Computer Architecture: Lab b001

Lauren Anfenson
Anusha Datar
Will Fairman

October 05, 2018

1 Implementation

We've designed, implemented, and tested an Arithmetic Logic Unit (ALU) that performs the following operations: addition, subtraction, set-less-than (SLT), XOR, OR, AND, and NAND. Each operation can be accessed with a 3-bit address `S`. The indexes of each of the operations are shown in the table.

Address	Output
000	ADD
001	SUB
010	XOR
011	SLT
100	AND
101	NAND
110	NOR
111	OR

In designing the ALU we took a modular bitslice approach. Each of the operations inside of the ALU were written into their own modules and tested, then combined into a 1-bit ALU. 32 1-bit ALUs were cascaded to create the 32-bit ALU, with the 32nd module being slightly modified to include an output for both the overflow and zero flags for the entire 32-bit module. An overview of our architecture can be seen in the figure below.

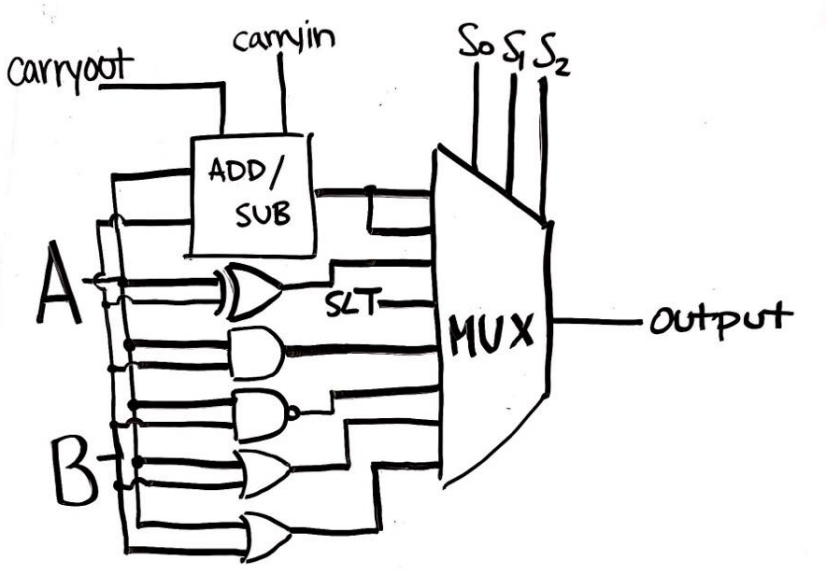

Figure 1: One bitslice of the ALU

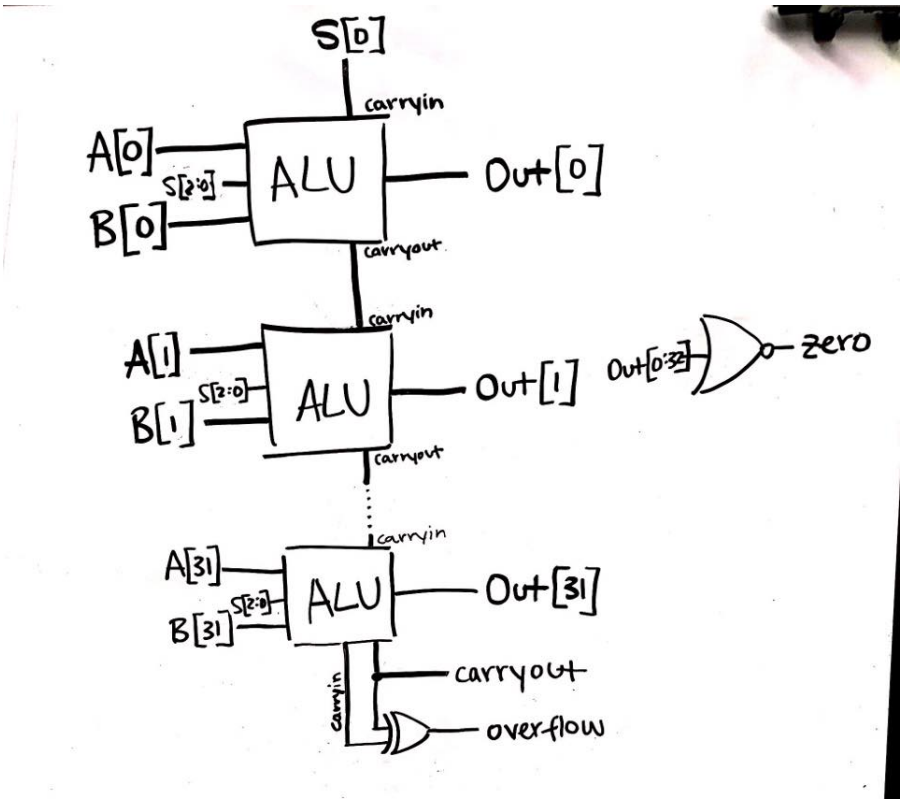

Figure 2: Architecture of our full 32-bit ALU

The addition and subtraction operations were combined into one module to reduce area. The carryin input of the first 1-bit ALU module is connected to the 0th address bit, meaning it is 0 if addition is being performed and 1 if we want to do subtraction. The 1-bit ALUs also have an internally modified B that is the output of an XOR gate with S[0] and B as inputs so that, when subtraction is being performed and S[0] is

1, B is inverted. The `ZERO` flag was implemented with a single 32-input NOR gate at the output of the ALU that outputs a 1 in the most significant bit position when our output is zero. All of the operations have been built structurally in Verilog, while the multiplexer that controls which output is shown was implemented behaviorally.

2 Test Results

We performed self-checking tests on each of the operations independently to verify correct operation before they were all integrated into the ALU. One bit addition, subtraction, and all of the single gate operations were tested exhaustively due to the limited number of potential test cases. The multiplexer was also tested independently and completely to make sure all 8 possible outputs worked as expected. By starting from the gate level, we were able to check each step of our implementation before building on top of it.

After we integrated all of the operations into our 1-bit ALU, we again exhaustively tested each operation to ensure correct operation. One major flaw we caught in this testing was our definition of the address `S`. We had defined it in least significant bit, causing us to accidentally call incorrect outputs and use the wrong carryin when performing addition or subtraction. This was fixed by flipping our definition of S from `[0:2]` to `[2:0]`. In addition to testing the general 1-bit ALU, we also wrote a suite of tests for the final ALU in the cascade to ensure that the overflow flag worked as expected.

After we confirmed that the 1-bit ALU was working we cascaded them into the full 32-bit ALU, with the first thirty-one gates identical and the final gate built to output an overflow flag. This was not tested exhaustively due to the huge number of test cases this would require. So, we chose some select cases for each operation in order to test our full ALU.

For our single gate operations, we chose two cases for each gate that would produce a result that was all true and a result that was all false. Then we included a few test cases with a variety of bit combinations to ensure the bit-level logic was accurate.

To test addition we chose cases that purposefully caused overflow in order to test the overflow flag, a few that resulted in a carryout to make sure that was working, and a few general addition cases like all zeros and all ones to make sure it was operating correctly. Cases for subtraction were chosen in much the same way, also including test cases for combinations like subtracting a negative number from a positive number, or subtracting a negative number from a negative number. This was to make sure that all combinations of signed numbers could be handled by our module.

For SLT we chose test cases that would check a variety of sign combinations – checking a positive number against a negative number, positive against positive, etc.

The testing for the `zero` was incorporated into the testing of the addition and subtraction modules, given that they were the only modules that relied on`zero` functioning properly.

3 Timing Analysis

All of our timing analysis depends on an assumption of 10 units/input gate delay.

3.1 AND

3.1.1 Theoretical

Each AND gate has two inputs and a single inverter. Therefore, each individual gate has $3 * 10 = 3)$ units of delay. With 32 gates (one per 1-bit ALU), this is $30 * 32 = 960$ units of gate delay.

3.2 Simulated

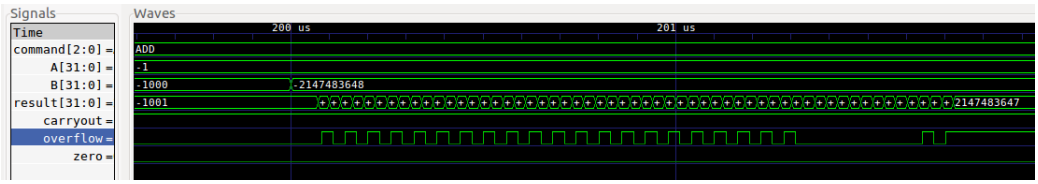

Figure 3: propagation delay of the 32-bit adder during an AND operation.

3.3 OR

3.3.1 Theoretical

Each OR gate has two inputs and a single inverter. Therefore, each individual gate has $3 * 10 = 30$ units of delay.

3.4 NAND

3.4.1 Theoretical

Each NAND gate has two inputs and no inverters. Therefore, each individual gate has $2 * 10 = 20$ units of delay.

3.5 NOR

3.5.1 Theoretical

Each NOR gate has two inputs and no inverters. Therefore, each individual gate has $2 * 10 = 20$ units of delay.

3.6 XOR

3.6.1 Theoretical

Each XOR gate has two inputs and no inverters. Therefore, each individual gate has $2 * 10 = 20$ units of delay. With 32 gates (one per 1-bit ALU), this is $20 * 32 = 640$ units of gate delay.

3.7 ADD

3.7.1 Theoretical

Each individual adder (including accounting for carryin and carryout) has 3 XOR gates, 2 AND gates, and 1 OR gate. This implies a $2 * 20 + 2 * 30 + 30 = 130$ unit delay. With 32 adders, this is $32 * 150 = 4800$ unit gate delay for the adder.

3.8 SUB

3.8.1 Theoretical

Each individual adder (including accounting for carryin and carryout) has 3 XOR gates, 2 AND gates, and 1 OR gate. This implies a $3 * 20 + 2 * 30 + 30 = 150$ unit delay. With 32 adders, this is $32 * 150 = 4800$ unit gate delay for the adder.

3.9 SLT

3.9.1 Theoretical

The SLT operation completes subtraction and then compares the overflow and sum with an XOR gate. Therefore, the value $4800 + 20 = 5000$ unit gate delay.

3.9.2 Simulated

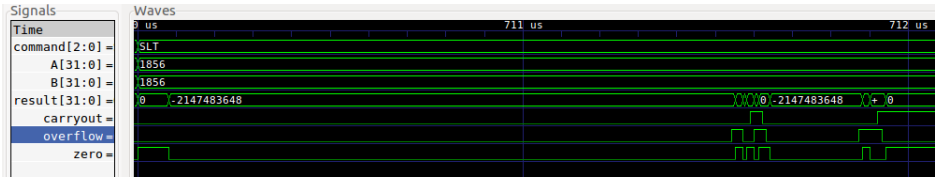

Figure 4: Propagation delay of our 32-bit adder while calculating SLT.

3.10 MUX

3.10.1 Theoretical

Each multiplexer consists of 8 4-input AND gates and one 8-input OR gate – each signal has a worst-case path through one AND gate and the OR gate to the output. This implies a $4 * 10 + 8 * 10 = 120$ unit delay.

4 Work Plan Reflection

In our work plan we broke this lab up into 6 tasks: Design modules for each operation, test modules for each operation, integrate modules into the ALU, test the ALU, look at and document propagation delays, and write report. We then gave each of these tasks an estimate of long it would take and a due date so we could make sure we were on track to finish the lab on time.

Task	Time Alloted	Due Date
Design operation modules	2.5 hours	10/2
Test operation modules	3 hours	10/2
Integrate operations	1 hour	10/4
Test ALU	1.5 hours	10/4
Look at propagation delays	1 hour	10/4
Write report	2.5 hours	10/5

Table 1: Work Plan

Our overall timeline was almost exactly what we detailed in the work plan. Our time estimates were also pretty accurate, as our work plan estimates we would need 11.5 hours and it took us about 12 hours to complete the lab.