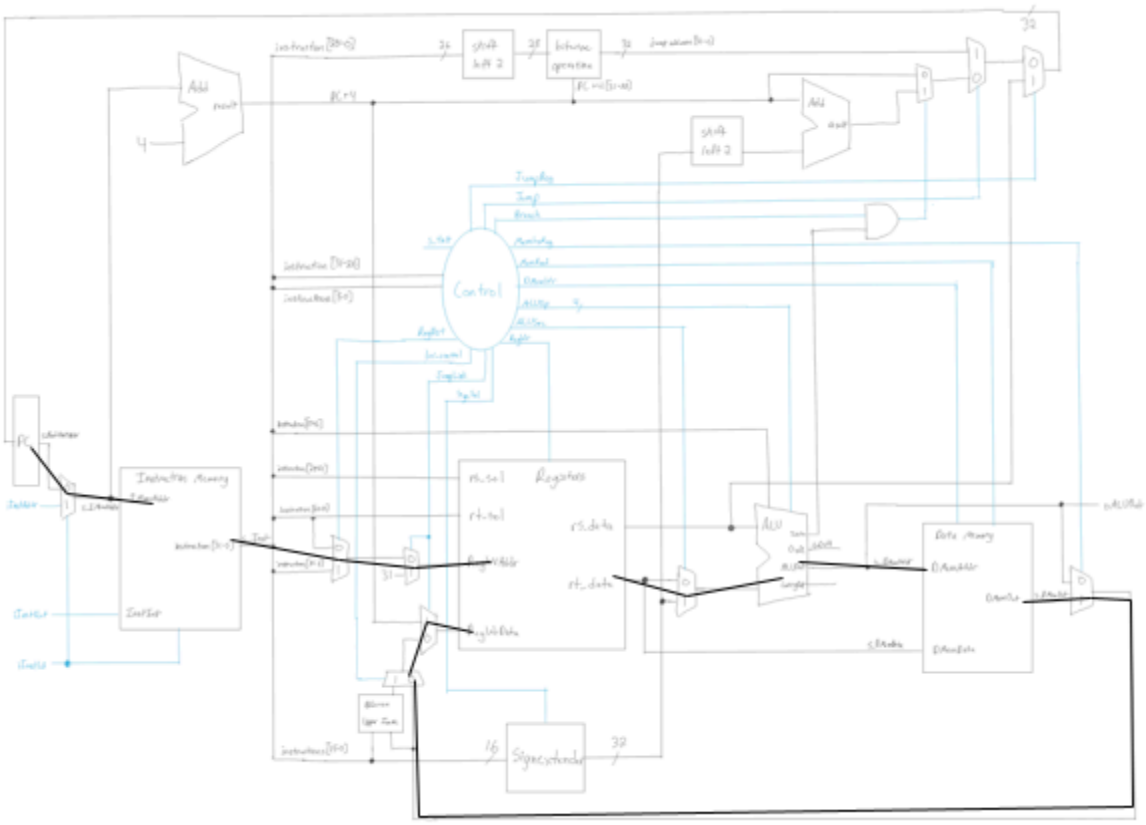
```
#
# CprE 381 toolflow Timing dump
#

FMax: 26.75mhz Clk Constraint: 20.00ns Slack: -17.38ns
```

Max frequency: 26.75 Mhz

Cycle Time: 1/26.75 Mhz = 37.383 nanoseconds

Top-level Schematic Critical Path Diagram (lw):



Critical Path (lw):
I_mem
RegFile (read)
Mux (Alu Src B Conditional)

Alu
Mux (Alu out Conditional)
D_mem
Mux(JumpLink Conditional)
RegFile (write)

Frequency Improvement:

About one eighth of our cycle is spent in the ALU component, making it the slowest component, so focusing on optimizing and speeding up our ALU would greatly improve our frequency. Within the ALU, the component that seemed to take the most time was the barrel shifter. If this component could be optimized, then our cycle time would improve.