Chris Aring, Rocco Diverdi

Computer Architecture: Lab 1

I. Implementation:

We decided to go with the bit-slice and LUT approach. With this approach, each bit of each operand was calculated separately starting with the least significant bit to the most significant bit. Each bit-slice inputs one bit of **a** and one bit of **b**, **carryin**, **command,** and **select.** The output of each bit-slice is **result**, and a **carryout**. One interesting choice we made was to use a 4 input, 2 select line **MUX**. We found thought that you can reduce the potential number of outputs to 4 if there is some other logic in the main alu.

To subtract on the ALU we needed to invert the **b** operand. Instead of inverting **b** within each bit-slice using an **invert** line, we decided to invert **b** at each bit-slice by **XOR**ing each **b** operand and **invert**. We decide if **b** should be inverted using the LUT and then invert **b** before it enters each bit-slice.

To create the full ALU we attached the **carryout** of each bit-slice to the following bit-slice's **carryin** except that our bitslice implementation required that external logic to eliminate carryover in **XOR** and **AND** operations. To do this, we added a **carry** line to the LUT, and **AND**ed each **carryout** with **carry** before passing it to the next bit's **carryin**. The first bit-slice's **carryin** was set by **AND**ing the **carry** signal and the first bit of the **command** signal. This is because the **carryin** for **SUB** and **SLT** is 1 while **carryin** for every other command is 0.

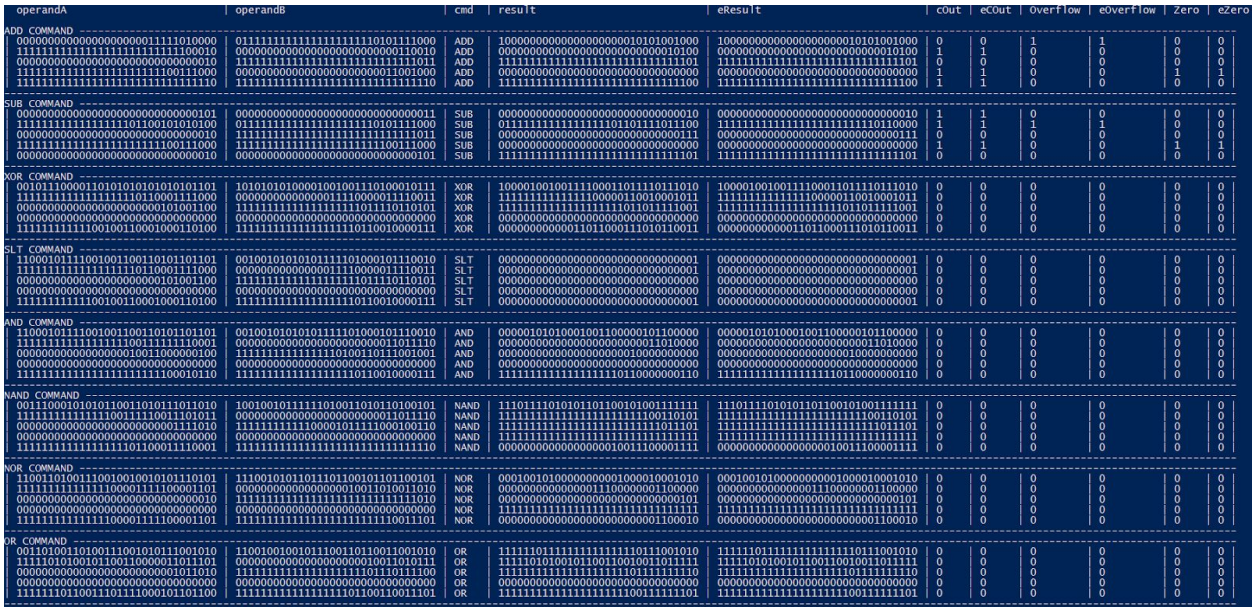
The **zero** flag was set by **OR**ing the **zero** flag of each bit-slice with the **zero** flag of the next bit-slice. We then **NOT** the final **zero** flag because we need the flag to be 1 when all the flags were 0's. Finally we have to **AND** the **zero** flag with the **command** line because we only want to output flags if the operation was addition or subtraction. We do the same **AND**ing process for all the flags for the same reason.

The **overflow** is calculated after the calculation is done by simply **XOR**ing the last and second-to-last internal **carryouts**.

Before calculating **SLT**, the entire output must be set to 0. To do this, we made **SLT** and **nSLT** flags. Each bit of the result is **AND**ed with the **nSLT** flag to set everything to 0 during the **SLT** calculation. **SLT** is calculated by **XOR**ing the **overflow** with the final calculated bit (before it's **AND**ed with the **nSLT** flag). To propagate that to the output, it is first **AND**ed with the **SLT** flag, then **OR**ed with

the first bit of the previously calculated result (note, that is **AND**ed with the **nSLT** flag, so it is always 0 if while calculating **SLT**).

II. Test Results:



operandA	operandB	cmd	result	eResult	cOut	eCOut	Overflow	eOverflow	Zero	eZero
ADD COMMAND										
00000000000000000000011111010000	01111111111111111111101011111000	ADD	10000000000000000000001010001000	10000000000000000000001010001000	0	0	1	1	0	0
11111111111111111111111111100010	00000000000000000000000000110010	ADD	00000000000000000000000000010100	00000000000000000000000000010100	1	1	0	0	0	0
00000000000000000000000000000010	11111111111111111111111111111011	ADD	11111111111111111111111111111101	11111111111111111111111111111101	0	0	0	0	0	0
11111111111111111111111100111000	00000000000000000000000011001000	ADD	00000000000000000000000000000000	00000000000000000000000000000000	1	1	0	0	1	1
11111111111111111111111111111110	11111111111111111111111111111100	ADD	11111111111111111111111111111100	11111111111111111111111111111100	1	1	0	0	0	0
SUB COMMAND										
00000000000000000000000000000101	00000000000000000000000000000011	SUB	00000000000000000000000000000010	00000000000000000000000000000010	1	1	0	0	0	0
11111111111111111111001010101100	01111111111111111111101011111000	SUB	01111111111111111111101011111100	11111111111111111111111110110000	1	1	1	1	0	0
00000000000000000000000000000010	11111111111111111111111111111011	SUB	00000000000000000000000000000111	00000000000000000000000000000111	0	0	0	0	0	0
11111111111111111111111100111000	11111111111111111111111001111000	SUB	00000000000000000000000000000000	00000000000000000000000000000000	1	1	0	0	1	1
00000000000000000000000000000010	00000000000000000000000000000101	SUB	11111111111111111111111111111101	11111111111111111111111111111101	0	0	0	0	0	0
XOR COMMAND										
00101110000110101010101010101101	10101010100000100100111010000010111	XOR	10000100100111100011011110111010	10000100100111100011011110111010	0	0	0	0	0	0
11111111111111111111101100011110000	00000000000000001111000000111100111	XOR	11111111111111110000011001000101	11111111111111110000011001000101	0	0	0	0	0	0
00000000000000000000000000101001100	11111111111111111111111111010101101	XOR	11111111111111111111111101011111001	11111111111111111111111101011111001	0	0	0	0	0	0
00000000000000000000000000000000000	00000000000000000000000000000000000	XOR	00000000000000000000000000000000000	00000000000000000000000000000000000	0	0	0	0	0	0
11111111111100100110001000110100	11111111111111111111101100100001111	XOR	00000000000011101100011101011001	00000000000011011000111010110011	0	0	0	0	0	0
SLT COMMAND										
11000101111000100110011010110101	00100101010101111101000101110010	SLT	00000000000000000000000000000001	00000000000000000000000000000001	0	0	0	0	0	0
11111111111111111111111011000111000	00000000000000001111000001110011	SLT	00000000000000000000000000000001	00000000000000000000000000000001	0	0	0	0	0	0
00000000000000000000000000101001100	11111111111111111111111011110101	SLT	00000000000000000000000000000000	00000000000000000000000000000000	0	0	0	0	0	0
00000000000000000000000000000000000	00000000000000000000000000000000000	SLT	00000000000000000000000000000000000	00000000000000000000000000000000000	0	0	0	0	0	0
11111111111100100110001000110100	11111111111111111111101100100001111	SLT	00000000000000000000000000000001	00000000000000000000000000000001	0	0	0	0	0	0
AND COMMAND										
11000101111000100110011010110101	00100101010101111101000101110010	AND	00000101010000100110000010110000	00000101010000100110000010110000	0	0	0	0	0	0
11111111111111111111001111111110001	00000000000000000000000000110111110	AND	00000000000000000000000000001101000	00000000000000000000000000001101000	0	0	0	0	0	0
00000000000000000000001001100000100	11111111111111111101001101111001001	AND	00000000000000000000000010000000000	00000000000000000000000010000000000	0	0	0	0	0	0
00000000000000000000000000000000000	00000000000000000000000000000000000	AND	00000000000000000000000000000000000	00000000000000000000000000000000000	0	0	0	0	0	0
11111111111111111111111100010110	11111111111111111111101100100001111	AND	11111111111111111111101100000000110	11111111111111111111101100000000110	0	0	0	0	0	0
NAND COMMAND										
00111000101010110011010111011010	10010010111111010011010110100101	NAND	11101111010101101100101001111111	11101111010101101100101001111111	0	0	0	0	0	0
11111111111111110011111001101011	00000000000000000000000000110111110	NAND	11111111111111111111111111001101011	11111111111111111111111111001101011	0	0	0	0	0	0
00000000000000000000000000111010	11111111111100010101111100010011010	NAND	11111111111111111111111111011101101	11111111111111111111111111011101101	0	0	0	0	0	0
00000000000000000000000000000000000	00000000000000000000000000000000000	NAND	11111111111111111111111111111111111	11111111111111111111111111111111111	0	0	0	0	0	0
11111111111111111011000111100001	11111111111111111111111111111110	NAND	00000000000000000010011100001111	00000000000000000010011100001111	0	0	0	0	0	0
NOR COMMAND										
11001101001110011001001010111010101	11100101011011011001011011001101	NOR	00010010100000000010000100101010	00010010100000000010000100101010	0	0	0	0	0	0
11111111111111111000011111000011101	00000000000000000000100110101011010	NOR	00000000000000001100000001100000	00000000000000001100000001100000	0	0	0	0	0	0
00000000000000000000000000000000010	11111111111111111111111111111111010	NOR	00000000000000000000000000000000101	00000000000000000000000000000000101	0	0	0	0	0	0
00000000000000000000000000000000000	00000000000000000000000000000000000	NOR	11111111111111111111111111111111111	11111111111111111111111111111111111	0	0	0	0	0	0
11111111111111111000011111000011101	11111111111111111111111111011101	NOR	00000000000000000000000001100010	00000000000000000000000001100010	0	0	0	0	0	0
OR COMMAND										
00110100110100111001010111001010	11001001001011100110110011001010	OR	11111101111111111111111011001010	11111101111111111111111011001010	0	0	0	0	0	0
11111010100101100110000011011101	00000000000000000000000100110101111	OR	11111010100101100110010011011111	11111010100101100110010011011111	0	0	0	0	0	0
00000000000000000000000000001011010	11111111111111111111111011110111100	OR	11111111111111111111111011111111110	11111111111111111111111011111111110	0	0	0	0	0	0
00000000000000000000000000000000000	00000000000000000000000000000000000	OR	00000000000000000000000000000000000	00000000000000000000000000000000000	0	0	0	0	0	0
11111110110011011111000101101100	11111111111111111111001100111101	OR	11111111111111111111111001111111101	11111111111111111111111001111111101	0	0	0	0	0	0

For each command we did five tests.

1. For our **ADD** command we chose to test the addition of two positive numbers, two negative numbers, two numbers adding to zero, and two more tests where either number **a** or **b** was negative and the other positive. We were able to check for the zero, carryout and overflow flags.
2. For our **SUB** command we chose the same type of tests just with subtraction instead of addition. This allowed us to test the same flags as our **ADD** command.
3. For our all the basic boolean commands we chose a variety of tests. They all passed and no flags were set which is what we wanted since a MIPS ALU only sets flags for the **ADD** and **SUB** commands.
4. Out **SLT** command did not initially work. The first test did not have the right expected result and broke every other test. This was because we chose to **AND** our partial results with an **SLT** flag when we should have been **AND**ing it with a **NOT SLT** flag.

In addition, we exhaustively tested each module (mux, aluBit, and aluFullBit). Those tests caught an issue with our logic for the select lines and revealed the requirements on carrying between bit-slices.

III. Timing Analysis:

Alubit without mux: 50 (2 input nands + 3 input nand)
Mux: 80 (not + 3 input nand + 4 input nand)
Full bitslice: 130 (bit + mux)
Carry: 50 (same as bit)
Alu repeated: 190 (2x 2 input and + full bitslice)
Carry: 80 (full bitslice carry + 2 input and)
Alu before: 40 (3 input nand + not)
Alu after: 80 (2 input: xor + and + or)
Alu total: 2790 (40+80*31+190+80)

IV. Work Plan Reflection:

Original Work plan:

Completing test bench: 2hr, done by Tues Oct 10
Writing and testing logic verilog: 2hr, done by Tues Oct 10
Finishing report: 1hr, by Thursday Oct 12
Total: 5hr

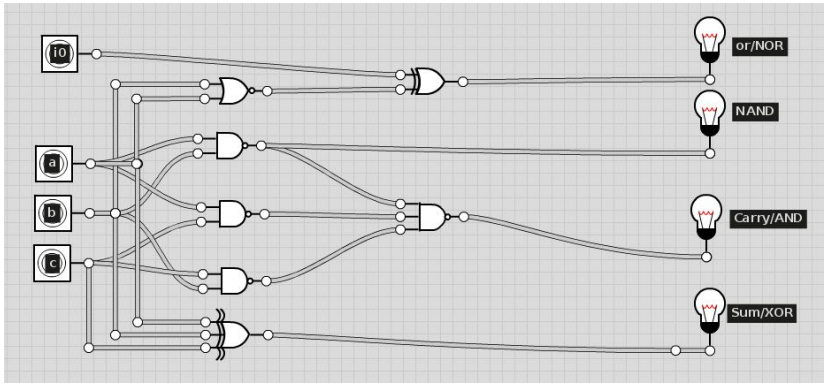
Actual:

Testbenches actual: 3hr, done Oct 12
Writing and debugging logic: 5hr, done Oct 12
Report: 3hr, done Oct 12

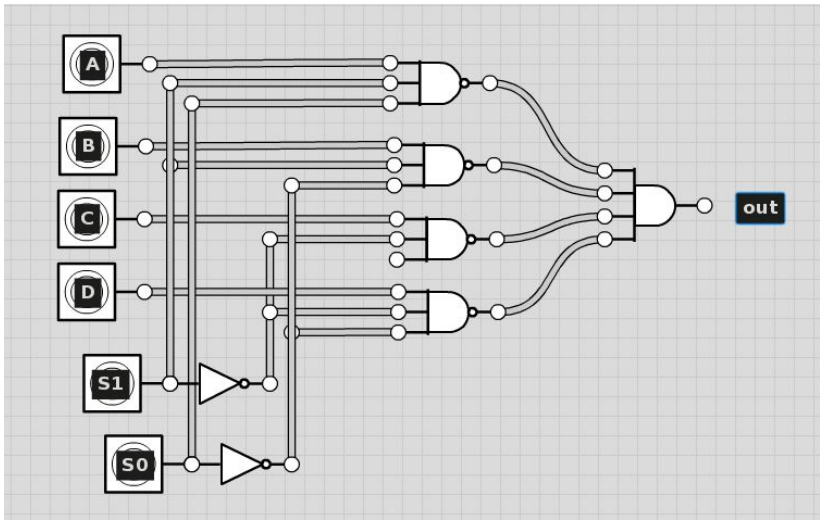
Actually making things work took a lot longer than expected, also we were busy and didn't start working until yesterday, which probably wasn't a good plan.

V. Block Diagrams

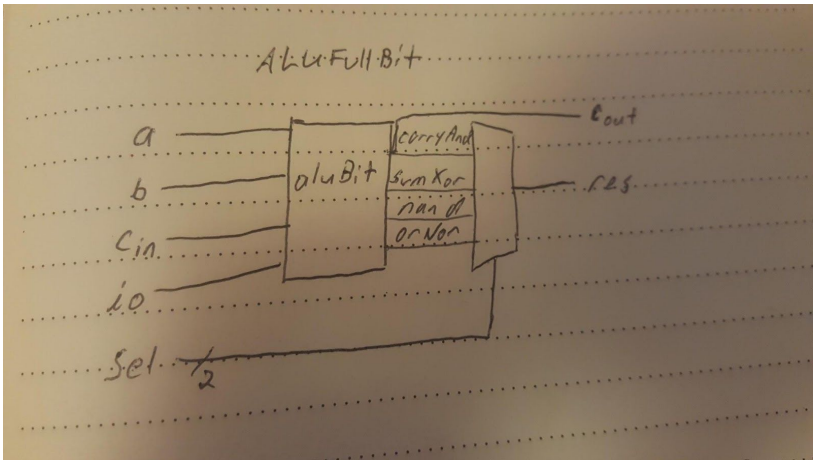
Single bit without mux:



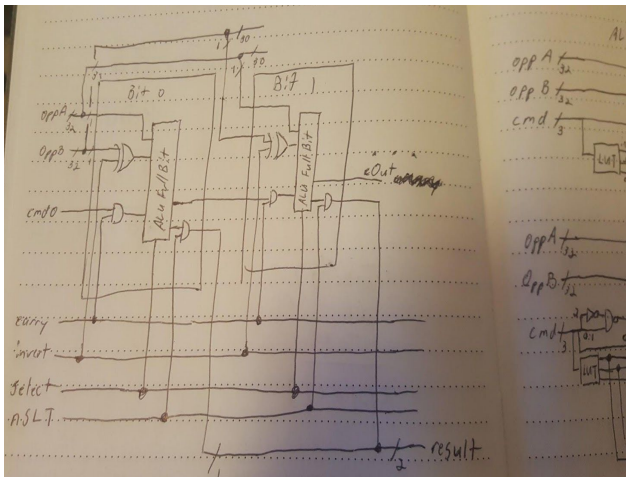
Mux:



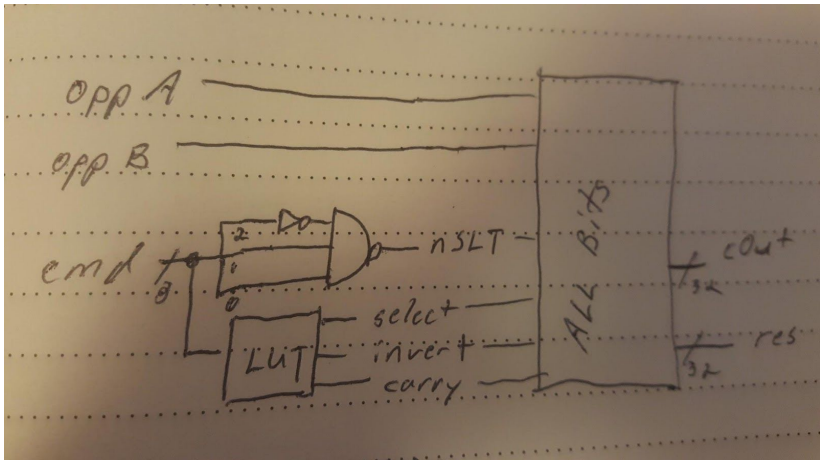
Full ALU bitslice:



Alu Repeated bit pattern



Alu pre-repeated section:



Alu post-repeated section: