Lab 1: Arithmetic Logic Unit

Emma, March, Vivien

October 5, 2018

Implementation

Our ALU has a full addition, subtract, xor, set less than, and, nand, or, and nor functionality. It has been designed with a bitslice approach. Working backwards, we are able to select the functionality of the ALU using a 5 input multiplexer with two select signals. The relationship between the two select signals and the inputs of the system is shown below.

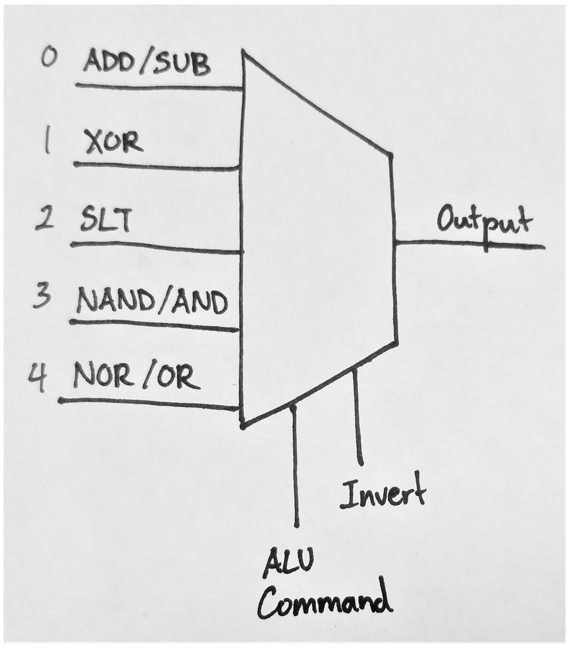

Figure 1: 5-input multiplexer

ALU Command	MUX Index	Invert
000 (ADD)	0	0
001 (SUB)	0	1
010 (XOR)	1	0
011 (SLT)	2	0
100 (NAND)	3	0
101 (AND)	3	1
110 (NOR)	4	0
111 (OR)	4	1

How the invert signal is used to choose between the two functionalities on the same input is shown below.

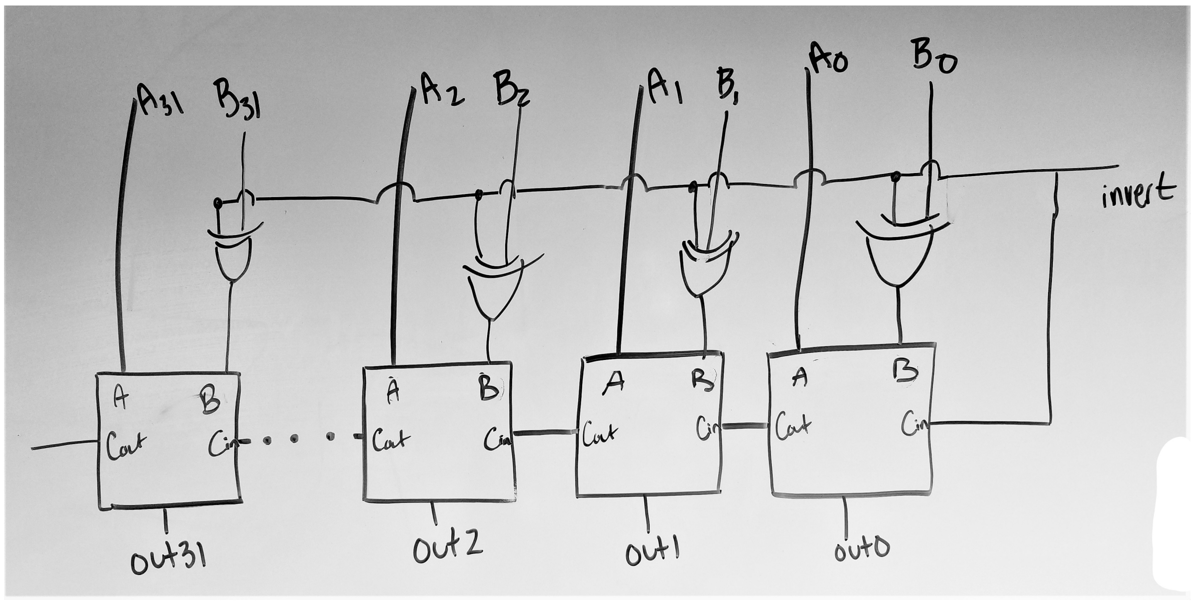

Figure 2: 32-bit adder/subtractor

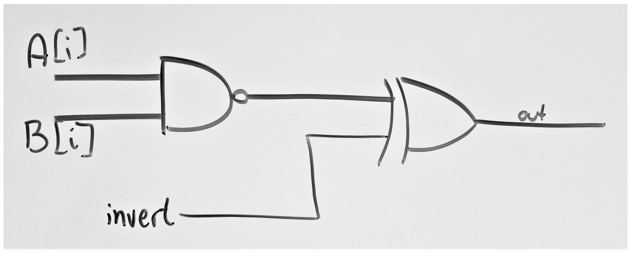

Figure 3: AND module XOR-ed from NAND and invert

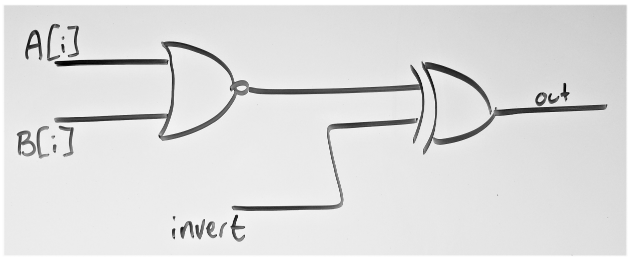

Figure 4: OR XOR-ed from NOR and invert

The xor module is implemented using a standard xor gate.

The SLT module is implemented using its own Add-Subtract module that is always set to subtract the two inputs, then look at the sign of the output. A design flaw is that we had to implement another Add-Subtract module instead of using the one that exists on port 0 of the multiplexer. Allowing these two to share some circuitry would decrease the size of the overall system.

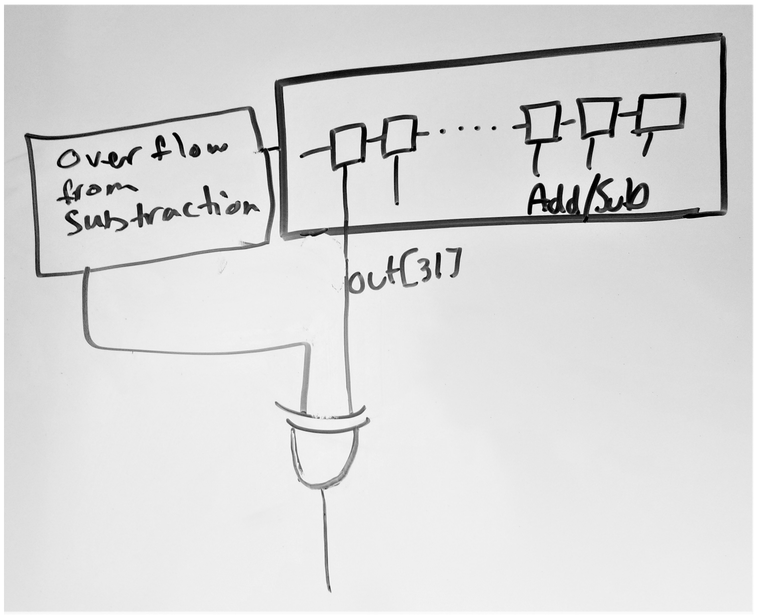

Figure 5: SLT module XOR-ed from the difference between A and B and the resulting overflow

Test Bench

For the adder, we chose test cases that resulted in no carryout and no overflow, overflow only, carryout only, and both carryout and overflow, as well as two instances in which zero was true. Both positive and negative operands and results were accounted for.
For the subtracter, we used similar test cases as in the adder that resulted in no carryout and no overflow, overflow only, carryout only, and both carryout and overflow, and one instance in which zero was true. Both positive and negative operands and results were accounted for.
For the xor module, since we wrote all of our test cases in hexadecimal, we chose operands with repeating 4-bit values. The 4-bit values were matched in such a way that exhaustively tested 1-bit xor results.
For the SLT, we chose test cases that compared two positive numbers, two negative numbers, and a positive and a negative number, with one case for each in which A is less than B and one in which A is greater than B. We also compared two zeroes to make sure that SLT is still false when A is equal to B. We also added two cases to try to create worst-case propagation delays.
For the and, nand, nor, and or modules, we used the same operands as in the xor module with repeating 4-bit values. The 4-bit values were matched in such a way that exhaustively tested 1-bit and, nand, nor, and or results.

Changes Made / Learnings From Testing

1. Failing the zero tests showed us we had to change from ORing everything to NORing everything.

2. We found that our Adder did not work when either overflow or carryout was not zero - they

defaulted to x. This told us that we were not setting the values correctly. We were assigning carryout to the carryin instead of the other way around.

3. In addition, overflow was being calculated inside the one-bit adder module instead of after generating (adding/subtracting) all 32 bits. The tests passed after moving the overflow calculation into the ALU module after the generate for adder.

Module Testing

We tested 3-bit instantiations of our ADD/SUB and SLT units to make sure they were working as expected before we jumped into deeper waters.
When there was no carryout/carryin passed between modules, the functionality was as expected, but when there were things didn't work right. After some amount of debugging we changed the internals of a generate loop from:
AddSubN adder(.sum(sub[i]), .carryout(carryout), .overflow(overflow0), .a(a[i]), .b(b[i]), .carryin(carryin0), .subtract(subtract)); assign carryin0=carryout
to
AddSubN adder(.sum(sub[i]), .carryout(carryin0[i+1]), .overflow(overflow0), .a(a[i]), .b(b[i]), .carryin(carryin0[i]), .subtract(subtract));
The issue was a translation issue between software and hardware. Instead of assigning values like we would in software, we used an n+1 length value to contain the carryin/carryout data. Each instance will reference the carryin value at location i, and would ouput the carryout value (the next instance's carryin) at location i+1. When i increments, the loop is able to repeat.

Timing Analysis

Note, we modeled the gate delay of a XOR as 50 based on our assumption of how the default XOR is implemented.

AND/NAND

Because the AND/NAND operation happens through bitslicing, the total gate delay for that operation is 10 (for the NAND gate) + 50 (for the XOR gate) + 280 (for the 5 input MUXes of the 32-bit result bus, Carryouts and Overflows) = 340.

OR/NOR

The OR/NOR hardware follows that of the AND/NAND hardware so the gate delay is calculated as 340 as well.

XOR

The XOR module has a gate delay of 50 (for the bitsliced XOR operation) + 280 (explained above) = 330.

ADD/SUB

To analyze these gate delays, we have to look at the one-bit add/sub module (since this was approached through the bitslice method as well). This one-bit module has gate delays of:

	Cout	Sum
a/b	90	100
Cin	40	50

But, our system XORs input ”b” so the gate delays become:

	Cout	Sum
a/b	140	150
Cin	40	50

The delay of the ADD/SUB module is limited by the propagation of the carryout signal. The first carryout signal is created with a gate delay of 90. The remaining carryout signals have a delay of 40 resulting in a total propagation delay of 90 + 3 1 * 40 = 1330 before MUXing.
Because we used Generate we could not calculate overflow until after the signal had propagated, so we have an XOR gate to find the overflow after that happens. This adds a delay of 50. Finally the MUXes add a delay of 3 * 280. This sums to a propagation delay of 1330 + 50 + 280 = 1660.

SLT

The SLT relies on the calculation of a subtraction an so has a baseline propotagion delay of 1330. Two more XOR gates calculate overflow from the subtraction and decide what the SLT result will be. This adds a delay of 50. Finally the 3 * 280) delay from the muxes applies. This sums to 1330 + 50 + 50 + 280 = 1710.

GTK Waves

Figure 6 and Figure 7 show the waveform outputs from running our test bench. We tried to select cases that would result in worst-case gate delays (in addition to cases that would thoroughly test our system).

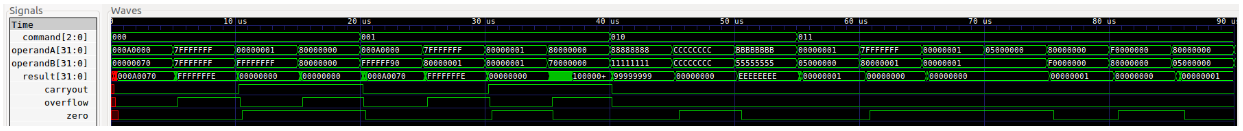

Figure 6: Waves from test bench for ADD/SUB, XOR and SLT.

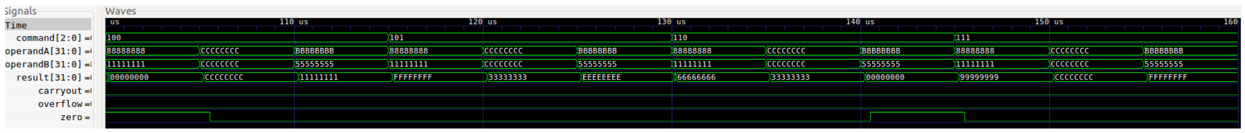

Figure 7: Waves from test bench for AND/NAND and OR/NOR.

Encouragingly, the worst-case gate delays matched on the order of magnitude level to the ones predicted above:

	Delay
ADD	1750
SUB	1780
XOR	460
SLT	1790
AND/NAND/OR/NOR	430

It makes sense that the propagation delays do not match exactly as our calculated delays are based on a simple model or we may not have chosen absolute worst-case scenarios.

Work Plan Reflection

1. Add/Sub module and SLT, associated test cases and testing
   (a) Expected: 1.5 hours
   (b) Actual: 2 hours
2. NOR/OR/AND/NAND/OR/XOR, associated test cases and testing
   (a) Expected: 1.5 hours
   (b) Actual: 2 hours
3. Integrate MUX
   (a) Expected: 2 hours
   (b) Actual: 3 hours
4. Test benches
   (a) Expected: 3 hours. 1 hour to write. 2 hours to fix
   (b) Actual: 5+ hours. 1 hour to write. 5+ hours to fix
5. Writing a report
   (a) Expected: 2 hours
   (b) Actual: 3 hours