Computer Architechture Lab 1: Arithmetic Logic Unit

Lisa Hachmann & Anisha Nakagawa

October 7, 2016

1 Introduction

We have created an Arithmetic Logic Unit that takes in two 32-bit inputs, performs the specified operation, and returns a 32-bit result with flags for overflow, carryout, zero. The ALU takes in a 3 bit command to specify which operation to calculate according to table 1. For addition and subtraction, the ALU will raise a flag if overflow or carryout occurs. The zero flag outputs true only if the result is zero.

Operation	Command Bit Sequence
ADD	b000
SUB	b001
XOR	b010
SLT	b011
AND	b100
NAND	b101
NOR	b110
OR	b111

Table 1: Control table for ALU operations

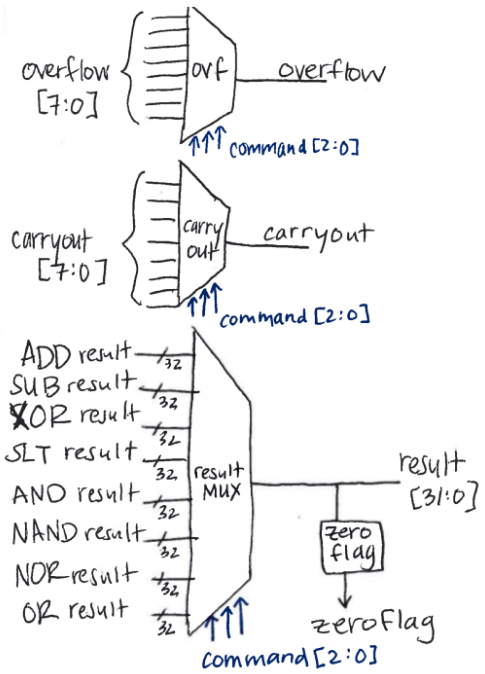

Figure 1: Block diagram of top-level ALU commands

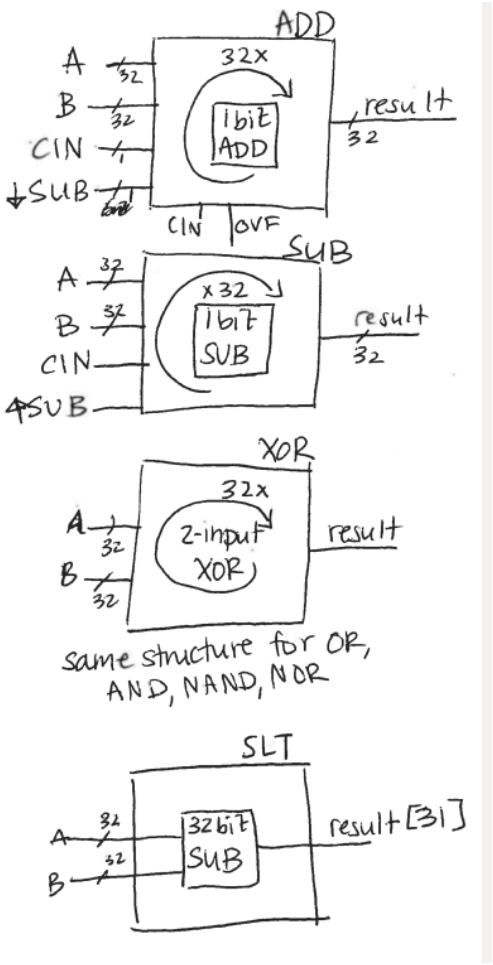

Figure 2: Block diagrams of 3 operands that are stereotypical of the total 8.

The top-level multiplexing of the results of each operand in the ALU can be seen in figure 1. Then the operands are expanded in the block diagram shown in figure 2, which shows 3 cases of the 8 operands. Five of the operands (NAND, NOR, XOR, AND, OR) act very similarly, and so they are all summarized in the XOR diagram.

2 Test Results

Each of the individual modules for the ALU were created separately and tested against a test bench. The test cases were designed to cover the range of possible inputs and operations, including separate categories for cases that result in overflow, carryout, or zero tags. The table of test cases is included in the results.txt file in the repository.

2.1 Test Cases Explanations

In order to test the addition operation, it was important to test cases with overflow and also cases with no overflow. Overflow can occur when adding two positive or two negative numbers that cause the result to be beyond the range of 32-bit 2s complement numbers. There are also three possible cases for no overflow: adding two positive numbers (as long as the result is within range), adding two negative numbers within range, and adding any positive and negative number. Finally, we tested two cases that result in zero. We also selected the cases so that we could confirm the behavior of the carryout flag.

The subtraction module had a similar set of test cases, because it is built using the same module as addition. Therefore, the subtraction test cases also fell into the two main categories of overflow and no overflow. For subtraction, overflow can occur when subtracting two numbers of opposite sign where the result is outside the range of 32-bit 2s complement numbers. The overflow was tested both for positive minus negative, and negative minus positive. We also tested four cases without overflow: positive minus positive, negative minus negative, positive minus negative, and negative minus positive. Finally, we tested cases where the result was zero.

The Set-Less-Than operation compares two numbers, and sets the result to 1 only if input A is greater than input B. This was tested under three scenarios: when A is greater than B, when A is less than B, and when A is equal to B. We wrote two tests for each of these categories. These test cases also confirm that the zero flag works correctly, and that there is no output from the overflow and carryout flags.

The XOR operation should never have overflow and carryout, so we do not need to design test cases explicitly for those. The XOR operation is tested for a case that results in zero in order to test the zero flag, and then three other cases. The cases were chosen so that they demonstrate the full functionality of the XOR in calculating the result for two 0s, two 1s, and one 1 and one 0.

Similarly, the AND and OR gates both had the same test cases. We tested standard cases and also cases where the result was zero. We confirmed the result of the AND and OR gates by comparing it to the expected output of the behavioral logic. The NAND and NOR gates also have very similar logic, so the same test cases were used.

2.2 Test Case Failures and Design Changes

We made some initial design decisions to simplify the implementation and readability of the ALU. We treated each input as a 32 bit number in 2s complement rather than the IEEE 32 bit standard, because this followed the precedent from previously designed modules. We also displayed the elements in the test bench in hex, because it is much more readable than 32 bit binary numbers.

Our first major test case failure occurred in some cases where the most significant bits of the ALU output were different from the expected result. We realized that this occurred because the time delay between the test cases was too short, since the gate delays in a 32 bit adder are much larger than single modules. Therefore we changed out test bench delays to 8000ns.

We also encountered the problem where the overflow and carryout from the adder and subtractor appeared as an "x" instead of a 1. This occurred because we had initially tried to write over the same overflow and carryout wires repeatedly. To fix this, we wrote the overflow and carryout from each operation into different wires in a wire array. We then selected the appropriate wire using a MUX, depending on which operation was being performed.The carryout and overflow MUXes had the same control signals as the top level ALU MUX.

After we had tested all the modules independently, we combined them into the single ALU that selected the appropriate output based on the ALU command. We initially failed our test cases because the wrong operation was being performed for each test case. Eventually we realized that we were processing the bits of the command signal in the wrong order, which fixed this problem.

Finally, we made the design decision to calculate the zero flag after the final result was calculated, in the ALU module rather than each individual module. This mean that we only had to calculate the zero flag once, which takes fewer resources than calculating it within each module. We could have calculated the zero flag with a 32 input or-gate between each bit of the result, but we instead decided to calculate the zero flag iteratively using a loop. We acknowledge that this uses more OR gates, and thus has a greater time delay, but we chose this method because it is easier to re-use this code in the future.

3 Timing Analysis

The worst case propagation delay is through the adder operand. The 32 bits are added or subtracted one at a time, meaning a 1-bit operand needs to be cycled through 32 times. With every 1-bit addition or subtraction, there is a 180 nanosecond (ns) time delay, as there is an XOR gate in front of every "b" input to the adder (for enabling subtraction), plus the path of worst propagation delay of a 1bit full adder (120

ns from 3XOR, AND and OR gates, 2 inputs each,). The timing delay of an XOR gate (60 ns) was found using the circuit in figure 3, which uses NAND gates.

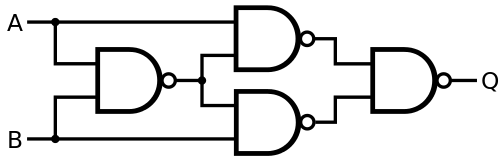

Figure 3: XOR circuit out of NANDs

Therefore, cycling through the 32 bits leads to a 5760 ns delay, but the overflow, carryout and zero flag cases have to be computed after that. The overflow is 60 ns (1 XOR gate), the carryout is 30 ns (1 OR gate), and the zero flag is 970 ns (32 bit cycle of 2 input OR gate). This leads to a total worst propagation delay of 6820 nS. Looking back, a good way to improve this delay is to remove the XOR gate from input B of each 1-bit adder and replacing it with an inverter.

We simulated the ALU in GTKwave and found similar delays on the carryout, sum and flag zero lines, as seen in figure 4. The delays span 74-80.5 µseconds, or around 6.5 µseconds (6500 ns), which is very close to the calculated delay. While the worst delay was determined to be through the carryout of each 1-bit adder, propagating through the 32 bit adder, the sum and zero flag depend on that timing, so they are delayed similarly long.

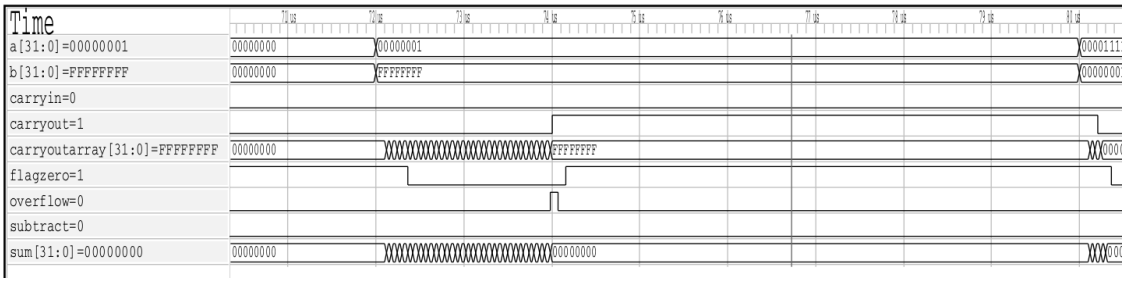

Figure 4: Delay seen on carryout, sum and flag zero lines spanning 74- 80 µseconds

4 Work Plan Reflection

We had estimated the lab take 8.5 hours, and that turned out to be a large underestimation. We spent around 14 hours in this lab, mostly due to the syntax errors of our result multiplexer. By the end, we had drastically improved our verilog skills and were ten times faster than at the beginning, but we expect that this will be the case with most labs. We plan on adding a third member to our team for upcoming labs to make everyone's share of the workload lighter. The categories that we were most off in our estimation were the devices (we hadn't considered zero flag, overflow and carryout separate items), and the combination of devices into the ALU.