Implementing a 32-bit Arithmetic Logic Unit

Bryan Werth, Wilson Tang

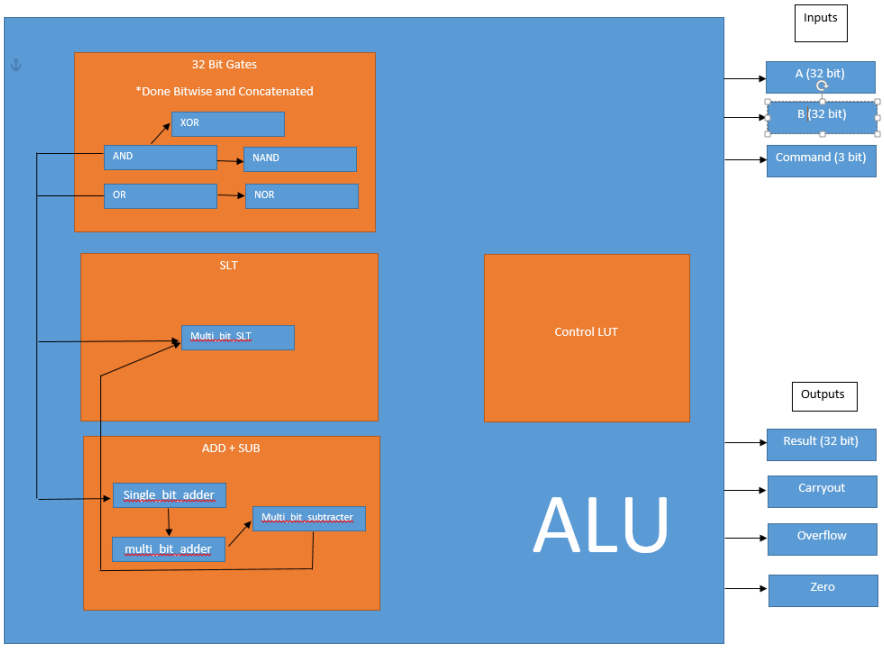
October 2017

1 Introduction

We implemented a 32-bit Arithmetic Logic Unit(ALU) in Verilog with reference to the previous lab's 4-bit adder. Given 2 operands and a control signal we implemented a ALU that could do the following operations: ADD,SUB,XOR,SLT,AND,NAND,OR,NOR.

2 Implementation

In implementing the 32-bit ALU, we started on a couple of different designs until we settled on the final build. We started by building a standard single-bit ALU, but when we started to extend the single-bit to create a 32-bit ALU we realized that it was easier to take the single-bit add functionality than to fully integrate the single-bit ALU into the final design. The ADD is an extension of a single-bit adder similar in design to the adder from the first lab. The SUB operation is built using the multi-bit add module. The two's complement of operand B is produced by inverting operand B and adding one to the result using the multi-bit adder. The SLT operation outputs one if B is greater than A. We can solve for this by using the subtract module we built, because if B is greater than A we will get a 1 in the most significant bit (negative) and a 0 if B is less than or equal to A. Then to account for overflow we will xor the most significant bit with the overflow which will give us a1 if A is less than or equal to B, and 0 if B is less than A. For the logic gates (XOR, AND, NAND, NOR, OR), the output is calculated for each bit in the bus in parallel before everything is concatenated together. The NAND and NOR gates are produced by inverting the output of the AND and OR gate modules. See block diagram below:



3 Test Cases

We made different test cases for the ALU operation to verify our all the operations in our ALU: ADD, SUB, XOR, SLT, AND, NAND, OR, NOR.

3.1 ADD Operation

For the ADD operation we tested 5 different cases as seen in the figure below. We wanted to test such that the carryout, overflow, zero, and final results would be varied. We chose cases with all zeroes to test for zero, cases with 1's placed such that we can test if the carryout would be carried throughout the entire case, and finally a random case to test the sum result.

```
//Add
command = 0; operandA = 0; operandB = 0; #2910
checkTestCase(0, 0, 0, 1);
command = 0; operandA = 32'b11111111111111111111111111111111; operandB = 32'b11111111111111111111111111111111; #2910
checkTestCase(32'b11111111111111111111111111111110, 1, 0, 0);
command = 0; operandA = 32'b10000000000000000000000000000001; operandB = 32'b01111111111111111111111111111111; #2910
checkTestCase(0, 1, 0, 1);
command = 0; operandA = 32'b10000000000000000000000000000000; operandB = 32'b10000000000000000000000000000000; #2910
checkTestCase(0, 1, 1, 1);
command = 0; operandA = 32'b11010101010101010101011010101010; operandB = 32'b00000001111111111111111101101010; #2910
checkTestCase(32'b11010111010101010101011000010100, 0, 0, 0);
```

3.2 SUB Operation

For the SUB operation we tested 5 different cases. We chose the same test cases as for addition, but with operand B being in two's complement form to test for the same things as above as well as successful conversion to two's complement form.

This was our resulting truth table for the ADD operations:

```
//Sub
command = 1; operandA = 0; operandB = 0; #5800
checkTestCase(0, 0, 0, 1);
command = 1; operandA = 32'b11111111111111111111111111111111; operandB = 32'b00000000000000000000000000000001; #5810
checkTestCase(32'b11111111111111111111111111111110, 1, 0, 0);
command = 1; operandA = 32'b10000000000000000000000000000001; operandB = 32'b10000000000000000000000000000001; #5810
checkTestCase(0, 1, 0, 1);
command = 1; operandA = 32'b10000000000000000000000000000000; operandB = 32'b10000000000000000000000000000000; #5810
checkTestCase(0, 1, 1, 1);
command = 1; operandA = 32'b11010101010101010101011010101010; operandB = 32'b11111110000000000000000010010101; #5810
checkTestCase(32'b11010111010101010101011000010101, 0, 0, 0);
```

3.3 XOR Operation

For the XOR operation we tested 3 different cases like the other gates: One with zeroes, one with 1's, and a 1 random one to stress test it. This was our resulting truth table for the XOR operations:

```
//XOR
command = 2; operandA = 32'b11111111111111111111111111111111; operandB = 32'b00000000000000000000000000000000; #80
checkTestCase(32'b11111111111111111111111111111111, 0, 0, 0);
command = 2; operandA = 32'b11111111111111111111111111111111; operandB = 32'b11111111111111111111111111111111; #80
checkTestCase(32'b00000000000000000000000000000000, 0, 0, 1);
command = 2; operandA = 32'b00101010101010101010111010111101; operandB = 32'b00001010101010110101010010110101; #80
checkTestCase(32'b00100000000000011111010000001000, 0, 0, 0);
```

3.4 SLT Operation

For the SLT operation tested 3 different cases. We chose one in which B¿A, one in which A¡B, and one in which A=B to test for the three possible cases SLT can get.

This was our resulting truth table for the SLT operations:

```
//SLT
command = 3; operandA = 32'b11111111111111111111111111111111; operandB = 32'b00000000000000000000000000000000; #5850
checkTestCase(32'b11111111111111111111111111111111, 0, 0, 1);
command = 3; operandA = 32'b11111111111111111111111111111111; operandB = 32'b11111111111111111111111111111111; #5850
checkTestCase(32'b00000000000000000000000000000000, 0, 0, 1);
command = 3; operandA = 32'b00000000000000000000000000000000; operandB = 32'b11111111111111111111111111111111; #5850
checkTestCase(32'b00000000000000000000000000000000, 0, 0, 0);
```

3.5 AND Operation

For the AND operation tested 3 different cases like the other gates: One with zeroes, one with 1's, and a 1 random one to stress test it.

This was our resulting truth table for the AND operations:

```
//AND
command = 4; operandA = 32'b11111111111111111111111111111111; operandB = 32'b00000000000000000000000000000000; #40
checkTestCase(32'b00000000000000000000000000000000, 0, 0, 1);
command = 4; operandA = 32'b11111111111111111111111111111111; operandB = 32'b11111111111111111111111111111111; #40
checkTestCase(32'b11111111111111111111111111111111, 0, 0, 0);
command = 4; operandA = 32'b00000000000000011111111111111111; operandB = 32'b01000100000101011101110111011011; #40
checkTestCase(32'b00000000000000011101110111011011, 0, 0, 0);
```

3.6 NAND Operation

For the NAND operation tested 3 different cases like the other gates: One with zeroes, one with 1's, and a 1 random one to stress test it. This was our resulting truth table for the NAND operations:

```
//NAND
command = 5; operandA = 32'b11111111111111111111111111111111; operandB = 32'b00000000000000000000000000000000; #50
checkTestCase(32'b11111111111111111111111111111111, 0, 0, 0);
command = 5; operandA = 32'b11111111111111111111111111111111; operandB = 32'b11111111111111111111111111111111; #50
checkTestCase(32'b00000000000000000000000000000000, 0, 0, 1);
command = 5; operandA = 32'b00000000000000011111111111111111; operandB = 32'b01000100000101011101110111011011; #50
checkTestCase(32'b11111111111111110001000100100100, 0, 0, 0);
```

3.7 OR Operation

For the OR operation tested 3 different cases like the other gates: One with zeroes, one with 1's, and a 1 random one to stress test it.

This was our resulting truth table for the OR operations:

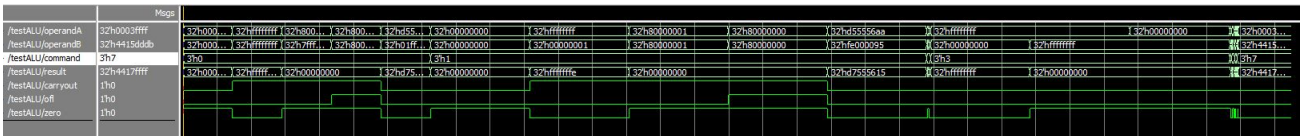
```
//OR
command = 7; operandA = 32'b11111111111111111111111111111111; operandB = 32'b00000000000000000000000000000000; #40
checkTestCase(32'b11111111111111111111111111111111, 0, 0, 0);
command = 7; operandA = 32'b00000000000000000000000000000000; operandB = 32'b00000000000000000000000000000000; #40
checkTestCase(32'b00000000000000000000000000000000, 0, 0, 1);
command = 7; operandA = 32'b00000000000000011111111111111111; operandB = 32'b01000100000101011101110111011011; #40
checkTestCase(32'b01000100000101111111111111111111, 0, 0, 0);
```

3.8 NOR Operation

For the NOR we tested 3 different cases like the other gates: One with zeroes, one with 1's, and a 1 random one to stress test it. This was our resulting truth table for the NOR operations:

```
//NOR
command = 6; operandA = 32'b11111111111111111111111111111111; operandB = 32'b00000000000000000000000000000000; #50
checkTestCase(32'b00000000000000000000000000000000, 0, 0, 1);
command = 6; operandA = 32'b00000000000000000000000000000000; operandB = 32'b00000000000000000000000000000000; #50
checkTestCase(32'b11111111111111111111111111111111, 0, 0, 0);
command = 6; operandA = 32'b00000000000000011111111111111111; operandB = 32'b01000100000101011101110111011011; #50
checkTestCase(32'b10111011111101000000000000000000, 0, 0, 0);
```

4 Timing Analysis



The longest delay would be the SLT operation which is a SUB operation, which is equal to 2 ADD operations, plus an XOR gate: 5,830 units of time

The shortest delay would be the XOR operation which is 60 units of time.

5 Work Plan Reflection

Task	Project Time Spent and Done By	Actual Time Spent and Done By
Single-Bit ALU Implementation	1-2 Hrs by 10/7	1 Hr by 10/7
Single-Bit ALU Test Bench and Debugging	1 Hr by 10/7	Not Done
Full ALU Implementation	0.5 Hr by 10/9	2-3 Hr by 10/12
Full ALU Test Bench and Debugging	3-4 Hr by 10/9	8-9 Hr by 10/12
Polishing and Report	2-3 Hr by 10/11	1-2 Hr by 10/12

We started off strong with the single-bit version but failed to appropriately finish the self checking aspect of the test bench and decided to move onto the full ALU implementation. Most the core calculations were completely quickly (albeit with multiple syntax errors), with the ADD, SUB, and SLT functions requiring actual modules. Most of our delays were in getting the Control LUT aspect of the ALU working as well as debugging the syntax errors in our code.

*We initially planned for implementation onto the FPGA but realized later that we didn't need to do that

5.1 Test Bench Failures

We had no test bench failures that were a result of our function failures. We had multiple errors in properly choosing and passing the results through the Control LUT and into the Test Bench.