CompArch Lab 1: Arithmetic Logic Unit ALU2000

Annie Kroo, Noah Rivkin, Nathan Yee

Ready for use

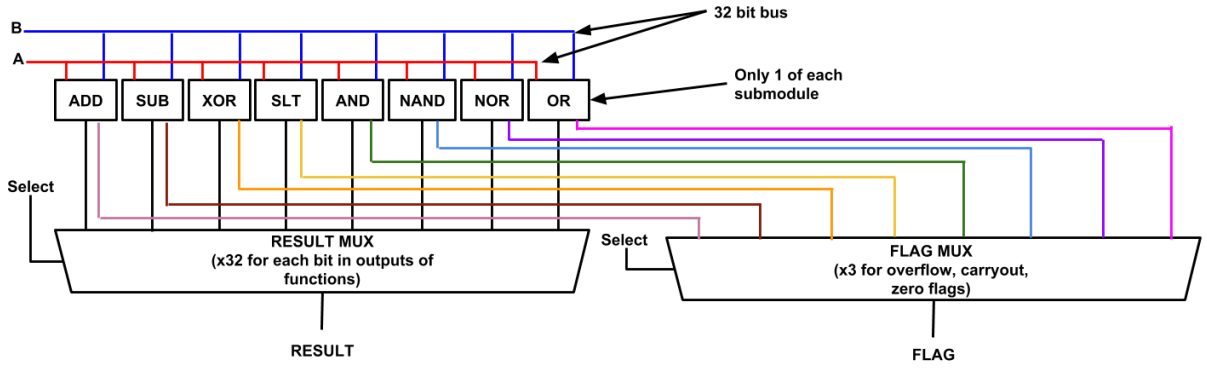
The Arithmetic Logic Unit we designed, the ALU2000, is ready to be included in the next CPU. It is capable of 32 bit operations for 8 different arithmetic operators. It sports a worst case gate delay less than 2000 units allowing for high throughput computation.

The ALU2000 is highly reliable as its functionality has been thoroughly tested. The arithmetic operators, Add, Subtract (Sub), and Set Less Than (SLT) each have their own dedicated test bench that covers all edge cases. These modular submodules were taken from previous designs and are well tested and reliable.

All 8 arithmetic operators of the ALU have comprehensive tests in the ALU test bench. The basic arithmetic operators, XOR, AND, NAND, NOR, and OR are tested with full coverage with all combinations of two binary inputs.

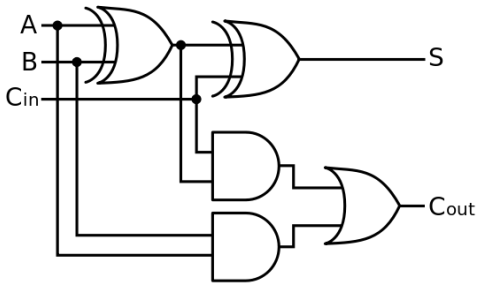
Implementation

We chose to implement our ALU in a modular fashion that uses full 32 bit submodules to calculate each desired arithmetic operator. The 32 bit results of each submodule are split bitwise into 32, 8 to 1 multiplexers that allow each selection of the desired arithmetic operator. We also use 3, 8 to 1 multiplexers, to select the proper flags for the arithmetic operator: overflow, carryout, and zero.

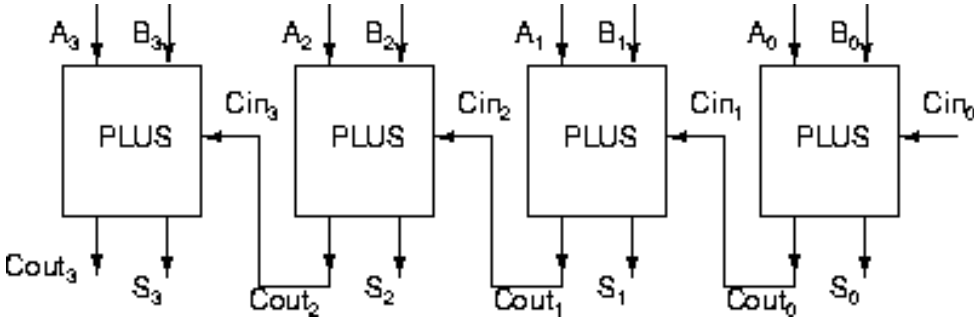


High level gate diagram. The “Flag MUX Thing” selects the correct flags (zero, carryout, and overflow) for Add and Sub and sets the flags to 0 for the other arithmetic operators.

For our bitspliced adder we used the following topology:



This unit of the adder was copied 32 times and connected in the following manner:



(cecs.anu.edu.au)

For our subtractor we used our adder but inverted our B input. This results in the two’s complement of B minus 1. To account for this, we included a carry in for our subtractor. As such we inputted into our adder A and -B.

For our SLT, we used our subtractor. By inputting A and B into our subtractor we were then able to ascertain if A was strictly less than B by checking the sign or most significant bit of our subtracted function. To ensure robustness on cases with overflow, we added an xor gate to the output of the subtractor to xor the MSB of our A-B and the overflow flag.

Test Benches and Test Results

ADD

We used several tests to confirm our adder behaved correctly. We chose the tests to cover different possible combinations of inputs and outputs. In testing the adder, we created a separate test bench so that we could test it on a module level before integrating it into our final ALU. While debugging our adder, our test cases detected a error with our timing delays. By running the adder test bench we noticed that some of our outputs, particularly lower indices

tended to be not connected. This showed us that we needed to change the amount of time the testbench waited before displaying the results after changing the inputs. Our adder test cases were chosen to test a combination of positive and negative numbers with different flag outputs. For example, we looked at cases in which we had two positive inputs with overflow and without and negative and positive cases. In addition, we investigated several zero cases such as when we sum two 0s and where we sum a number with its reciprocal, getting zero.

SUB

Our subtraction test cases were primarily designed to ensure that the flags were operating correctly. Because we utilized our adder module in our subtractor, we first got our adder working with the adder test bench before moving onto the implementation and testing of the subtractor. In the subtractor testbench, we were careful to focus on the edge cases as the majority of the content of the module was already tested in the adder.

SLT

Our SLT was similarly based on our subtractor. Since we validated our subtractor with a testbench, our primary focus when constructing our SLT test bench was to ensure that we handled overflow behavior correctly.

XOR, AND, NAND, NOR, OR

We tested our bitwise functions using a standard test that we used for all of the single gate operations within the higher level module. The simple bitwise operation tests were designed to ensure correct behavior for all possible inputs to a single gate. For example, we were able to in one test case with the test case of 3 and 5 as inputs because each different pairing of bit was represented (ie. As the least significant bits we had two ones, as the second to least significant bits we had a one and a zero, as the third to least significant bits we had the opposite combination of a zero and a one, and as the fourth to least significant bits we had two zeros).

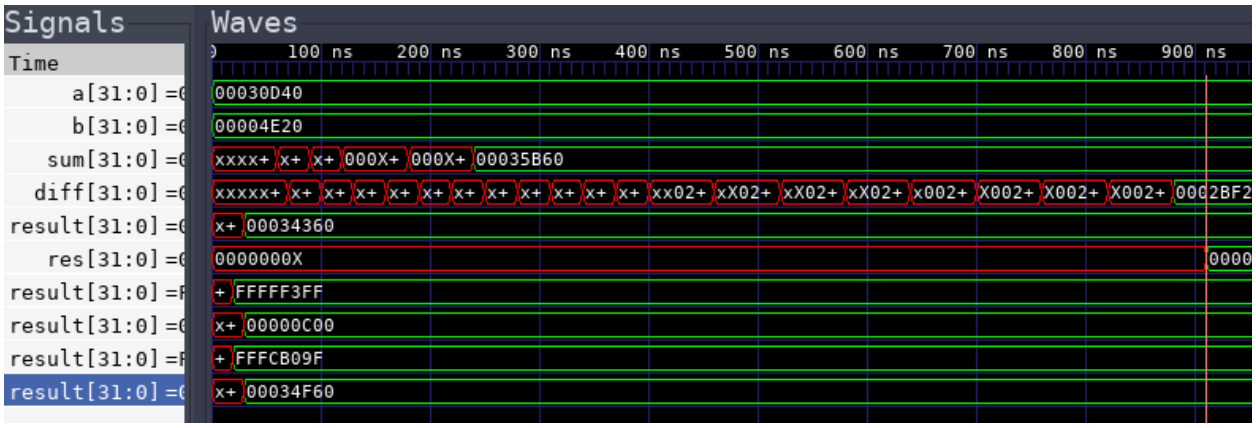
ALU

The ALU calls and tests each of the submodules. This testbench was set up to have a few tests for each module and to index through the modules, testing the overall functionality of the ALU in its ability to switch between different modes of arithmetic.

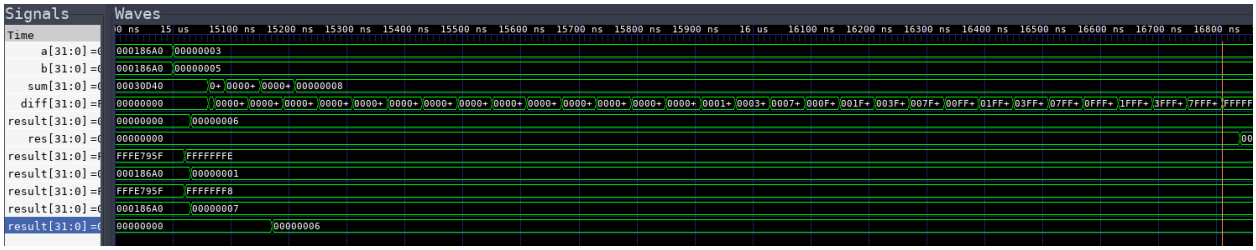
Timing Analysis

We implemented our delays based on a simplified model of our gates. In our model we set the delay caused by a gate equal to the number of inputs into that gate, multiplied by ten. If the gate in question was a positive logic gate, we treat it like its negative counterpart in series with an inverter and add ten units of delay to the gate.

Operator	Simulated Delay
Add	240ns
Sub	1810ns
XOR	30ns
SLT	1840ns
AND	30ns
NAND	20ns
NOR	20ns
OR	30ns



Inputs are a and b. The various sums, diffs, and results are in order of: Add, Sub, XOR, SLT, AND, NAND, NOR, OR.



Worst case for Sub and SLT

Work Plan Reflection

Our work plan did not change significantly during the course of this lab. We were able to maintain a slightly delayed version of our initial schedule, with many of the due dates displaced by around a day.

In hindsight, a bitslice approach would have probably lead to simpler verilog and faster implementation. In the future, we will include some sort of review session for initial gate top level diagram.