# Computer Architecture Lab 1: Arithmetic Logic Unit

Daniel Connolly and Josh Deng

October 4, 2018

## 1 Implementation

For our 32-bit ALU design, we chose a bit-sliced ALU design where we built a single-bit ALU capable of performing the standard ALU functions and then reused the module to expand the ALU to be capable of 32-bit operations. As can be seen at the bottom of Figure 1, one of our bit-slice ALUs consists of an adder/subtractor unit, an XOR logic gate, a wire for set less than input, a NAND logic gate with an output inverter, and a NOR logic gate with an output inverter. The outputs of all the above operations are fed into a multiplexer, which outputs according to the mux index bit number. A LUT controls all of the bit slices by setting the mux index (for selecting the output of the multiplexer), invB (for inverting B to perform subtraction), and invOut (for inverting the output of the NAND and NOR logic gates to achieve AND and OR without having to build them as separate gates). The full 32-bit ALU then checks for a ZERO condition by connecting the result bits to a 32-input NOR gate, which gives a one only when every single bit of the results is zero. We also ensured the ALU checks for overflow by comparing the carry-in and carry-out of the most significant bit during arithmetic operations: if they are the same, there is not an overflow; otherwise, there is an overflow. Additionally, there cannot overflow when performing one of the logical operations. For the first one-bit ALU, we wired the carry-in of the adder subtractor unit to invB, which satisfies the two's complement definition for $B$ to be

$$-B = \overline{B} + 1 \tag{1}$$

For the last one-bit ALU in our larger 32-bit ALU, we also explicitly wired the sum of the adder/-subtractor through an XOR gate with overflow bit. The output of the overflow would then be connected to the SLT input on the first one-bit ALU. The SLT inputs on all the other ALU bit slices are all wired to zero at all times.

During the later simulation in waveforms, we discovered that overflow would sometimes jump too high during logic operations. As a result, we implemented the following boolean expression to suppress any true output on the overflow output.

$$overflow_{out} = overflow(\overline{cmd_2}\ cmd_1\ cmd_0 + \overline{cmd_2}\ \overline{cmd_1}\ cmd_0 + \overline{cmd_2}\ \overline{cmd_1}\ \overline{cmd_0}) \tag{2}$$

$$overflow_{out} = overflow(\overline{cmd_2}\ cmd_1\ cmd_0 + \overline{cmd_2}\ \overline{cmd_1}) \tag{3}$$

where *cmd* stands for one binary bit of the mux selection command. Thus, 011, 001, and 000 correspond to the commands SLT, Subtract, and Add, respectively.

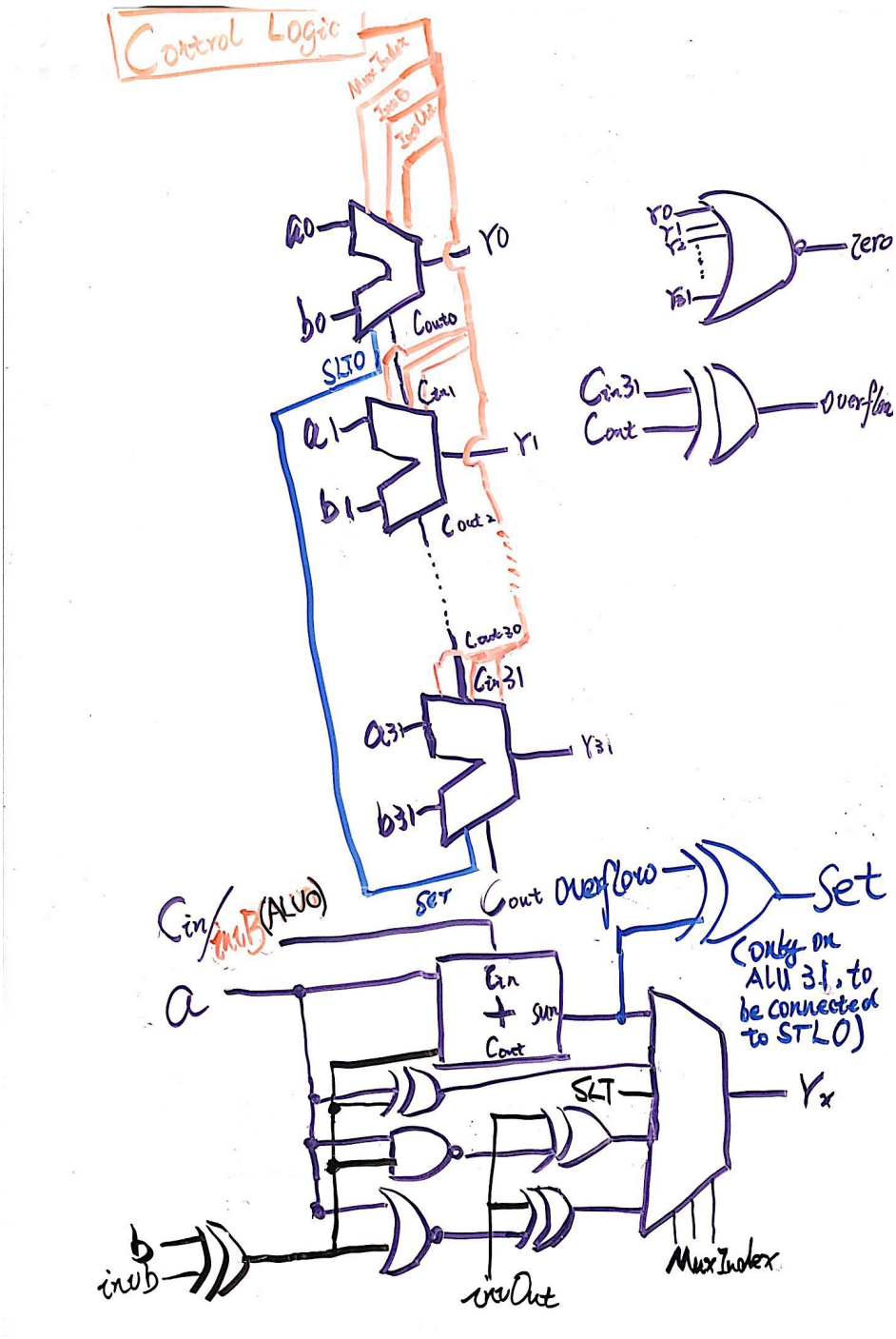


Figure 1: The block diagram of our 32-bit ALU design.

## 2 Test Case Strategy

The following equations display the test cases we chose for our adder and subtractor modules. The modules are identical, with the only difference being that for subtraction, we invert the second input and utilize a carry-in of one.

1. The "zero" case:

$$0_{10} + 0_{10} = 0_{10} \tag{4}$$
$$0_{10} - 0_{10} = 0_{10} \tag{5}$$

2. The "all one" case:

$$-1_{10} + -1_{10} = -2_{10} \tag{6}$$
$$-1_{10} - -1_{10} = 0_{10} \tag{7}$$

3. Adding a positive integer to a positive integer without overflowing; Subtracting a positive integer from a positive integer:

$$2312_{10} + 123_{10} = 2435_{10} \tag{8}$$
$$4_{10} - 2_{10} = 2_{10} \tag{9}$$
$$550_{10} - 1450_{10} = -900_{10} \tag{10}$$

4. Adding a positive integer to a negative integer to get a positive integer without overflowing; Subtracting a positive integer from a negative integer:

$$4_{10} + -2_{10} = 2_{10} \tag{11}$$
$$-231_{10} - 43_{10} = -274_{10} \tag{12}$$

5. Adding a positive integer to a negative integer to get a negative integer without overflowing; Subtracting a negative integer from a positive integer:

$$5_{10} + -7_{10} = -2_{10} \tag{13}$$
$$2500_{10} - -324_{10} = 2824_{10} \tag{14}$$

6. Adding a negative integer to a negative integer to get a negative integer without overflowing; Subtracting a negative integer from a negative integer:

$$-5_{10} + -7_{10} = -12_{10} \tag{15}$$
$$-550_{10} - -1450_{10} = 900_{10} \tag{16}$$
$$-1450_{10} - -550_{10} = -550_{10} \tag{17}$$

7. Adding a positive integer to a negative integer to get zero; Subtracting a positive integer from a positive integer to get zero:

$$1351_{10} + -1351_{10} = 0_{10} \tag{18}$$
$$100_{10} - 100_{10} = 0_{10} \tag{19}$$

8. Adding zero to a positive integer to get back the same positive integer; Subtracting zero from a positive integer:

$$1321_{10} + 0_{10} = 1321_{10} \tag{20}$$
$$107_{10} - 0_{10} = 107_{10} \tag{21}$$

9. Adding zero to a negative integer to get back the same negative integer; Subtracting zero from a negative integer:

$$-1213_{10} + 0_{10} = -1213_{10} \tag{22}$$
$$-3421_{10} - 0_{10} = -3421_{10} \tag{23}$$

10. Adding a positive integer to another positive integer to get a positive integer with overflowing:

$$2147483647_{10} + 1_{10} = 2147483648_{10} \tag{24}$$

11. Adding a negative integer to another negative to get a negative integer with overflowing:

$$-2147483648_{10} + -1_{10} = -2147483649_{10} \tag{25}$$

For our Set Less Than operator, we tested a different variety of cases, which are shown below. All of the operations below will be in terms of $A$ and $B$, with $A$ being the first operand and $B$ being the second operand

1. A > B, A > 0, B > 0

$$4_{10} < 2_{10} = b'0(>) \tag{26}$$

2. A < B, A > 0, B > 0

$$2_{10} < 4_{10} = b'1(<) \tag{27}$$

3. A = B

$$2_{10} < 2_{10} = b'0(=) \tag{28}$$

4. A < B, A < 0, B < 0

$$-123_{10} < -21_{10} = b'1(<) \tag{29}$$

5. A > B, A < 0, B < 0

$$-23423_{10} < -43213_{10} = b'0(>) \tag{30}$$

6. A > B, A > 0, B < 0

$$23423_{10} < -43213_{10} = b'0(>) \tag{31}$$

7. A < B, A < 0, B > 0

$$-23423_{10} < 43213_{10} = b'1(<) \tag{32}$$

8. Overflow tests

$$2147483647_{10} < -3_{10} = b'0(>) \tag{33}$$
$$-2147483648_{10} < 30_{10} = b'1(<) \tag{34}$$

Finally, we tested the logic operations, which include XOR, NAND, NOR, AND, and OR. We utilized the same test cases, which are shown below in hexadecimal, for all of these operations.

1. Cases:

$$A = F0F0F0F0_{16}, B = F0F0F0F0_{16} \tag{35}$$
$$A = 0F0F0F0F_{16}, B = F0F0F0F0_{16} \tag{36}$$
$$A = 0F0F0F0F_{16}, B = 0F0F0F0F_{16} \tag{37}$$
$$A = FFFFFFFF_{16}, B = FFFFFFFF_{16} \tag{38}$$
$$A = 00000000_{16}, B = 00000000_{16} \tag{39}$$

Since overflow and carry-overs between bits do not impact the results of the logical operations, these five test cases should cover most scenarios in which neither, one, or both of any individual bits of the inputs to the ALU are b'1 or b'0.

# 3 Timing Analysis

As we implemented our ALU, we often used waveforms produced in GTKWave Analyzer to debug any errors that arose with our code. After fully implementing our ALU, we analyzed the propagation delays to see how long it took for the outputs to change after we altered the inputs to our ALU. In Figures 2 through 9 we show the worst case propagation delays for each of the eight operations our ALU performs. The worst case delay would travel through the adder module of our ALU. As a result, the worst delay for our one bit adder is ninety nanoseconds, as it travels through a xor gate, an and gate, and an or gate to calculate the carryout. There is also an additional 30 nanosecond xor gate delay when you xor B and invert B before passing the result to the adder. Next, you pass the output of the adder to the multiplexer, which has a worst case delay of 120 nanoseconds, as the input may travel through an inverter, a four input and gate, and a five input or gate. Together, these delays total a delay of 240 nanoseconds per one bit ALU. Because we are utilizing a ripple-carry ALU, we can multiply this delay by 32 to find a total worst case propagation delay of 7,680 nanoseconds for our 32 bit ALU. In practice, the largest delay we saw in our waveforms was only a bit more than 2,000 nanoseconds for the adder.

For NAND/AND and NOR/OR logic units, they have a worst case propagation delay of 80 nanoseconds based on our calculations before passing it into the multiplexer, which has a worst case delay of 120 nanoseconds. The combined total worst case delay is 200 nanoseconds, but what we observed in waveforms is mostly around 150 nanoseconds. For the two input XOR logic gate, which has a 30 nanosecond delay, adding up to a total delay of 150 nanoseconds in the worst case when you take the multiplexer into account. The waveform again confirms our calculation on this.

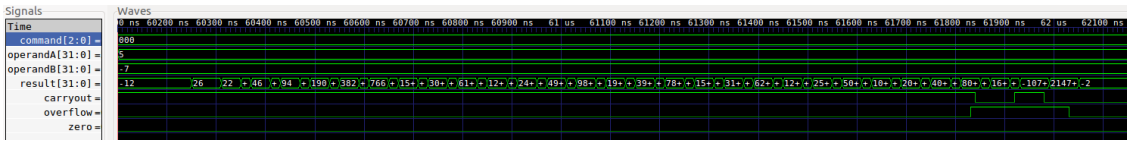


Figure 2: The waveform output from the simulated 32 bit adder

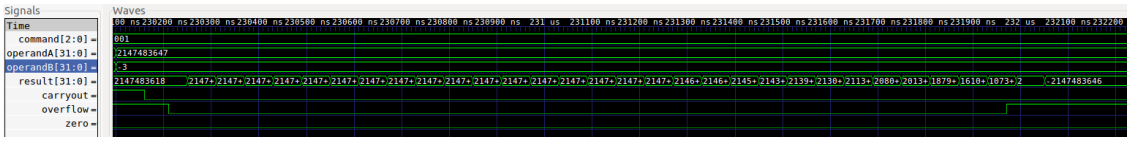


Figure 3: The waveform output from the simulated 32 bit subtractor.

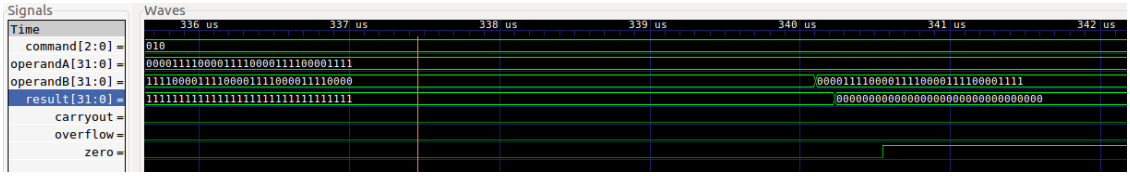


Figure 4: The waveform output from the simulated 32 bit xor gate

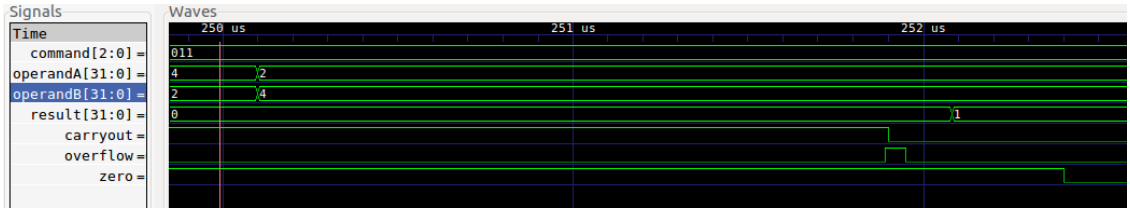


Figure 5: The waveform output from the simulated 32 bit Set Less Than operator

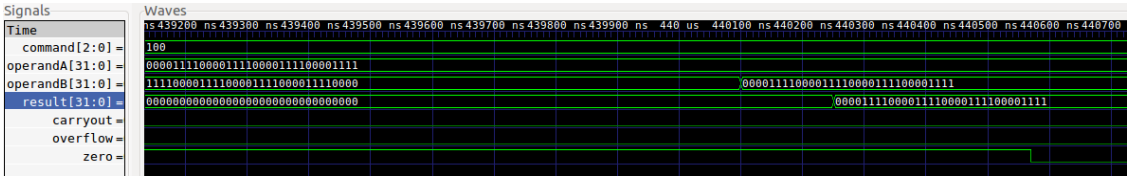


Figure 6: The waveform output from the simulated 32 bit and gate

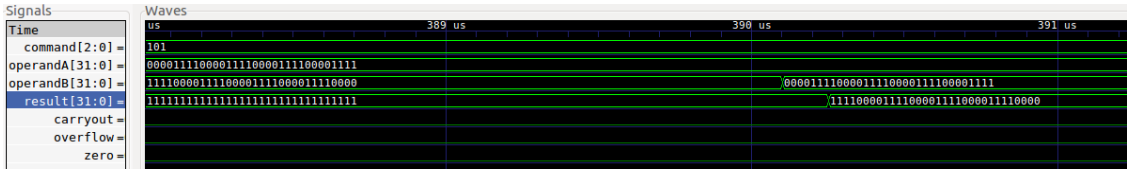


Figure 7: The waveform output from the simulated 32 bit nand gate

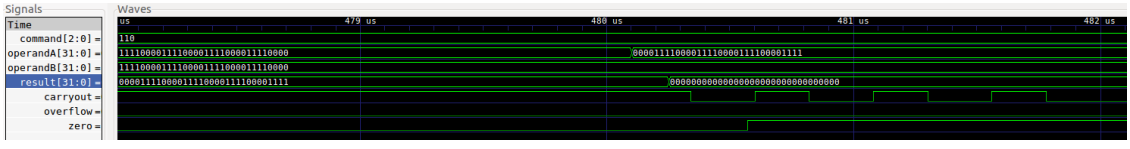


Figure 8: The waveform output from the simulated 32 bit nor gate

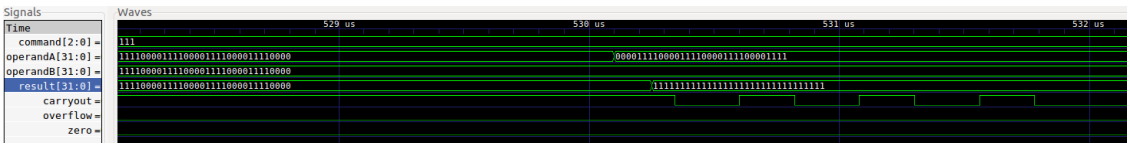


Figure 9: The waveform output from the simulated 32 bit or gate

# 4 Work Plan Reflection

We devised our work plan with the idea that we would start out writing the test bench first. However, we soon realized that it was a way too formidable goal as neither of us have great command of Verilog. As a result, we were unable to test the syntax of our test bench, and especially of the automation of our test bench, as we did not have a device with which to test.

Consequently, we pivoted and started by writing the ALU device itself. Again, we failed to anticipate how much of a barrier Verilog syntax would prove to us. Initially, we had planned to divide up the arithmetic and logical operations into separate thirty-two bit modules before integrating them into a single ALU. In the end, we settled on the simplest designs of thirty-two slices of a one-bit ALU joined together in a ripple-carry style. In the end, the time we spent designing the ALU and writing the test benches greatly exceeded the four hours we expected to put into the design at the beginning. We spent four hours together on Saturday alone, but only ended up with a 32-bit bit slice design that was not even able to add appropriately. We both individually and as a team over the next three days spend on average of three hours each day trying to debug our errors.

Luckily, many of our errors came from typos (typing NAND as AND, declaring a wire with one name but then using a totally different name, etc.) and Verilog syntax(0b'0 is not valid; modules with bus inputs do not accept curly bracket busses , etc.). Propagation delays in the test bench (The ripple-carry style turns out to generate an extraordinarily long delay especially when there's a negative number or a subtraction operation.) were also a source of error. For instance, we were thoroughly convinced that a part of our ALU was not working at one point, and completely redesigned parts of it as a result, when the real error was that the delays in our test bench were not long enough for the result to propagate all the way through the ALU.

By the end of Wednesday night, the device was fully functional and tests were written. We had probably spent close to fifteen hours in total, counting the time we worked individually plus the time we worked as a group. At this point, we still needed to make slight changes to our test bench and write our report. Finally, we were able to wrap up on Verilog and finish our report by late Thursday afternoon. This lab, in retrospect, had probably taken us close to 20 hours of work, which is double the amount of time we expected to spend working on it.