CprE 381: Computer Organization and Assembly-Level Programming

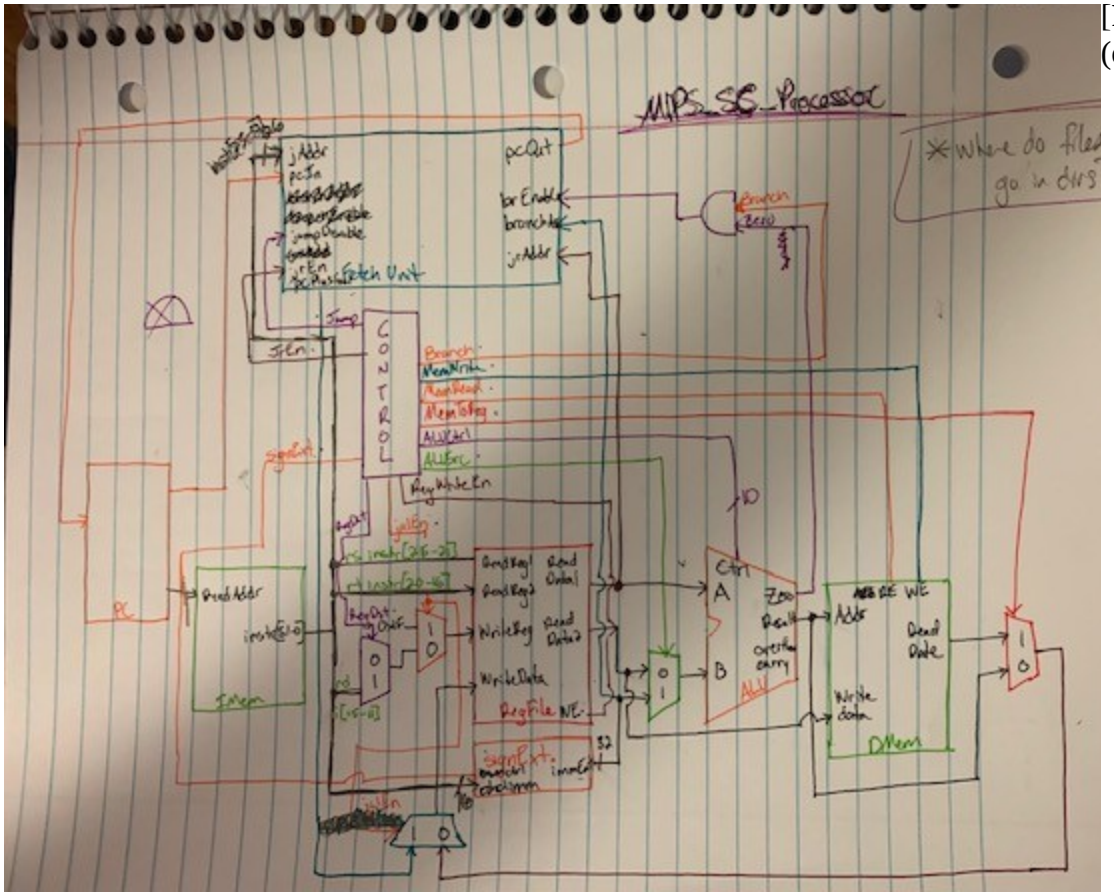
Project Part 1 Report

Team Members: Anna Huggins

Christopher Sell

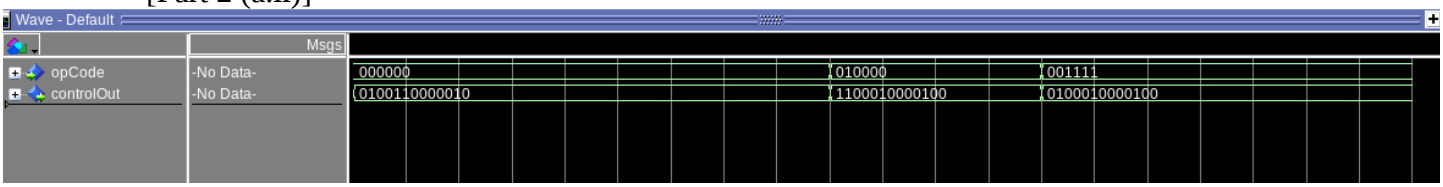
Project Teams Group #: 8-8

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



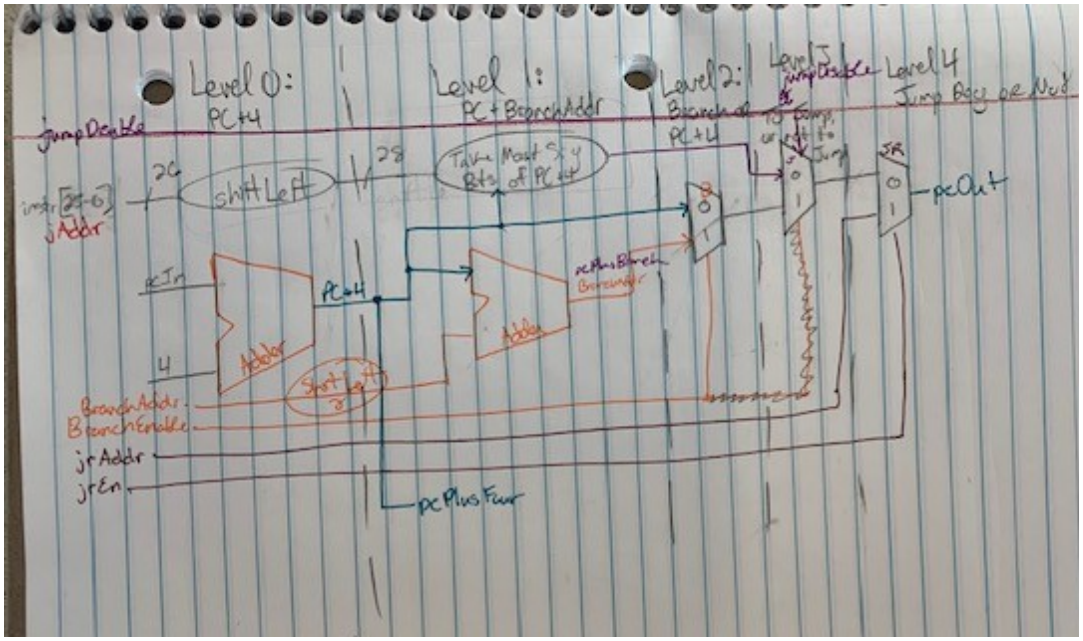
[Part 1 (d)]

[Part 2 (a.ii)]

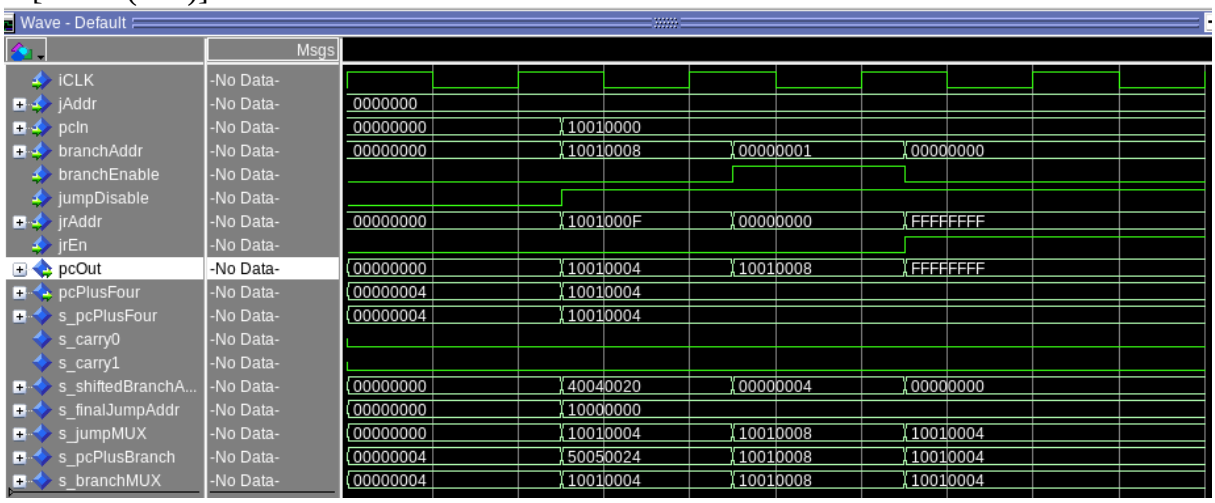


[Part 2 (b.i)] In order to implement the required control flow possibilities we will need to add more control bits to enable branch, jump, and jump.

[Part 2 (b.ii)] We also needed to add jump, branch, and jump reg control signals.



[Part 2 (b.iii)]



-- Test case 0: let pcPlusFour get through
s_pcIn = x"10010000" and pcOut = pcPlusFour = 0x”10010004”:
Other signals are given input values to clearly indicate that the are not setting the output.

```
-- Test case 1: let PCPlusBranch get thru
s_pcIn= x"10010000";
s_branchAddr  = x"00000001", so our shifted branch address is 0x00000004.
s_branchEnable  <= '1';                    -- Enable branch
```

Then, our output for PCPlusFour + our branch address = 0x10010008

```
-- Test case 2: let jReg get through
s_jAddr = "00000000000000000000000000";
s_pcIn = x"10010000";              -- current PC address
s_jrAddr = x"FFFFFFFF";
s_jrEn  = '1';                     -- Enable jump reg
```

Here, we jump to the address stored in the “specified” register, so out PCOut = 0xFFFFFFFF.

[Part 2 (c.i.1)]
srl: shifts the bits contained in a register to the right x times, and fills in the left with 0s.

sra: shifts the bits contained in a register to the right x times and preserves the sign of the data by keeping the most significant bit.

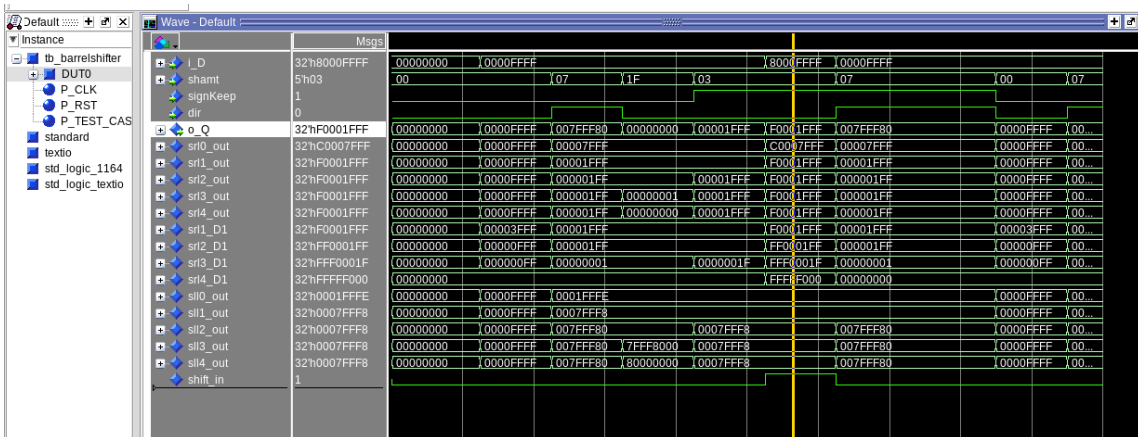
sll: shifts the bits contained in a register to the left x times, and fills the right with 0s.

There is not sla because there would be no purpose for preserving the sign bit, as we are shifting left, and if we were to consider preserving the rightmost bit (the least significant bit), there is no reason to do this.

[Part 2 (c.i.2)]
Our VHDL code implements arithmetic operations by using a signal to keep track of the input’s sign bit and then shifting in the corresponding bit (1 or 0). For logical operations, we simply shift in 0.

[Part 2 (c.i.3)] In order to support left shifting operations, we simply do the same procedure as we did for srl but shift in the opposite direction at the same time, and then use a mux at the end to select which output we want, srl or sll.

[Part 2 (c.i.4)] Our waveform below



corresponds to the following as written below:Case 1: x”0000FFFF” is allowed to pass through the simulation by having shamt = 0.Case 2: x”0000FFFF” is shifted to the left by 7 bits to the left. [sll]
Case 3: x”0000FFFF” is shifted to the right by 31 bits, thus the output is 0. [srl]
Case 4: x”0000FFFF” is shifted to the right by 31 bits and sign is kept, output = 0. [sra]
Case 5: x”8000FFFF” is shifted to the right by 3 bits and sign is preserved. [sra]

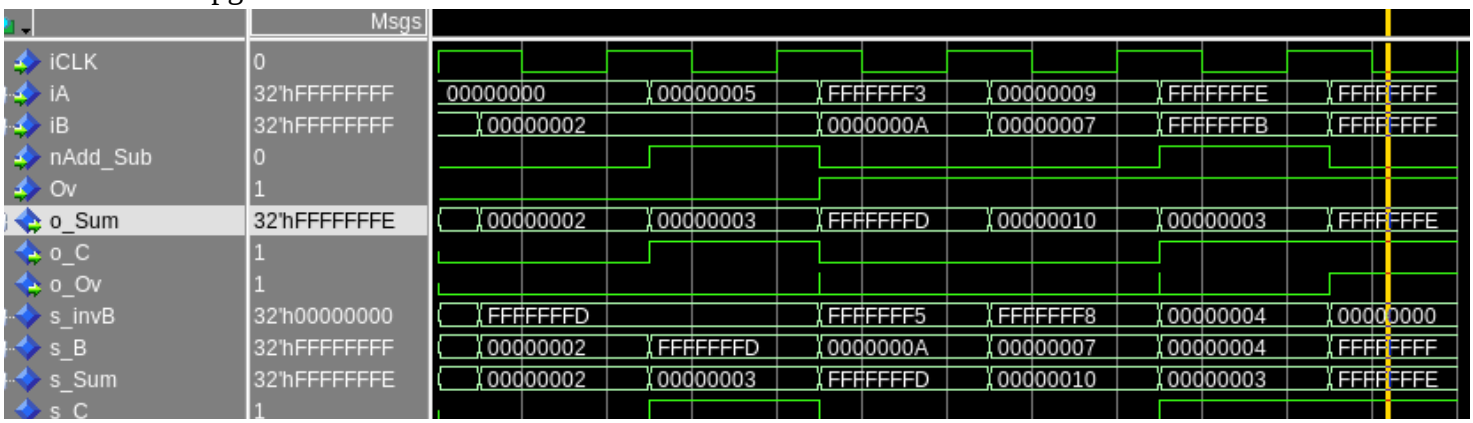
[Part 2 (c.ii.1)]

Our design approach was to analyze what functions still required additional signals in our processor, and then we dove into how they would need to be implemented in our ALU (or other units). Overall, we broke our processor down into some primary components: Fetch Unit, Control Unit, ALU, and then the memory and registers files. One design decision we made was in our ALU, we broke this component into 4 primary blocks: adderSub, barrelShifter, logicalOpsUnit, and branchUnit. Each of these units execute specific intructions.

Adder Sub: all add/sub instructions (sign, unsigned, etc)
Barrel Shifter: srl, sra, sll
Logical Ops Unit: and, xor, or, nor, not, slt, lui, repl.qb
Branch Unit: bne, beq

For the ALU, we first divided instructions into their units, and then considered how previously existing components would need to be updated (adder sub) and then drew diagrams for the new components. After this, we converted the drawings into VHDL and created test benches for each one. After each unit was tested, they were integrated into upgradedALU.

[Part 2 (c.ii.2)]
Upgraded Adder/Subtractor:



– Unsigned Operations
-- Test case 0: Sum <= A + B = 0 + 1 (no overflow, uns.)
o_Sum = 0 + 2 = 2; Overflow = 0 because we aren’t detecting it. Carry = 0.

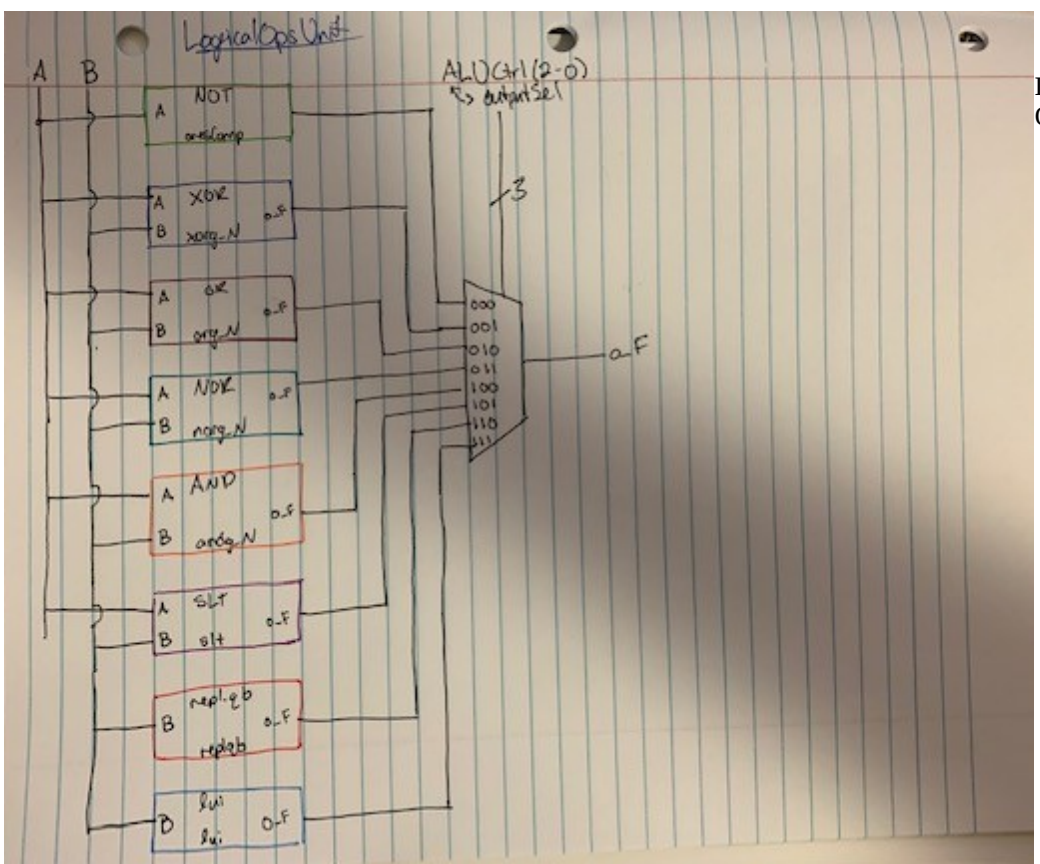
-- Test case 1: Sum <= A - B = 2-1 (no overflow, uns.)
o_Sum = 5 – 2 = 3; Overflow = 0 because we aren't detecting it here. Carry = 1 because we are subtracting.

-- Signed Test Cases
-- Test case 3: -13 + 10 (no overflow, signed)
o_Sum = -13 + 10 = 3 = 0xFFFFFFFD
Overflow = 0, no overflow detected. Carry = 0.

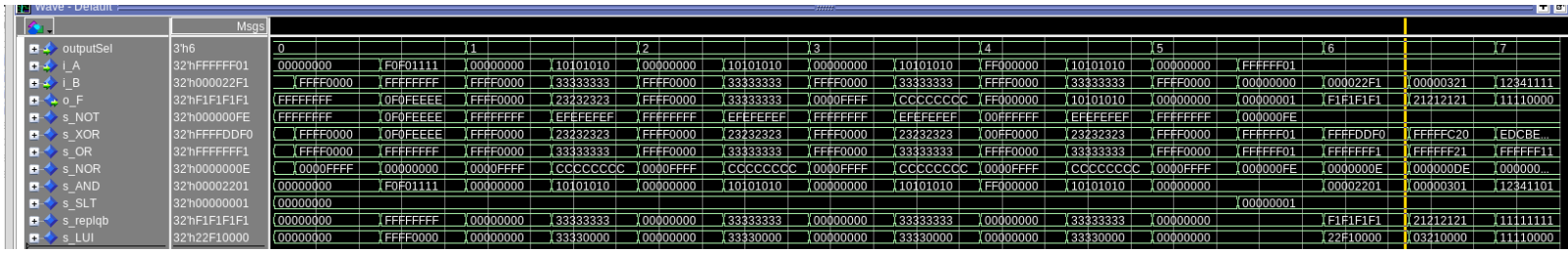
-- Test case 4: 9+7 (no overflow, signed)
o_Sum = 9 + 7 = 13. Overflow = 0, no overflow detected. Carry = 0.

-- Test case 5: -2-(-5) (no overflow, signed, 32-bit)
o_Sum = -2 - (-5) = 3; Overflow = 0, none detected. Carry = 1 because we are subtracting.

-- Test case 6: -1 + -1 (overflow, signed, 32-bit)
o_Sum = -1 + -1 = -2; Overflow = 1 because the sign bit is 1, and so it the carry.Carry = 1

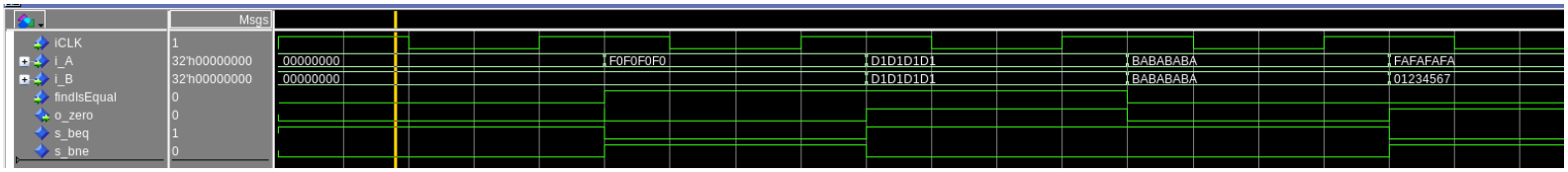


Logical Ops Unit:

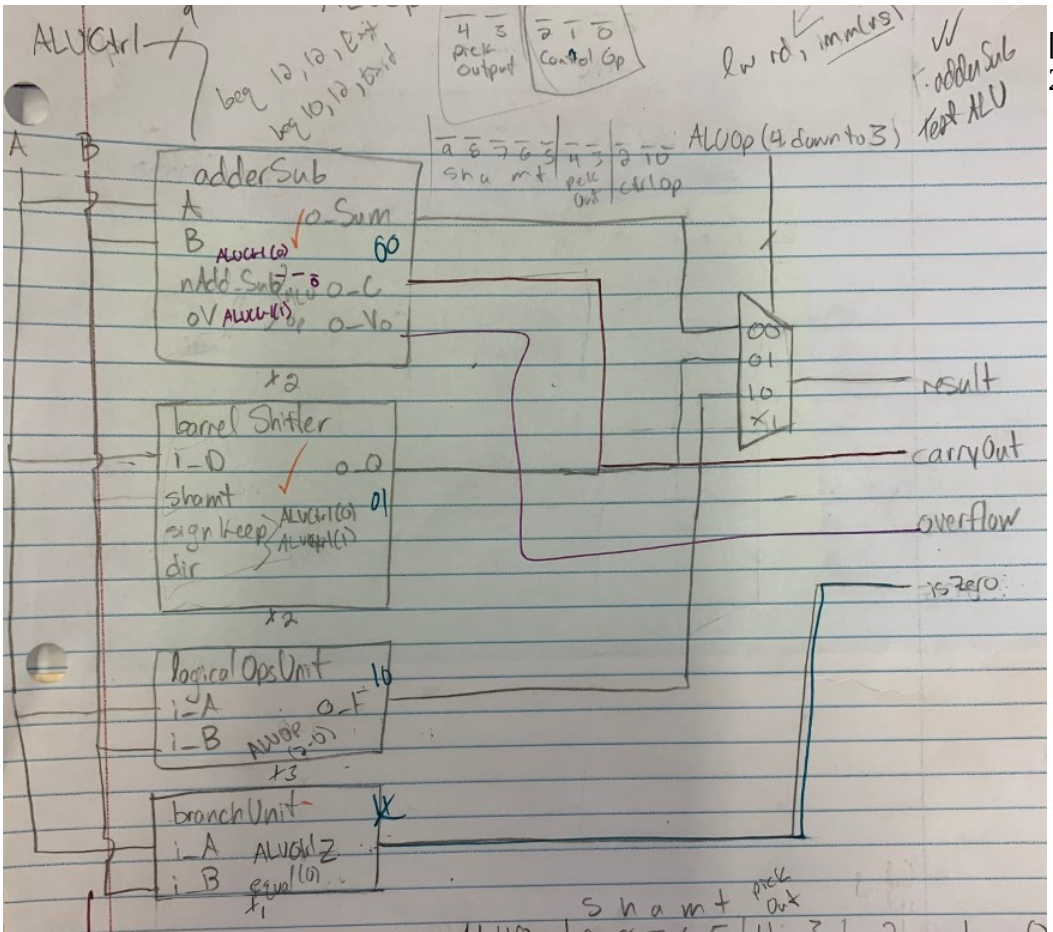


The above waveform corresponds to the test benches written in tb_logicalOpsUnit.

Branch Unit:



The above waveform corresponds to the test benches written in tb_branchUnit.



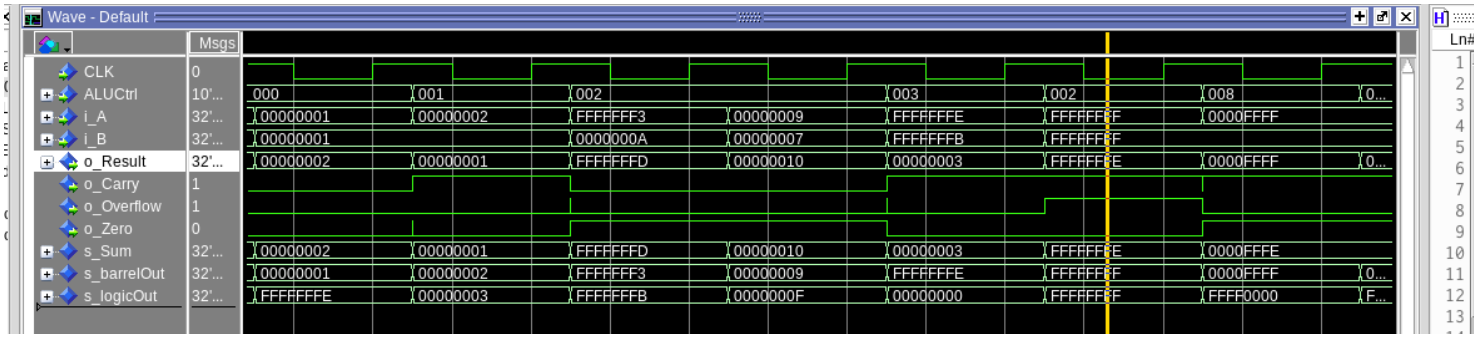
[Part 2

(c.iii)] Overflow is calculated in adderSub by ANDing o_C with oV. The oV bit determines if we are performing a signed operation or not. If oV = 1 and o_C = 1, we have overflow. In all other cases, there is no overflow.

Zero is calculated from the branchUnit by determining if the bne or beq operation is successful. For beq, if A and B are equal, then Zero = 1, otherwise Zero = 0. For bne, if A and B are not equal, Zero = 1 otherwise Zero = 0.

Slt is implemented in the logical operations unit by determining if A < B, and then setting the output o_F = to 1 when A < B and 0 otherwise.

[Part 2 (c.v)]



Add/Sub-- Unsigned Cases

-- Test case 0: addu: Sum <= A + B (no overflow, unsigned)
1 + 1 = 2; nAdd_Sub = 0, oV = 0 → don't detect overflow
o_Overflow = 0

-- Test case 1: subu: Sum <= A - B = 2 - 1 (no overflow, unsigned)
2 – 1 = 1; nAdd_Sub = 0, oV = 0 → don't detect overflow
o_Overflow = 0

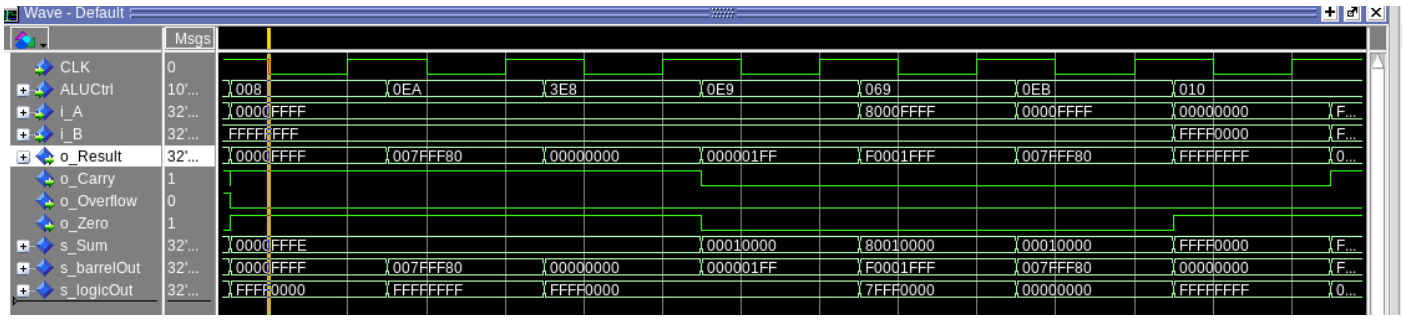
-- **Signed Test Cases**

-- Test case 3: add: Sum <= A + B (no overflow, signed)
-13 + 10 = 3 ; nAdd_Sub = 0, oV = 1 → detect overflow
o_Overflow = 0

-- Test case 4: add: Sum <= A + B 9+7 (no overflow, signed)
9 + 7 = 13; nAdd_Sub = 0, oV = 1 → detect overflow
o_Overflow = 0

-- Test case 5: sub: Sum <= A – B -2 - (-5) (no overflow, signed, 32-bit)
-2 - (-5) = 3 ; nAdd_Sub = 0, oV = 1 → detect overflow
o_Overflow = 0

-- Test case 6: add: -1 + -1 (overflow, signed, 32-bit)
-1 + -1 = -2 ; nAdd_Sub = 0, oV = 1
o_Overflow = 1



Barrel Shifter-- Test case 7: Let initial value pass through

A = x0000FFFF is shifted by 0, so it just passes through. Matches output o_Result.

-- Test case 8: Test sll

A = x0000FFFF is shifted to the left by 7.

-- Expect: oQ = 0x0007FF80

-- Test case 9: Test srl

A = x0000FFFF is shifted to the right by 31

-- Expect: oQ = 0x00000000

-- Test case 10: Test sra

A = 0000FFFF shifted to the right by 7, signKeep is set to 1. Doesn't effect output because signed bit is 0.

-- Expect: oQ = 0x000001FF

-- Test case 11: Test sra when sign bit = 1

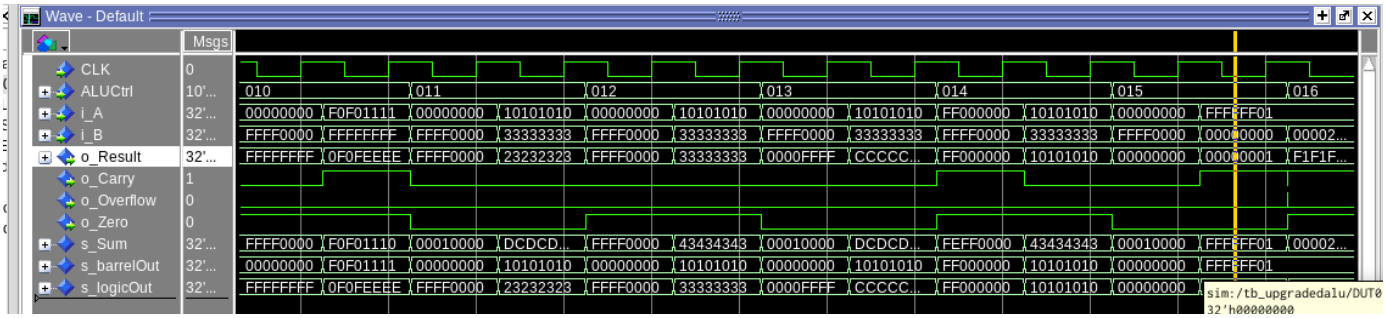
A = 8000FFFF shifted to the right by 7, signKeep is set to 1. Changes output because sign bit is 1.

-- Expect: oQ = 0xf0001fff

-- Test case 12: Test sll (when signKeep = 1)

A = 0000FFFF shifted to the left by 7, signKeep is set to 1. Doesn't effect output because we are doing sll.

-- Expect: oQ = 0x0007FF80



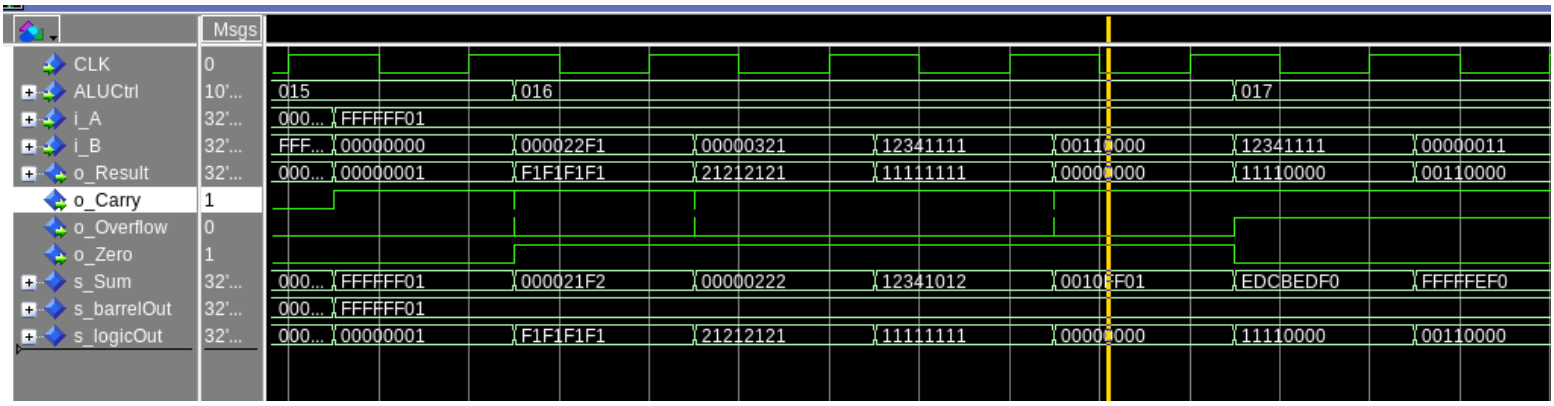
Logical Ops Unit

-- NOT Tests ----------

```
    -- Test case 13: NOT Gate
o_Result = !x"00000000"  = 0xFFFFFFFF

    -- Test case 1:4 NOT Gate
    ALUCtrl    <= "0000010000"; -- Select NOT Output
   o_Result = i_A  !x"F0F01111" = x0F0FEEEE

-- XOR Tests -----------------------------------------------------------------
    -- Test case 15: XOR Gate
    ALUCtrl    <= "0000010001"; -- Select XOR Output
   x"00000000" XOR x"FFFF0000" = 0xFFFF0000

    -- Test case 16: XOR Gate
    ALUCtrl    <= "0000010001"; -- Select XOR Output
 x"10101010" XOR x"33333333" = 0x23232323

-- OR Tests ------------------------------------------------------------------
    -- Test case 17: OR Gate
    ALUCtrl    <= "0000010010"; -- Select OR Output
 x"00000000" OR x"FFFF0000" = 0xFFFF0000

    -- Test case 18: OR Gate
    ALUCtrl    <= "0000010010"; -- Select OR Output
  x"10101010" OR x"33333333" = 0x33333333

-- NOR Tests -----------------------------------------------------------------
    -- Test case 19: NOR Gate
    ALUCtrl    <= "0000010011"; -- Select NOR Output
     x"00000000" NOR' x"FFFF0000" = 0x0000ffff

    -- Test case 20: NOR Gate
    ALUCtrl    <= "0000010011"; -- Select NOR Output
    x"10101010" NOR x"33333333" = 0xCCCCCCCC

-- AND Tests -----------------------------------------------------------------
    -- Test case 21: AND Gate
    ALUCtrl    <= "0000010100"; -- Select AND Output
    x"FF000000" AND x"FFFF0000" = 0xFF000000

    -- Test case 22: AND Gate
    ALUCtrl    <= "0000010100"; -- Select AND Output
    x"10101010"  AND x"33333333"  = 0x10101010

-- SLT Tests -----------------------------------------------------------------
    -- Test case 23: SLT
    ALUCtrl    <= "0000010101"; -- Select SLT Output
```

x"00000000" < x"FFFF0000" = false, so o_Result = 0x00000000

-- Test case 24: SLT
ALUCtrl <= "0000010101"; -- Select SLT Output
x"FFFFFF01" < x"00000000" = true, so o_Result = 0x00000001



-- Repl.qb Tests ----------
-- Test case 25: repl.qb
ALUCtrl <= "0000010110"; -- Select repl.qb Output
o_Result = i_B(8 downto 0) &… &… & … (i_B = x"000022F1") = 0xF1F1F1F1

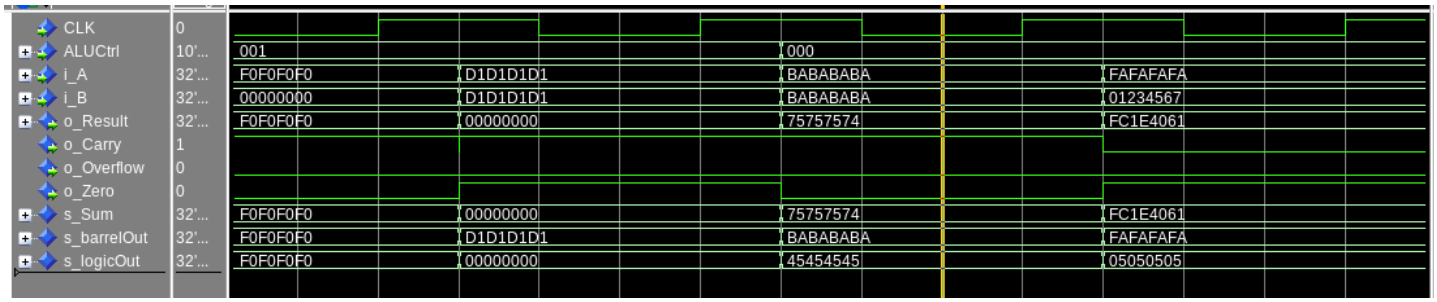
-- Test case 26: repl.qb
ALUCtrl <= "0000010110"; -- Select repl.qb Output
o_Result = i_B(8 downto 0) &… &… & … (i_B = x"00000321") = 0x21212121

-- Test case 27: Entire word has data
ALUCtrl <= "0000010110"; -- Select repl.qb Output
o_Result = i_B(8 downto 0) &… &… & … (i_B = x"12341111") = 0x11111111

-- Test case 28: Entire word has data
ALUCtrl <= "0000010110"; -- Select repl.qb Output
o_Result = i_B(8 downto 0) &… &… & … (i_B = x"00110000") = 0x00000000

-- Lui Tests ----------
-- Test case 29: Entire word has data
ALUCtrl <= "0000010111";
o_Result = i_B(15 downto 0) & 0x0000 (i_B = x"12341111") = 0x12340000

-- Test case 30: Entire word has data
ALUCtrl <= "0000010111";
o_Result = i_B(15 downto 0) & 0x0000 (i_B = x"00000011") = 0x00110000



-- Test case 31: Everything = 0
-- iA = 0x00000000, iB = 0x00000000, findIsEqual = 1
-- Expect: o_zero = 0

-- Test case 32: Testing beq, FAILS
-- iA = 0xF0F0F0F0, iB = 0x00000000, findIsEqual = 1
ALUCtrl <= "0000000001"; -- 0 = bne, 1 = beq
(x"F0F0F0F0" == x"00000000") = false, then o_Zero = 0

-- Test case 33: Testing beq, PASSES
-- iA = 0xD1D1D1D1, iB = 0xD1D1D1D1, findIsEqual = 1
ALUCtrl <= "0000000001"; -- 0 = bne, 1 = beq
(x"D1D1D1D1" == x"D1D1D1D1") = true, then o_Zero = 1

-- Test case 34: Testing bne, FAILS
-- iA = 0xBABABABA, iB = 0xBABABABA, findIsEqual = 0
ALUCtrl <= "0000000000"; -- 0 = bne, 1 = beq
(x"BABABABA" != x"BABABABA") = false, then o_Zero = 0

-- Test case 35: Testing bne, PASSES
-- iA = 0xFAFAFAFA, iB = 0x01234567, findIsEqual = 0
ALUCtrl <= "0000000000"; -- 0 = bne, 1 = beq
(x"FAFAFAFA" != x"01234567") = true, then o_Zero = 1

[Part 2 (c.viii)] This can be indicated in the previous section's waveforms and testbench explanation for the upgradedALU unit. Our test plan is comprehensive because it does multiple test cases for each function, especially in the case that all of these functions are implemented into one unit.

[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.

[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.

[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.

[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?