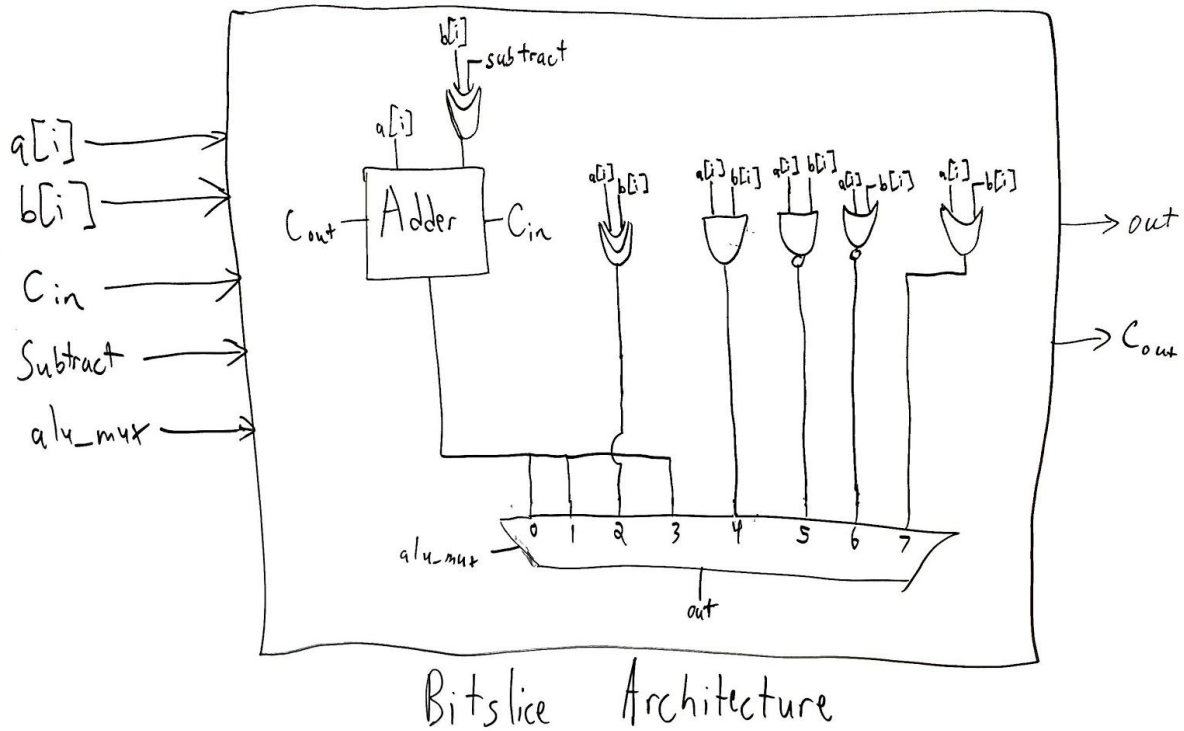
Coleman, Nick, & Yichen
10/03/2018

Lab 1

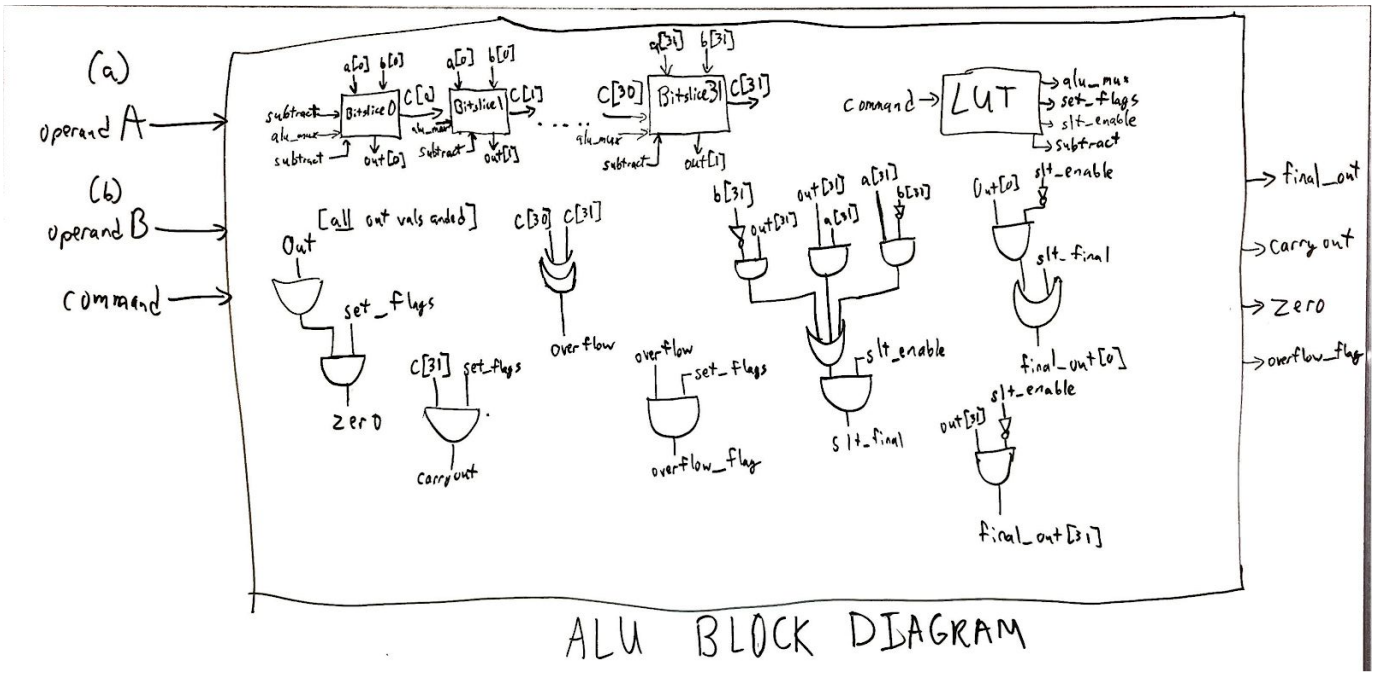
Section 1. Implementation:

We decided to use a “bitslice and LUT control logic” implementation for our ALU, as suggested in the lab document. This implementation begins with a single bitslice unit:



In this unit, we calculate the sum of two input bits; a[i], as well as b[i] XOR’d with the subtract flag for the bitslice (if we’re subtracting b from a, we need to take the complement of b). If the mode is SLT, we subtract b from a as well (as this lets us determine whether a is less than b). We also take the XOR, AND, NAND, NOR, and OR of a[i] and b[i]. All of these outputs are fed into the inputs of a mux (which has 8 inputs despite there only being 5 distinct outputs of the module, as we needed a three bit select signal for 5 distinct outputs, so we decided to use the full 8 possible options with 3 select bits). The bitslice then outputs the output of the mux as well as the carry out of the addition.

From there, we wire bitslices together in the full ALU:



ALU BLOCK DIAGRAM

The ALU starts with 32 bitslice units wired together. They follow the general pattern of having taking in a[i], b[i], alu_mux, subtract, and the carry from the previous bitslice (except for the first unit, which takes in the subtract flag as its carry in, to account for 2's complement subtraction requiring you to add 1). This gives us the output expression, which might need further processing.

Alu_code and subtract for the bitslice units are determined by the LUT, which takes in a command and outputs alu_mux (which is just the same code, for the bitslice muxes), the subtract flag for the bitslices, as well as set_flags (if addition or subtraction is occurring) and an slt_enable flag (if the result has to be zeroed for the slt output).

There's also a few pieces of additional processing circuitry:

- All of the output bits are OR'd together, then AND'd with set_flags, to determine whether to raise the zero flag output.
- The final carryout is AND'd with set_flags, to determine whether to raise the carryout output flag
- Overflow is determined by XORing the last and second to last carry bits, and that is AND'd with set_flags to determine whether to raise the overflow flag
- Based on the input signs and the sign of the output result, we determine whether a was less than b. Then, if slt_enable is high, we zero all but the least significant bit, then set the least significant bit to the result of the slt comparison. If slt_enable is low, we simply output the bitslice results.

Commands

In order to test our alu in verilog, the following commands should be run:
iverilog -o alu alu.t.v
./alu

This will cause our alu to become an executable that can be run repeatedly. You can analyze the individual components by importing it into gtkwave.

Section 2. Test Results:

To test our implementation of the ALU, we picked a number of representative cases for each operations. The test branch will automatically check if the results of the operations and print out an error message if a problem has been detected.

1.ADD:

The selected test cases for ADD operation involves addition of two 0s, two positive numbers, two negative numbers, and one positive and one negative number. Within these subcases, we make sure there are cases that will trigger every possible flags to check the correctness of our implementation for those flags.

32 Bit ADD tests

Control	A	B	R	COut	OFL	ZERO	Cout Exp	OFL Exp	ZERO Exp
000	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	0	0	1	0	0	1
000	00000001001000110100010101100111	00001111111111111111111111111111	00010001001000110100010101100110	0	0	0	0	0	0
000	01111111111111111111111111111111	01111111111111111111111111111111	11111111111111111111111111111110	0	1	0	0	1	0
000	11111111111111111111111111111111	11111111111111111111111111111111	11111111111111111111111111111110	1	0	0	1	0	0
000	10010000000000000000000000000000	10000000000000000000000000000000	00010000000000000000000000000000	1	1	0	1	1	0
000	10000001001000110100010101100111	00010010001101000101011001111000	10010011010101111001101111011111	0	0	0	0	0	0
000	00000000000000000000000000001111	11111111111111111111111111110001	00000000000000000000000000000000	1	0	1	1	0	1
000	11100001001000110100010101100111	01110010001101000101011001111000	01010011010101111001101111011111	1	0	0	1	0	0

Truth table for ADD operation

2.SUB:

The selected test cases for SUB operation involves subtraction of two positive numbers, two negative numbers, and one positive and one negative number. Within these subcases, we make sure there are cases that will trigger every possible flags to check the correctness of our implementation for those flags.

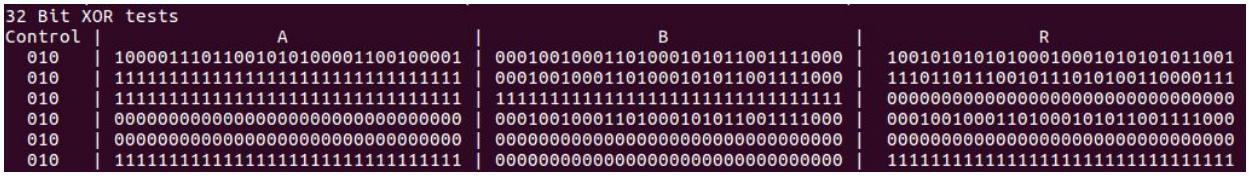
32 Bit SUB tests

Control	A	B	R	COut	OFL	ZERO	Cout Exp	OFL Exp	ZERO Exp
001	00010010001101000101011001111000	00010010001101000101011001111000	00000000000000000000000000000000	1	0	1	1	0	1
001	01111111111111111111111111111111	01110001001000110100010101100111	00001110110111001011101010011000	1	0	0	1	0	0
001	01110001001000110100010101100111	01111111111111111111111111111111	11110001001000110100010101101000	0	0	0	0	0	0
001	11111111111111111111111111111111	11110001001000110100010101100111	00001110110111001011101010011000	1	0	0	1	0	0
001	11110001001000110100010101100111	11111111111111111111111111111111	11110001001000110100010101101000	0	0	0	0	0	0
001	01110001001000110100010101100111	11110001001000110100010101100111	10000000000000000000000000000000	0	1	0	0	1	0
001	00110111011001010100001100100001	11100001001000110100010101100111	01010110010000011111110110111010	0	0	0	0	0	0
001	11100111011001010100001100100001	00110001001000110100010101100111	10110110010000011111110110111010	1	0	0	1	0	0
001	10100111011001010100001100100001	01110001001000110100010101100111	00110110010000011111110110111010	1	1	0	1	1	0

Truth table for SUB operation

3.XOR:

The selected test cases for XOR operation involves XOR two non-typical numbers, and XOR with all 0 or all 1 cases to check the correctness of our ALU implementation under extreme cases.



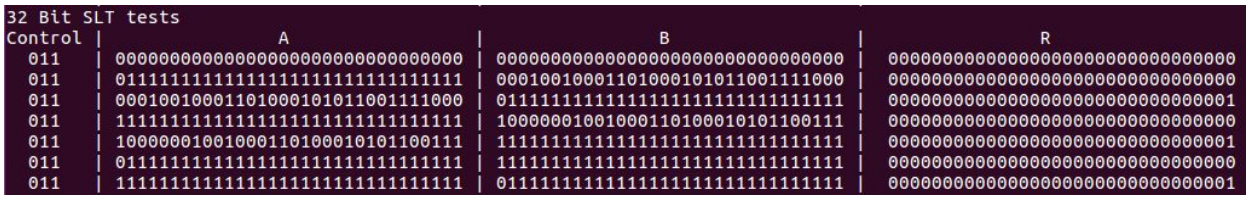
32 Bit XOR tests

Control	A	B	R
010	10000111011001010100001100100001	00010010001101000101011001111000	10010101010100010001010101011001
010	11111111111111111111111111111111	00010010001101000101011001111000	11101101110010111010100110000111
010	11111111111111111111111111111111	11111111111111111111111111111111	00000000000000000000000000000000
010	00000000000000000000000000000000	00010010001101000101011001111000	00010010001101000101011001111000
010	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
010	11111111111111111111111111111111	00000000000000000000000000000000	11111111111111111111111111111111

Truth table for XOR operation

4.SLT:

The selected test cases for SLT operations involves comparison between 0's, two positive numbers, two negative numbers and one positive and one negative numbers. We believe these should exhaust all possible ways of making comparisons between integers.



32 Bit SLT tests

Control	A	B	R
011	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
011	01111111111111111111111111111111	00010010001101000101011001111000	00000000000000000000000000000000
011	00010010001101000101011001111000	01111111111111111111111111111111	00000000000000000000000000000001
011	11111111111111111111111111111111	10000001001000110100010101100111	00000000000000000000000000000000
011	10000001001000110100010101100111	11111111111111111111111111111111	00000000000000000000000000000001
011	01111111111111111111111111111111	11111111111111111111111111111111	00000000000000000000000000000000
011	11111111111111111111111111111111	01111111111111111111111111111111	00000000000000000000000000000001

Truth table for SLT operation

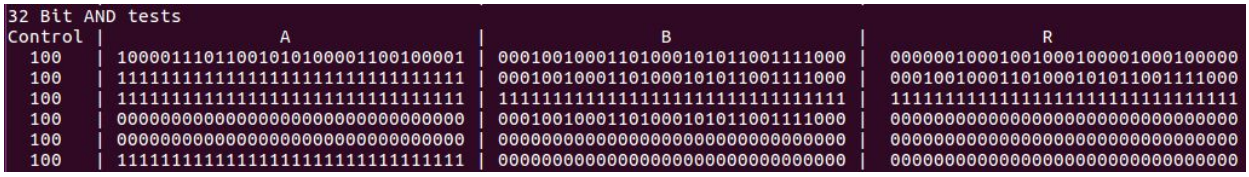
SLT Problems Caught

Our SLT tests caught that we were having issues with the least significant digit in the SLT tests. After analysis, this was determined to be because we didn't check to make sure that STL was enabled based on what test we were completing at the time.

Our SLT tests caught a second error, where our truth table which we had based the SLT results off of had been wrong. We were able to correct this, which resolved all of our SLT problems.

5.AND:

The selected test cases for AND operation involves AND two non-typical numbers, and AND with all 0 or all 1 cases to check the correctness of our ALU implementation under extreme cases.



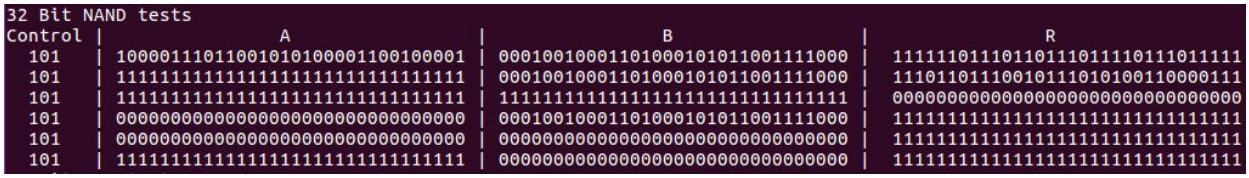
32 Bit AND tests

Control	A	B	R
100	10000111011001010100001100100001	00010010001101000101011001111000	00000010001001000100001000100000
100	11111111111111111111111111111111	00010010001101000101011001111000	00010010001101000101011001111000
100	11111111111111111111111111111111	11111111111111111111111111111111	11111111111111111111111111111111
100	00000000000000000000000000000000	00010010001101000101011001111000	00000000000000000000000000000000
100	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
100	11111111111111111111111111111111	00000000000000000000000000000000	00000000000000000000000000000000

Truth table for AND operation

6.NAND:

The selected test cases for NAND operation involves NAND two non-typical numbers, and NAND with all 0 or all 1 cases to check the correctness of our ALU implementation under extreme cases.

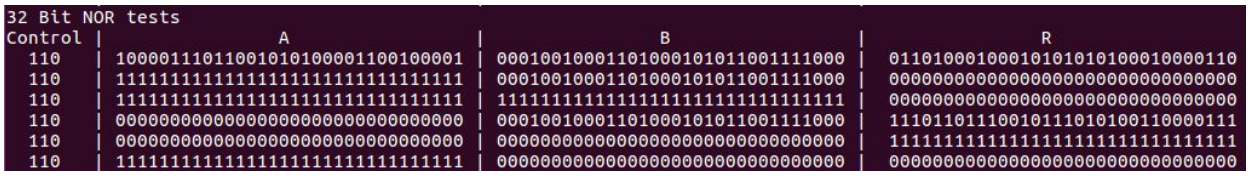


```
32 Bit NAND tests
Control |                A                |                B                |                R
  101   | 10000111011001010100001100100001 | 00010010001101000101011001111000 | 11111101110110111011110111011111
  101   | 11111111111111111111111111111111 | 00010010001101000101011001111000 | 11101101110010111010100110000111
  101   | 11111111111111111111111111111111 | 11111111111111111111111111111111 | 00000000000000000000000000000000
  101   | 00000000000000000000000000000000 | 00010010001101000101011001111000 | 11111111111111111111111111111111
  101   | 00000000000000000000000000000000 | 00000000000000000000000000000000 | 11111111111111111111111111111111
  101   | 11111111111111111111111111111111 | 00000000000000000000000000000000 | 11111111111111111111111111111111
```

Truth table for NAND operation

7.NOR:

The selected test cases for NOR operation involves NOR two non-typical numbers, and NOR with all 0 or all 1 cases to check the correctness of our ALU implementation under extreme cases.

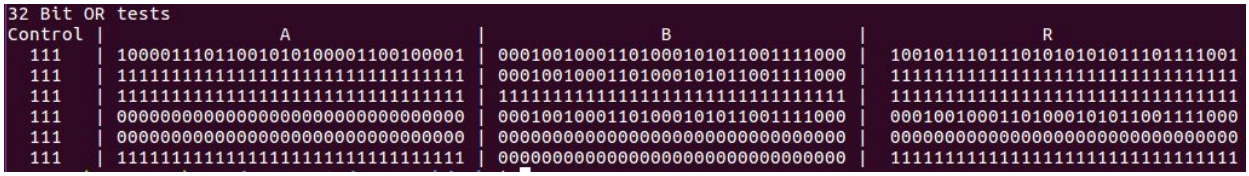


```
32 Bit NOR tests
Control |                A                |                B                |                R
  110   | 10000111011001010100001100100001 | 00010010001101000101011001111000 | 01101000100010101010100010000110
  110   | 11111111111111111111111111111111 | 00010010001101000101011001111000 | 00000000000000000000000000000000
  110   | 11111111111111111111111111111111 | 11111111111111111111111111111111 | 00000000000000000000000000000000
  110   | 00000000000000000000000000000000 | 00010010001101000101011001111000 | 11101101110010111010100110000111
  110   | 00000000000000000000000000000000 | 00000000000000000000000000000000 | 11111111111111111111111111111111
  110   | 11111111111111111111111111111111 | 00000000000000000000000000000000 | 00000000000000000000000000000000
```

Truth table for NOR operation

8.OR:

The selected test cases for OR operation involves OR two non-typical numbers, and OR with all 0 or all 1 cases to check the correctness of our ALU implementation under extreme cases.



```
32 Bit OR tests
Control |                A                |                B                |                R
  111   | 10000111011001010100001100100001 | 00010010001101000101011001111000 | 10010111011101010101011101111001
  111   | 11111111111111111111111111111111 | 00010010001101000101011001111000 | 11111111111111111111111111111111
  111   | 11111111111111111111111111111111 | 11111111111111111111111111111111 | 11111111111111111111111111111111
  111   | 00000000000000000000000000000000 | 00010010001101000101011001111000 | 00010010001101000101011001111000
  111   | 00000000000000000000000000000000 | 00000000000000000000000000000000 | 00000000000000000000000000000000
  111   | 11111111111111111111111111111111 | 00000000000000000000000000000000 | 11111111111111111111111111111111
```

Truth table for OR operation

Miscellaneous Errors:

We found errors that left us scratching out head because we were getting the completely wrong result sometimes. After careful analysis, it was determined that the issue was due to the multiplexing in the bitslice, where we had incorrectly labelled some of the cases. This quickly remedied most of our problems.

In summary, we felt that the selected test cases well represented the spectrum of all possible inputs and allowed us to catch a few errors in our implementation. Because of this, we didn't add any additional test cases to our test bench.

Section 3: Timing Analysis:

Pre-note: All times are in "time units," and as was agreed upon the predetermined estimate for each transistor adds 10 time units to a gate.

The timing analysis worst-case scenarios are broken down by operation for each of the possible operands. It is important to note that in the theoretical, it is unknown how the LUT was synthesized, and therefore we cannot be sure of the time delay. A reasonable time delay to assume is 70; 30 deriving from a 3-size NAND gate for a given state, 10 for an inverter to get an AND gate, and 30 for new values to propagate through another AND gate size 2. The overall worst-case timing of the ALU is as follows:
The worst-case is for SLT, where in order to get the correct output we send the flag through 3 AND gates of size 2 and 2 OR gates of size 2. This gives us a total delay of 3*30 + 2*30 or 150.
The worst-case for all other gates at the ALU level are:
(for addition and subtraction, the overflow flag was the slowest part, which i san XOR and an AND)
ADD: 90 (+70)
SUB: 90 (+70)
XOR: (+70)
SLT: 150 (+70)
AND: (+70)
NAND: (+70)
NOR: (+70)
OR: (+70)
NOTE: Only ADD, SUB, and SLT had anything other than the LUT increasing its time at the ALU level.

For a bitslice, the end multiplexer takes a maximum of 70 + 100 = 170 (70 to get all of the individual values, 100 for the final OR gate). This is added for each multiplexer
The total theoretical for all inputs are as follows (with ADD, SUB, and SLT sharing almost the same input):

1.ADD:

As ADD, SUB, and SLT follow nearly the same mechanics within the adder, the delay for add is:
60 to XOR subtract and b (outside of adder)
40 + 30 + 30 to have changes propagate through all gates to the carryout (inside of adder)
This gives a total delay for just the ADD block of 140
As there are 32 bitslices and then we need to add the ALU timeframe, we get a total theoretical time of (160 + 170)*32 + 160 = 10720.

2.SUB:

Subtraction follows the same mechanics as addition. Based on this, we have a total time delay of 10720.

3.XOR:

The XOR function is an XOR gate, which takes 60 time units. This, combined with the multiplexer causes 160 units of delay per bitslice. As there are 32 bit slices, with the 160 delay from the ALU we get a total of 160*32 + 70 = 5210 time units.

4.SLT:
SLT follows the same mechanics as addition. Based on this, we have a total time delay of (160 + 170)*32 + 220 = 10780.

5.AND:
The AND function is summarized as an AND gate, which takes 30 units of time. Combined with the multiplexer, this is 130 units of delay per bitslice. When summing all bitslices and delay from the ALU we get a total of 130*32 + 70= 4230.

6.NAND:
The NAND function is summarized as a NAND gate, which takes 20 units of time. Combined with the multiplexer, this is 120 units of delay per bitslice. When summing all bitslices and delay from the ALU we get a total of 120*32 + 70= 3910.

7.NOR:
The NOR function is summarized as an NOR gate, which takes 20 units of time. Combined with the multiplexer, this is 120 units of delay per bitslice. When summing all bitslices and delay from the ALU we get a total of 120*32 + 70= 3910.

8.OR:
The OR function is summarized as an OR gate, which takes 30 units of time. Combined with the multiplexer, this is 130 units of delay per bitslice. When summing all bitslices and delay from the ALU we get a total of 130*32 + 70= 4230.

As the slowest theoretical was SLT, the two gtkwave tests we ran were when A changes in SLT and when B changes. As the least significant bit takes longest, we flipped exclusively the least significant bit. The first test (where we flip A from 0 to 1) takes from 5 us to 10 us; after that we zeroed everything again. Once a significant amount of time had passed (5 us), we ran the second test where we flipped B's least significant bit (starting at 15 us). The full tests can be seen below.

Section 4: Work Plan Reflection:

The implementation of LUT and the test bench took about the same time that we previously estimated in our work plan. The integration of the individual bitslice components with the LUT and the miscellaneous circuitry needed for SLT took a period of time that was longer than planned, but still reasonable.