Lab 4'b0001

Ruby Spring, Kyle Mayer, Casey Alvarado, Pratool Gadtaula

October 10, 2014

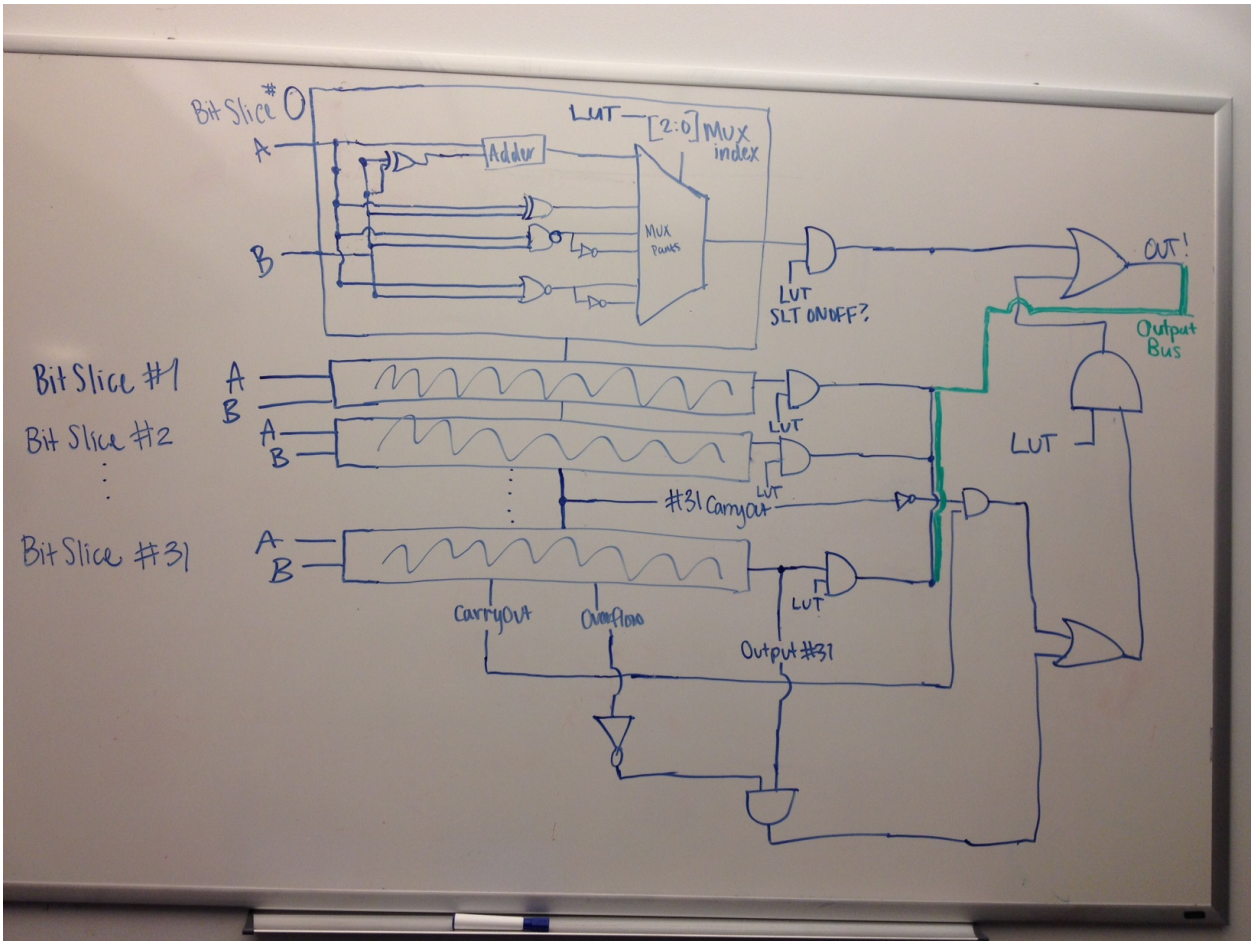

Figure 1: 32 bit ALU schematic made from 1 bit slices. The first 1 bit slice includes the inner structure of our 1 bit slice, but the following 31 bits are abstracted into black boxes. The green line is the 32-bit result bus.

1 ALU Design

Project Manager,

The Kyle, Casey, Pratool, and Ruby group has done a promisingly fine job of designing an ALU, or a pair of pants, to fit your needs! We have designed an ALU that performs one of eight operations on two 4-byte inputs. The eight operations include: add, subtract, SLT, AND, NAND, OR, XOR, and NOR. We

have put much thought into our test cases and we have shown our results below. We discussed the trade offs between reusing gates and creating new ones. Benefits of reusing gates include minimizing silicon space and maximizing modularity. Cons of reusing gates can sometimes cause an increase in time delay. It also increases the complexity of the design and makes our circuit less human readable.

The design involved constructing the 32 Bit ALU from 1-bit slices. The pros of creating our structure allows for smaller, more easily scalable, and more comprehensible modules. The cons include exercising extra caution when stringing the bits together for fear of tangling the outputs.

2 Test Cases

Methodology for testing ALU: A single bitslice was tested exhaustively. A bitslice has a 3 bit command input, a 1 bit A input, and a 1 bit B input. This means $2^5 = 32$ possible testcases. Rather than try to decide which ones would sufficiently test the bitslice, all were tested. The add, subtract, and SLT cases also have a carryin. For each of the other operations, one case was repeated with carryin set high to show that carryin does not affect the result. Note that with our hierarchy, there may be a carryout from an individual bitslice even for modes that are not addition and subtraction, but there will not be a carryout from the entire ALU.

Next, a four bit ALU was constructed, and heavily tested. We tested anywhere from 3 to 6 cases for each type of operation, and selected these cases specifically to test all the corner scenarios. Below is a brief discussion of how these were selected for each operation.

An eight bit ALU was also constructed, and light testing was performed. We have included the eight bit test bench; however, we do not believe to have learned additional useful information from this test, so the results have been omitted.

Finally, we performed light testing on a 32 bit ALU, to verify that increasing the size didn't break anything. Since the four bit ALU was heavily tested, and the 32 bit ALU was created using the same parameterized code, these tests were primarily used to make sure that all of the parameters were correctly implemented. These test cases included one test of each operation, pseudo-randomly generated.

For the four bit addition, we made sure to try one case that had a carryout, one case that had an overflow, and one case that added to zero. For the subtraction, we tried a case involving a positive minus a negative number, one case involving a positive minus a positive number (testing carryout), one case with an overflow, and one case involving a negative number minus a negative. For SLT, we tested a case where A<B with no overflow, a case where A<B with overflow, a case where A>B with no overflow, a case where A>B with overflow, and a case where A=B For the simple gates (XOR, AND, OR, NAND, and NOR), we tested two random cases, as well as a third case designed to prove that the overflow and carryout are disabled (set to 0).

Below are figures depicting the results of the three testbenches implemented: an exhaustive test of a single bitslice, a thorough test of a four bit ALU, and a light testing of a 32 bit ALU.

Command	A	B	Cout	result	Zero	OFL	What are we testing?
000	11101010111011000010000100011011	11110011101011010110100000100010	1	11011110100110011000100100111101	0	0	Addition
001	01101101001111000011101001011010	01111000110000100010101001100001	0	11110100011110100000111111111001	0	0	Subtract
010	01001001100010001011111000000100	10111011001110101001101110001101	0	11110010101100100010010110001001	0	0	XOR
011	00010011111011100001110000000111	11110100010101010110010100100011	0	00000000000000000000000000000000	1	0	SLT
100	00001000001100110000000011011110	01111001111110010011001001001111	0	00001000001100010000000001001110	0	0	AND
001	11011110011111000011110100100100	11101101011010101000101010000100	0	11110001000100011011001010100000	0	0	Subtract
110	00100000101110111011111100101000	00000110010101000010010100001000	0	11011001000000000100000011010111	0	0	NOR
111	00100001111111010101000011000000	11110001110000010011101100100001	0	11110001111111010111011111100001	0	0	OR

Figure 2: Test bench and results for a 32 bit ALU.

Command	A	B	Cout	result	Zero	OFL	Expected Output	What are we testing?
000	0011	0100	0	0111	0	0	0111	Addition
000	0100	1111	1	0011	0	0	0011	Addition, Carryout, but NOT an overflow (pos plus neg)
000	0010	0111	0	1001	0	1	1001	Addition, overflow, but NOT a carryout (pos plus pos)
000	1011	0101	1	0000	1	0	0000	Addition, zero (pos plus neg). only official test for zero.
001	0111	0001	1	0110	0	0	0110	Subtraction (pos minus pos)
001	0011	0111	0	1100	0	0	1100	Subtraction w/ Carryout (ie, second number larger than first), no overflow (pos minus pos)
001	0011	1001	0	1010	0	1	1010	Subtraction, no carryout, yes overflow (pos minus neg)
001	1011	1001	1	0010	0	0	0010	Subtraction (neg minus neg), therefore no overflow.
010	0011	0011	0	0000	1	0	0000	XOR: random test case 1
010	0001	1101	0	1100	0	0	1100	XOR: random test case 2
010	1111	1111	0	0000	1	0	0000	XOR: carryout and overflow are always false
011	0101	0111	0	0001	0	0	0001	SLT A<B, no overflow
011	0111	0011	0	0000	1	0	0000	SLT A>B, no overflow
011	1110	0111	0	0001	0	0	0001	SLT A<B, overflow (flag should remain unset)
011	0111	1011	0	0000	1	0	0000	SLT A>B, overflow (flag should remain unset)
011	0101	0101	0	0000	1	0	0000	SLT A=B, no overflow
011	1111	1111	0	0000	1	0	0000	SLT carryout should be false (disabled)
100	1001	0011	0	0001	0	0	0001	AND: random case 1
100	1001	1101	0	1001	0	0	1001	AND: random case 2
100	1111	1111	0	1111	0	0	1111	AND: carryout and overflow are always false
101	1100	1011	0	0111	0	0	0111	NAND: random test case 1
101	1001	1000	0	0111	0	0	0111	NAND: random test case 2
101	1111	1111	0	0000	1	0	0000	NAND: carryout and overflow are always false
110	0001	1100	0	0010	0	0	0010	NOR: random test case 1
110	1100	1011	0	0000	1	0	0000	NOR: random test case 2
110	1111	1111	0	0000	1	0	0000	NOR: carryout and overflow are always false
111	0111	0001	0	0111	0	0	0111	OR: random test case 1
111	1100	1001	0	1101	0	0	1101	OR: random test case 2
111	1111	1111	0	1111	0	0	1111	OR: carryout and overflow are always false

Figure 3: Extensive testing of corner cases for a 4 bit ALU, along with reasons for why individual test cases were selected.

Command	A	B	carryin	Cout	result	What are we testing?
000	0	0	0	0	0	Addition: 00
000	0	1	0	0	1	Addition: 01
000	1	0	0	0	1	Addition: 01
000	1	1	0	1	0	Addition: 10
000	0	0	1	0	1	Addition: 01
000	0	1	1	1	0	Addition: 10
000	1	0	1	1	0	Addition: 10
000	1	1	1	1	1	Addition: 11
001	0	0	0	0	1	Subtraction: 01
001	0	1	0	0	0	Subtraction: 00
001	1	0	0	1	0	Subtraction: 10
001	1	1	0	0	1	Subtraction: 01
001	0	0	1	1	0	Subtraction: 10
001	0	1	1	0	1	Subtraction: 01
001	1	0	1	1	1	Subtraction: 11
001	1	1	1	1	0	Subtraction: 10
010	0	0	0	0	0	XOR: result false
010	0	1	0	0	1	XOR: result true
010	1	0	0	0	1	XOR: result true
010	1	1	0	1	0	XOR: result false
010	1	1	1	1	0	XOR: carryin and carryout don't matter
011	0	0	0	0	1	SLT (subtract): 01
011	0	1	0	0	0	SLT (subtract): 00
011	1	0	0	1	0	SLT (subtract): 10
011	1	1	0	0	1	SLT (subtract): 01
011	0	0	1	1	0	SLT (subtract): 10
011	0	1	1	0	1	SLT (subtract): 01
011	1	0	1	1	1	SLT (subtract): 11
011	1	1	1	1	0	SLT (subtract): 10
100	0	0	0	0	0	AND: result false
100	0	1	0	0	0	AND: result false
100	1	0	0	0	0	AND: result false
100	1	1	0	1	1	AND: result true
100	1	1	1	1	1	AND: carryin and carryout don't matter
101	0	0	0	0	1	NAND: result true
101	0	1	0	0	1	NAND: result true
101	1	0	0	0	1	NAND: result true
101	1	1	0	1	0	NAND: result false
101	1	1	1	1	0	NAND: carryin and carryout don't matter
110	0	0	0	0	1	NOR: result true
110	0	1	0	0	0	NOR: result false
110	1	0	0	0	0	NOR: result false
110	1	1	0	1	0	NOR: result false
110	1	1	1	1	0	NOR: carryin and carryout don't matter
111	0	0	0	0	0	OR: result false
111	0	1	0	0	1	OR: result true
111	1	0	0	0	1	OR: result true
111	1	1	0	1	1	OR: result true
111	1	1	1	1	1	OR: carryin and carryout don't matter

Figure 4: Nearly exaustive testing of a single bitslice.

Our testbenches enabled us to catch a few errors in our design, and correct them accordingly. One error was that our MSB and our LSB were swapped when we made an eight bit multiplexer. This caused errors in our testbenches due to us performing the incorrect operation.

3 Timing Analysis

We determined the Worst Case Delays (WCD) for each slice, then multiplied the delays of add, subtract, and SLT by 32 and added overflow.

OP	ADD	SUB	XOR	SLT	AND	NAND	NOR	OR
WCD	17080	18680	500	18780	470	450	450	450

4 Work Plan Analysis

When our group began this lab we decided to all push our own ALUs to completion (design and verilong code) and then come together at the end to discuss what we'd done and do the write-up together. Although this worked well in the past, this approach failed for this lab. Most of us did not have our ALU's fully designed or coded by Tuesday, October 8th (our deadline for a complete coded ALU and ALU test bench), and on our Wednesday October 9th deadline (write-up completion) we were all sitting in front of the same TV implementing the SLT for one team member's code. We realized this was how we should have been doing the lab from the start –learning by coding together as a team –as apposed to struggling individually with problems we were all having. This would have saved all of us time and sleep, but it's also worth noting that the team-coding approach may not allow for as much learning for each team member, especially in working through Verilog implementation of our designs.

In theory, we planned that the ALU design would take a maximum of 3 hours to design. We actually took closer to 6 hours designing and understanding our ALU over the course of two days. We also planned that coding the ALU in Verilog would take a maximum of 3 hours and making the test bench for our ALU would take a maximum of 2 hours. In reality, the average total time spent coding the ALU was about 12 hours per person. Much of this time was spent just understanding syntax and how to deal with 32-bit buses in Verilog, especially when wiring 32 one-bit adders together. The test bench took 1.5 hours.

While writing this lab report, we had already spent the number of days we expected to allocate in order to finish this lab, including the writeup. We scheduled to finish the write-up in 3 hours. This work plan analysis took about 1 hour to complete.

In conclusion this lab took much more time than we planned for. In the future, we hope to break down the lab into workable components for each member of the group and bring the parts together as a group. Each member would then explain how his/her part works to the group, and as a group incorporate the individual parts into the design.