CompArch Lab 1: ALU

Kimberly Winter, Andrew Pan, and Robbie Siegel

October 13, 2017

1 Introduction

For this lab, we implemented an Arithmetic Logic Unit with eight basic functions -addition, subtraction, exclusive or, greater than, and, not and, or, and nor. We were able to do this by creating separate modules for each component and then combining them in an ALU look up table, controlled by 3 inputs. Because of this structure, we were able to easily test each individual part separately to make sure that each component was working. Then, we were able to create a final ALU.

2 Implementation

We chose to implement our ALU as a lookup table that connected our output to the output of the function selected by the LUT. The LUT that maps the binary commands to operations is shown in Figure 2.

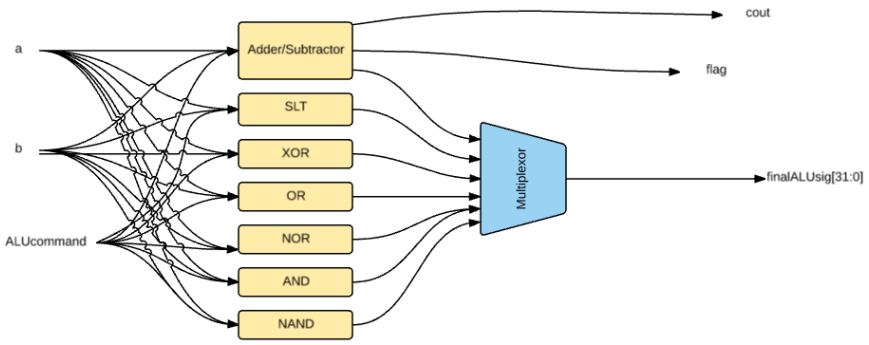

Figure 1: A block diagram of the high-level design of our ALU, including inputs and outputs.

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

Figure 2: The LUT for the ALU's operations

The inputs to the ALU were run through the module for every function, but only the relevant selected function would be reflected in the output. Our adding and subtracting operations used the same module, which was based off of our previous 4-bit adder, and simply inverted the second input to the module if the command bit indicating subtraction was high and added one to our series of adders, producing subtraction. If the command bits indicated only the addition operation, the input were fed through the adders as normal and produced the expected addition.
Our SLT was created by linking single bit comparison modules designed as in the following diagram:

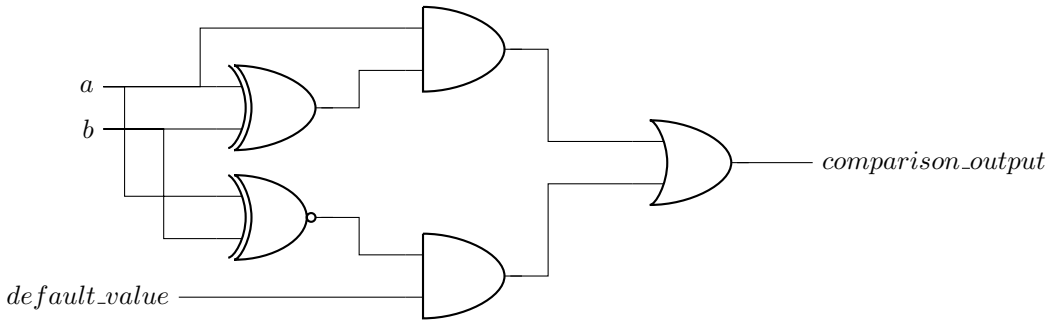

Figure 3: Single bit SLT circuit diagram

The output of the most significant SLT was linked to the output of the 32-bit SLT, and each subsequent less significant bit's SLT was fed into the the and gate of the SLT above it. The least significant bit SLT was fed a 0, because if both lines were the same then there are not other bits to look for, and A == B so 0 should be returned. Additionally, in order to handle comparisons between positive and negative numbers in two's complement, the most significant bit SLT had to be reversed to represent $(B < A)?1 : 0$ because the most significant bit in two's complement is always 1 for negative numbers, and 0 for positive numbers. All of our other functions were based off of pre-existing logic gates, so

we simply applied those single-bit logic gates to each bit of our 32-bit module version of the gate.

3 Test Results

The final test results from our ALU are shown below in Figure 4. The leftmost column represents the operation that was performed on the two inputs, A and B.

ALU Command	Input A	Input B	Output	Flag	Carryout
000	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	0	0
000	00000000000000000000000000000001	00000000000000000000000000000001	00000000000000000000000000000010	0	0
000	01111111111111111111111111111111	01111111111111111111111111111111	11111111111111111111111111111110	1	0
000	10000000000000000000000000000001	10000000000000000000000000000001	00000000000000000000000000000010	1	1
000	11111111111111111111111111111111	00000000000000000000000000000001	00000000000000000000000000000000	0	1
001	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	0	1
001	00000000000000000000000000000001	00000000000000000000000000000001	00000000000000000000000000000000	0	1
001	00000000000000000000000000000111	00000000000000000000000000000101	00000000000000000000000000000010	0	1
001	00000001000000000000000000000101	00000010000000000000000000000101	11111111000000000000000000000000	0	0
001	11111111111111111111111111111101	11111111111111111111111111111110	11111111111111111111111111111111	0	0
001	11111111111111111111111111111110	11111111111111111111111111111000	00000000000000000000000000000110	0	1
001	11111111111111111111111111111101	00000000000000000000000000000101	11111111111111111111111111111000	0	1
001	10000000000000000000000000000101	01111100000000000000000000000000	00000100000000000000000000000101	1	1
001	00000000000000000000000000000101	11111111111111111111111111111111	00000000000000000000000000000110	0	0
001	01111111111111111110111111111111	10000000000000001100000000000000	11111111111111110011101111111111	1	0
010	00000000000000000000000000000000	11111111111111111111111111111111	11111111111111111111111111111111	0	0
010	11111111111111111111111111111111	00000000000000000000000000000000	11111111111111111111111111111111	0	0
010	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	0	0
010	11111111111111111111111111111111	11111111111111111111111111111111	00000000000000000000000000000000	0	0
011	00000000000000000000000000000010	00000000000000000000000000000001	00000000000000000000000000000000	0	0
011	00000000000000000000000000000001	00000000000000000000000000000010	00000000000000000000000000000001	0	0
011	10000000000000000000000000000001	10000000000000000000000000000010	00000000000000000000000000000001	0	0
011	10000000000000000000000000000010	10000000000000000000000000000001	00000000000000000000000000000000	0	0
011	00000000000000000000000000000001	10000000000000000000000000000010	00000000000000000000000000000000	0	0
011	10000000000000000000000000000001	00000000000000000000000000000001	00000000000000000000000000000001	0	0
100	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	0	0
100	00000000000000000000000000000000	11111111111111111111111111111111	00000000000000000000000000000000	0	0
100	11111111111111111111111111111111	11111111111111111111111111111111	11111111111111111111111111111111	0	0
101	00000000000000000000000000000000	00000000000000000000000000000000	11111111111111111111111111111111	0	0
101	00000000000000000000000000000000	11111111111111111111111111111111	11111111111111111111111111111111	0	0
101	11111111111111111111111111111111	11111111111111111111111111111111	00000000000000000000000000000000	0	0
110	00000000000000000000000000000000	00000000000000000000000000000000	11111111111111111111111111111111	0	0
110	00000000000000000000000000000000	11111111111111111111111111111111	00000000000000000000000000000000	0	0
110	11111111111111111111111111111111	11111111111111111111111111111111	00000000000000000000000000000000	0	0
111	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	0	0
111	00000000000000000000000000000000	11111111111111111111111111111111	11111111111111111111111111111111	0	0
111	11111111111111111111111111111111	11111111111111111111111111111111	11111111111111111111111111111111	0	0

Figure 4: Our final test cases for each command of our ALU

We first tested each operation in isolation from the ALU to determine they were working. We then tested the operations going through the ALU.

3.1 Test Case Design

3.1.1 Addition

Our test cases for addition were similar to those we chose in Lab 0. We had base cases of $0 + 0$ and $1 + 1$. From there, we included cases to test for overflow with both two positive and two negative addends. We also included a test case with no overflow but with a carryout.

3.1.2 Subtraction

We used a similar approach to design test cases for subtraction as we did for addition. As with addition, we included the base cases of $0 + 0$ and $1 + 1$. We included test cases in which the input A was greater than B and vice versa for both positive and negative numbers. Overflow could only occur in the case that one operand was negative while the other was positive when doing subtraction. Therefore, we tested positive addends A and negative addends B that resulted in overflow and no overflow. In addition, we tested positive addends A and negative addends B that resulted in overflow and no overflow.

3.1.3 SLT

For SLT operations, six test cases were used. We tested when the first input was greater than the second input in three cases, where all were negative, all were positive, and the first input was positive and the second input was negative. Then we tested when the second input was greater than the first input when all inputs were negative, then all positive, then the second was positive and the first input was negative. This covers all edge cases for the SLT gates.

3.1.4 Basic Gates

It was simpler to design test cases for the and, nand, xor, nor, and or operations. Since the carryout and overflow flag would always be 0 for these operations assuming proper implementation. Because each of these operations was implemented by using identical gates on each of the 32 bits, we only implemented three test cases for each operation. For our first test case, we set both inputs A and B to all zero bits. For our second test case, we set A to all zero bits and B to all one bits. Finally, we set both A and B to all one bits for the third test case for each gate. These cases cover all three unordered possible inputs to each of these gates.

3.2 Errors and Test Redesign

We tested each subsection of our ALU functions individually with their own test benches before combining them into the ALU module. All of our individual functions worked, but we had issues integrating them all into the ALU module. After significant debugging, we found that we had neglected to include changes to our 32-bit inputs in our *always* statement for the ALU lookup table. After that error was fixed, all our of outputs for the test bench appeared to work correctly.

Following this realization of our mistake, we decided to make some changes to our test cases. Initially, we would always change the command and then the inputs before running a new test. Because our error arose since we were not updating our ALU's output upon change in inputs A and B, we decided to ensure that our ALU would respond given a change in the command. Therefore,

we added tests that would not change the two 32 bit inputs but would change the command determining the operation.

4 Timing Analysis

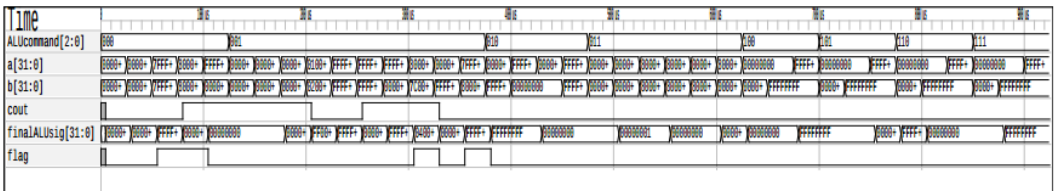

Figure 5: A simulation of our test's timing

The above PDF is a simulation of our system running the alu test bench with our delays using gtk wave. Because all of the values are green in our wave, with the exception of the initial values of cout, finalALUsig, and flag, which we would expect to not start with initial values anyway. This validates our choice of timing, because we have given each of our components enough time to complete the calculations.

5 Work Plan Reflection

We underestimated how long many sections of this lab would take. For example, we initially estimated that implementing add, sub, xor, and slt together would take 1 hour. This was not the case. Implementing only our adder/subtracter by itself took over an hour.

However, one lesson that we did learn was to work on test benches after the module has been created. This helped us to debug the original module while it was still fresh in our minds. One thing that we definitely overlooked was debugging our ALU, particularly the look up table aspect of it. Because we had never worked with behavioral Verilog before, we underestimated how confusing the process would be in terms of implementation and debugging.

We spent a lot of time debugging the Verilog code. This was an unexpected hurdle, that almost doubled our implementation time and set us back substantially. Next time we plan out our time, we will provide much more time for debugging than we initially did.