CompArch Lab 1

Paige Pfenninger, Lauren Pudvan, Samantha Young

October 5, 2018

1 Implementation

We attempted several different architectures for ALUs. One approach used modules that computed all 32 bits of each mathematical operation and then used look up tables to determine which 32 bit result was the correct result. Another approach was a bitslice ALU that had 32 1 bit ALU modules and a multiplexers in each module for operand selection. We decided to continue with the bitslice approach because we felt like it was easier to follow and involved less repetitive code.

1.1 Bitslice ALU

We took a bitslice approach to the ALU. We created a bitslice ALU module which computed all the necessary calculations on just 2 separate bits, a and b. This module then outputted the result for the selected operation and the carryout that was computed regardless of the operation selected. The bitslice ALU module inputs a bit 'a', a bit 'b', a carry in bit that is used in the adder/subtractor module and a 'is subtraction' bit that determines which operation the adder or subtractor is doing. This module includes all of the operations that the ALU can compute except for the SLT. The SLT is computed later once the entire 32 bit result is found. In order to determine what operation is being selected we decided to use a 3 bit multiplexer (8:1). It takes in 3 select lines s0, s1, s2, and maps them to 8 inputs. These inputs are each wired to an operation result. AND, NAND, NOR, OR, and XOR's results are merely computed by singular gates that are wired to inputs of the multiplexer. ADD, SUBTRACT, and SLT use an adder/subtractor module consisting of a 1 bit full adder that takes a 'is subract' control bit which will inverse the b bit in order to make a subracter. This module is wired to 3 inputs of the multiplexer ADD, SUBTRACT and SLT. This is because the SLT result is later computed from the subtraction result. Finally, the ALU bit slice outputs the bit result of the intended operation and the carryout. The carry out will be used later for future operations and detecting overflow.

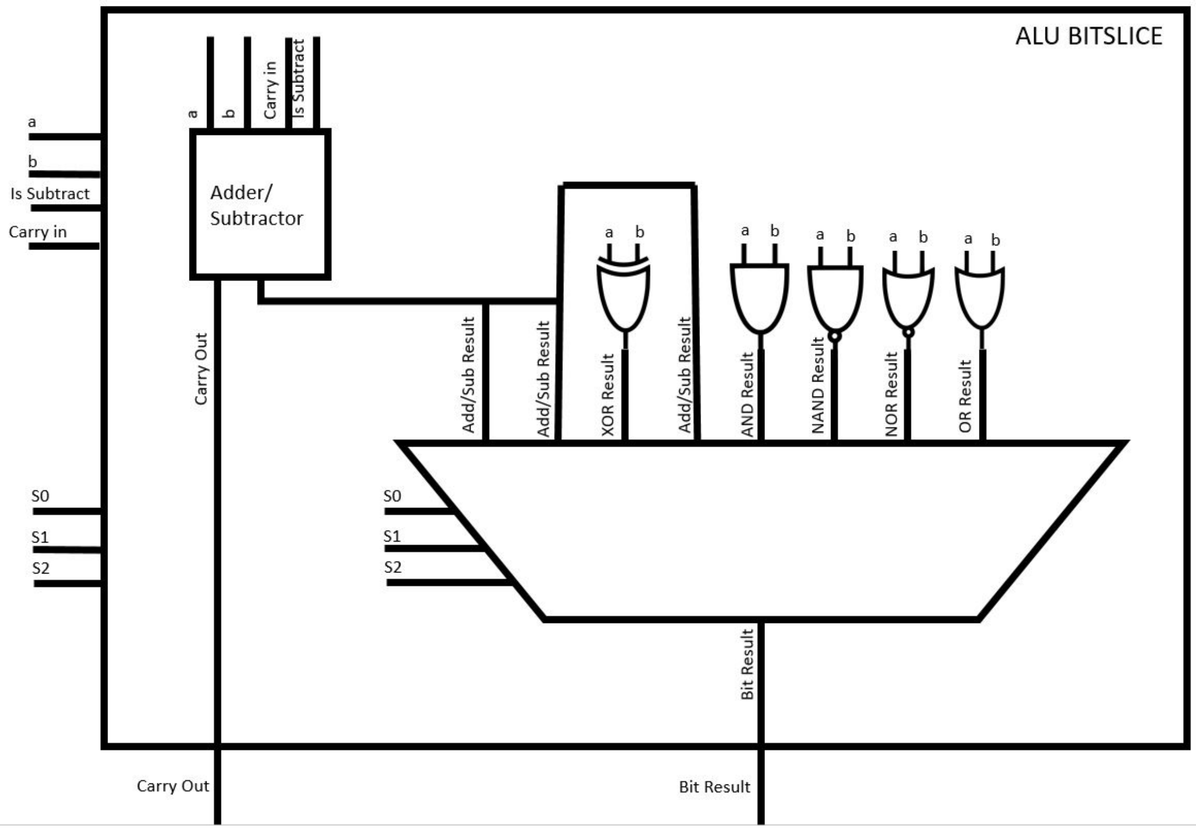

Figure 1: BLock diagram of bitslice ALU module.

1.2 32 bit ALU

The final 32 bit ALU takes two 32 bit numbers, A and B, as well as 3 bit command (s0, s1, s2). The command bits are wired to each bit sliced ALU in order to determine what operation will be computed. There are 32 of the ALU bit slice modules described above,in the final ALU, one for each index of bit. This 32 bit result is then wired through several modules which are used to determine what flags to trigger. Carryout is found by merely setting the carryout of the last bit to the carryout of the entire 32 bit number. If there is carryout this flag is triggered. Zeros is found by sending the 32 bit answer through another module. In this module all of the 32 bits are NOR'd together if any of the bits are a 1 the gate will return zero and the flag will not be triggered. If all of the bits are zero then then the gate will return a one and the flag will in fact be triggered. Like Zeros, overflow is calculated in a separate module. The overflow module looks at most significant bit and checks if it will overflow. Between the ALU bit sliced modules and the final output of the ALU, the 32 bit result passes through an SLT module. If SLT is not selected as per the command bits, this module merely passes the result through to the output of the ALU. However if SLT is selected, the module takes the outputs of the the subtractor and computes SLT. It is then passed to the output of the full ALU.

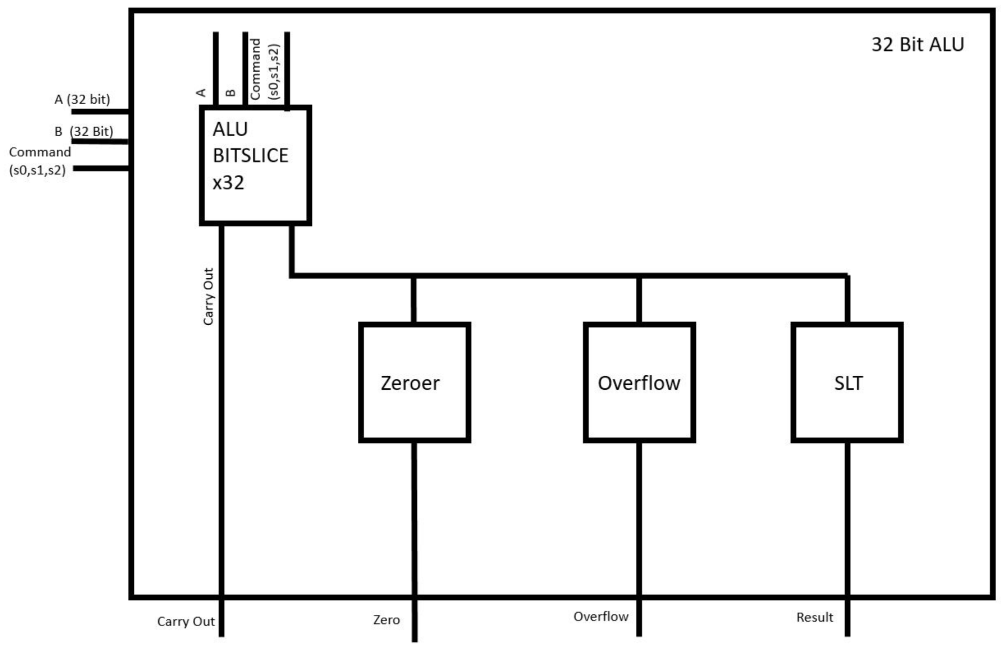

Figure 2: Full ALU diagram

2 Test Results

For the simple logical operations, OR, NOR, AND, NAND, and XOR, we inputted the same A and B. A = 1100, and B = 1010. We chose this A and B because they cover every possible combination ones and zeros for two bits. By inputting A = 1100 and B = 1010, we were able to cover every possible case with just one test per logical operator. The result of the logical operator depends on what the operator is. When combining the A and B stated above, the OR result is 1110, the NOR result is 0001, the AND result is 1000, the NAND result is 0001, and the XOR result is 0110.

For the Addition part of the ALU, we used the same cases that we used for the 4 bit adder, but converted the 4 bit numbers into 32 bit numbers. We defined our test cases assuming signed addition, but the adder also works for unsigned addition (the overflow value becomes irrelevant for unsigned addition) The cases we chose to test were:

1. Positive + Positive with and without overflow
2. Negative + Negative with and without overflow
3. Positive + Negative where the result is 0
4. Positive + Negative where the result is positive (and non-zero)
5. Positive + Negative where the result is negative
6. zero + a number

We believe that the test cases that we chose for the adder cover all of the possible combinations of positive and negative numbers with and without overflow and carryout.

The cases that we chose for the subtractor were very similar to the cases that we chose for the adder. The reason for this is that you can just think of the subtractor as an adder where the sign of B has been flipped. Our most basic test cases were 0 - positive and 0 - negative which will just ensure that our adder is correctly flipping bits. We added a few cases to account for the fact that subtraction is not commutative.

For the SLT, we also tried to get all of the different sign combinations. We assumed that the numbers going into the SLT would always be signed 32 bit integers. For each sign combination (including 0), we also included the the opposite case to get a 0 and a 1 for each number combination so that we know our SLT is working correctly.

To test the zero flag, we added checks for the zero flag on some tests where the zero flag should be 0 and some tests where it should be 1. If there was more than one of the same zero test in the same section we added part numbers to keep track of which tests failed.

The test bench caught a lot of failures in the SLT part of the ALU. Originally, we were not outputting anything from the bit-slice when SLT was selected. After many of the SLT tests failed, we realized that we needed to output the subtract signal of A and B when SLT was selected. Once we fixed that, we were running into an issue where if SLT was false, the ALU would output the result of the subtraction of A and B. This was because we were and-ing the initial results with the inverse of the value of the SLT signal when SLT was selected which. When SLT was false, the inverse was true, so we were and-ing the original signal with true and got the original signal back. To fix this, we made two separate signals, one that was true if SLT was selected and one that was true if SLT was selected and if SLT was true. We first and-ed the entire initial results with the inverse of the is SLT selected to get results. If SLT was selected, this would make the entire results signal zero. We then or-ed the least significant bit of results with the is SLT selected and is SLT true to make the least significant 1. After we fixed the issues with SLT, and the calculation mistakes in our test bench, our code passed our entire test bench.

We also had to increase the waiting time in our test bench to account for the massive increase in time caused by modeling delay based on the number of inputs.

3 Timing Analysis

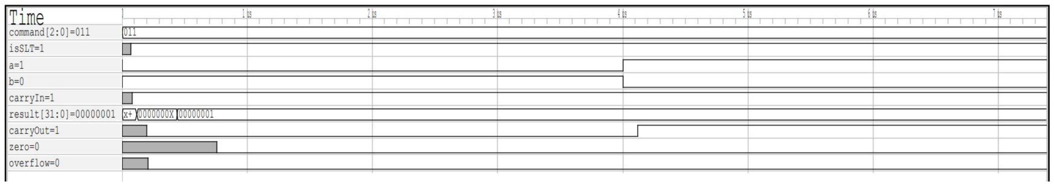

Figure 3: The waveform propagation of the least significant bit for an SLT when a=0 and b=1 then b=0 and a=1. See a larger version of this image on the last page.

This waveform shows 2 test cases. Both are SLTs because the command is 011 for the entire time frame. The first case a = d0 and b = d1. The second test casr is a = d1 and b = d0. We are only seeing the least significant bit because that is the only bit that changes between the test cases. After a delay, the variable that is high when the command is signifying for SLT and carryin, become high. This is because SLT utilizes the subtraction so carryin is set to 1. The result comes in after 0.4us and is the same for both test cases because this is the delay on the test bench. Carry out is 0 for the first test case and 1 for the second. This delay is about 0.25us because carry out is found after the subtract but before the SLT has its result. There are no zeros and overflow for these test cases. We chose these test cases because they show the longest delay. The delay for the subtractor is 240. The delay for the bit slice is 270 and the delay for the SLT and the rest of the alu is 270. This means that the greatest gate delay for the result is 780. The waveform propagation of the the least significant bit for an SLT can be found at the end of the document.

4 Work Plan Reflection

Everything took longer than we expected as per denoted on our work plan. Below was the expected work plan next to the actual times.
Verilog: alu.v - 2.5 hours expected - 10 hours ish actual
ADD / SUBTRACT: 20 minutes expected - 30 minutes actual
XOR - 5 minutes expected - 10 minutes actual
SLT - 30 minutes - 1 hour actual
AND - 5 minutes expected - 10 minutes actual
NAND - 5 minutes expected - 10 minutes actual
NOR - 5 minutes expected - 10 minutes actual
OR - 5 minutes expected - 10 minutes actual
LUT - Multiplexer - Bitsized ALU - controls which one is outputted - 20 minutes expected - 4 hours actual
Debugging - 1 hour expected - 3 hours actual
Test Bench: alu.t.v - 3 hours expected- 4 hours actual Come up with test cases - 1 hour expected - 1 hour actual
Self Checking - 1 hour - 1.5 hour actual
Debugging - less than 1 hour - 1.5 hour actual

We spent a lot of time debugging and trying to make the translation from hardware circuit to Verilog. We sank a lot of time working on an approach that we did not even utilize in the end. Though we did start early, in the future we should plan to start our work earlier start even earlier to create buffer room for these bumps in the road.

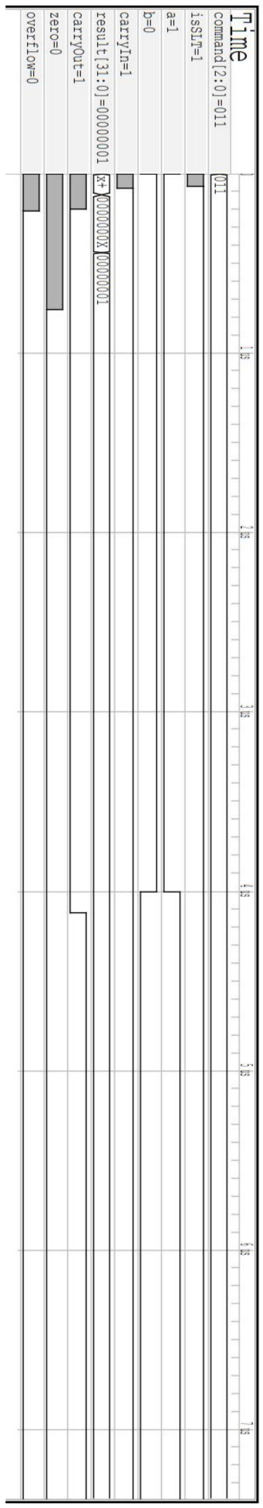

Figure 4: The waveform propagation of the least significant bit for an SLT when a=0 and b=1 then b=0 and a=1.