Report - Comp Arch Lab 1

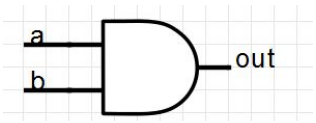
Jessica Diller, Marie-Caroline Finke, and Griffin Tschurwald

Implementation

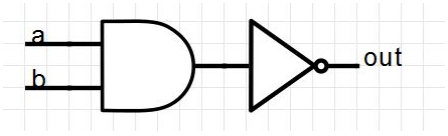
We took a bit-slice approach for all our implementation.

For the AND, OR, NAND, NOR, and XOR operations, we put a gate on each of the first, second, etc. bits of the two inputs with the output of each gate connected to the first, second, etc. bit of the 32-bit long output number.

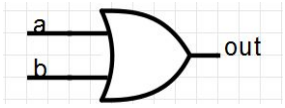
AND:



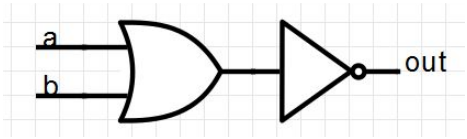
NAND:



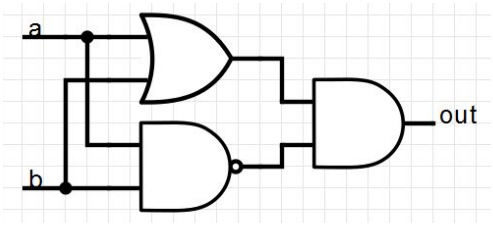
OR:



NOR:

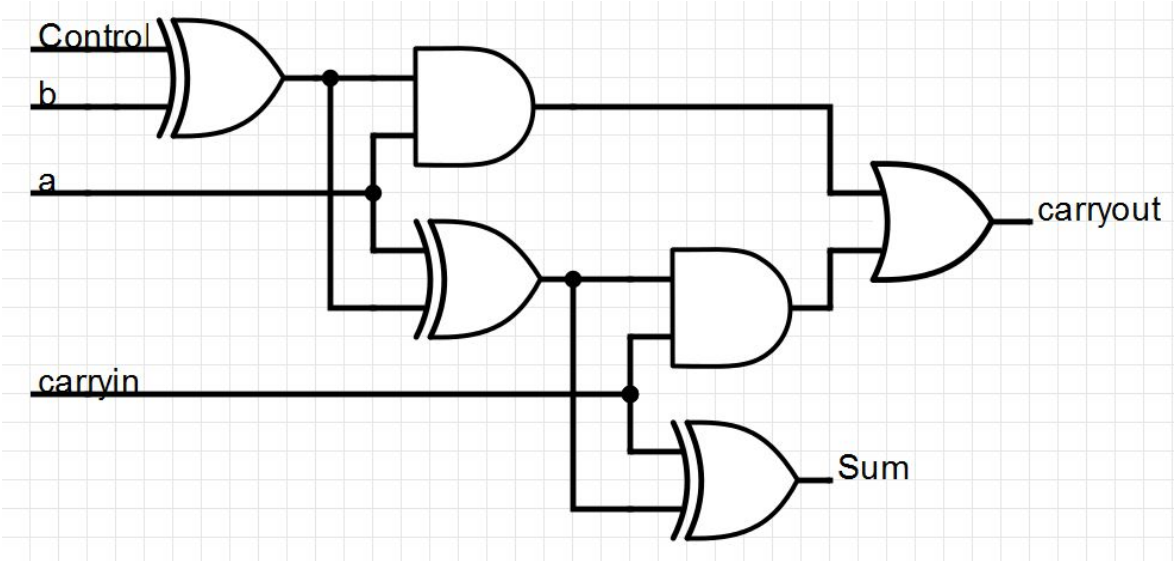


XOR:

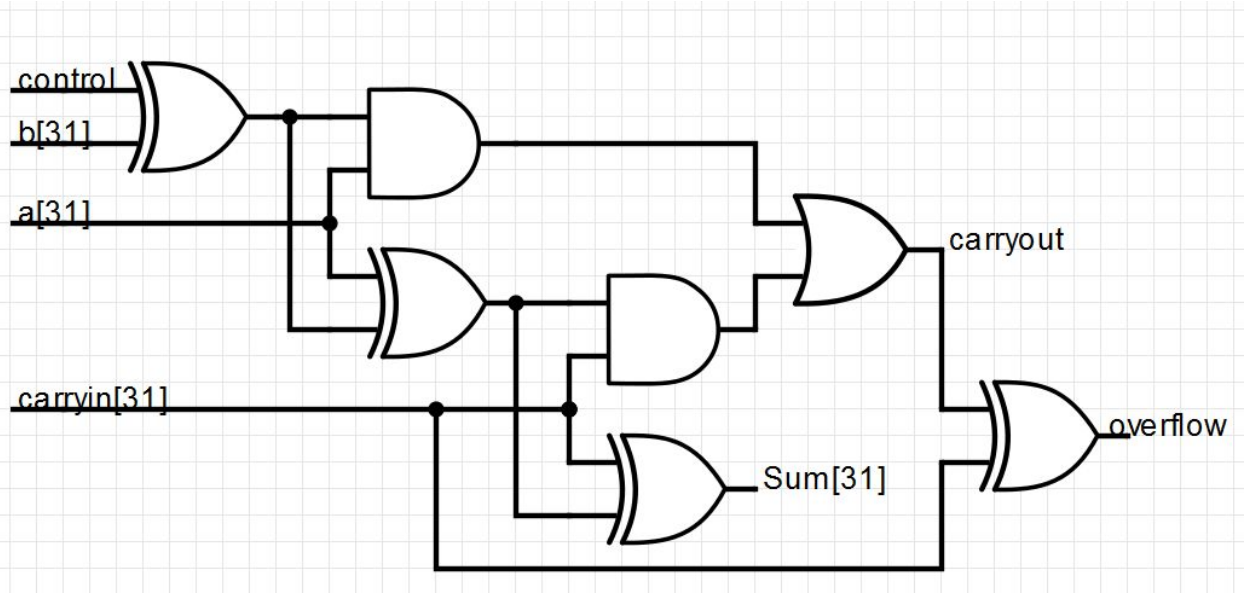


For ADD, we used the code and implementation of the previous lab. For each bit, we had a control and each put into an two different XOR gates to determine whether the ALU was selecting ADD or SUB. If the control is 0, ADD is selected. We started with a one-bit adder and then connected each carryout to the carryin of the next one-bit adder. We expanded that up to 32 bits.

General Structure for ADD/SUB for 1 bit:



Final bit of ADD/SUB:

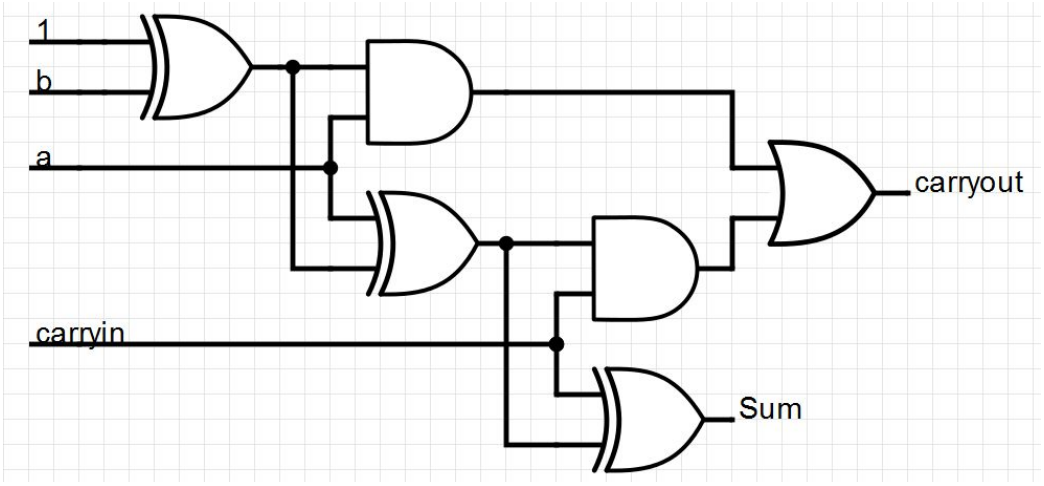


For SUB, we used the same module as for the adder. The control we input selects between adding and subtracting. When the control = 0 it adds, when the control = 1 it subtracts. This works as the XOR gate will only invert the number of the control is equal to 1. We also set the first carryin to equal the value of the control so that if we are

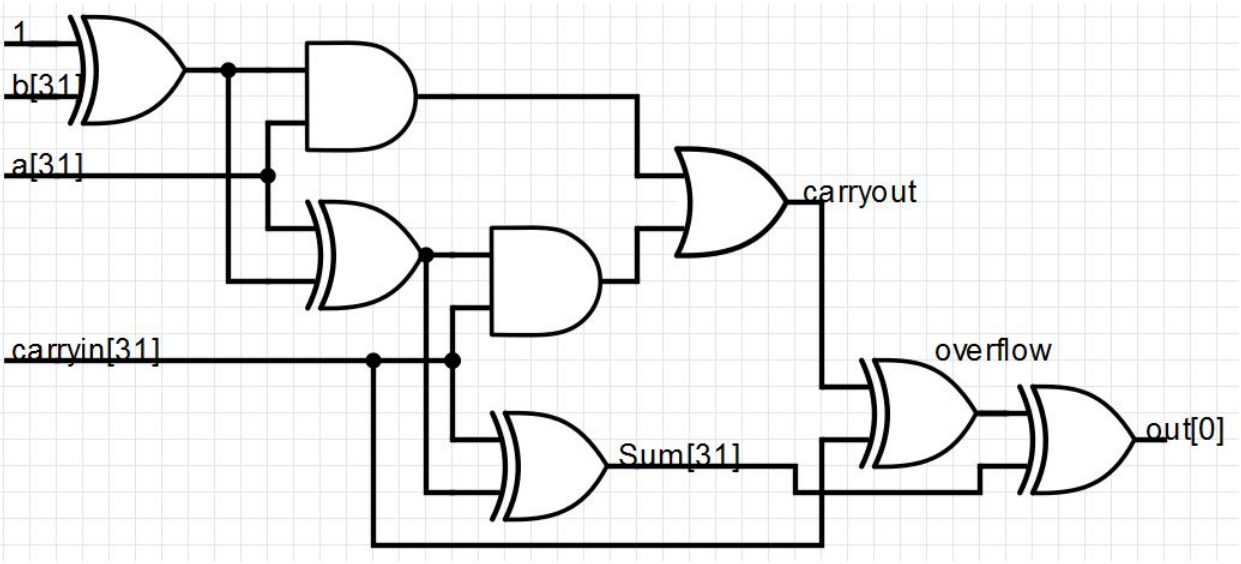
subtracting 1 gets added to finish inverting the number for two's complement. This control signal is generated by our lookup table.

For the SLT, we first thought we'd use a subtractor and depending on the most-significant bit value, we'd know if it was the first number was less than the second. If the most-significant bit of the output was one, the output number was negative, which meant that the second number is greater than the first. If the most-significant bit of the output was zero, the output number was positive, which meant that the second number is less than the first. However, upon further thought, we found there may be inputs that would cause the most-significant bit to not give the expected result to us as an output. We realized that in all cases where there was overflow to the adder the most-significant bit gave the opposite of expected output. We solved this by adding an XOR gate at the end of the SLT with two inputs: the most-significant bit and the overflow. In addition, to produce the correct response, we created a new 32 bit number (wire) and put the least-significant bit of that into an OR gate with the final result of the SLT. That will get an output number with the same amount of bits as the inputs, but all zeros except for the least-significant bit, which will depend on the result of the SLT.

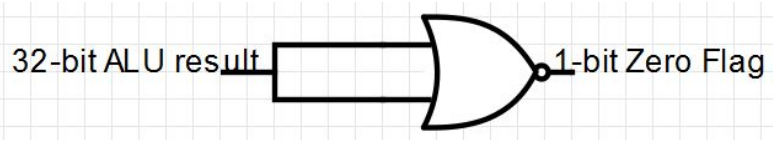
General SLT Circuit for 1 bit:



SLT Circuit for final bit:

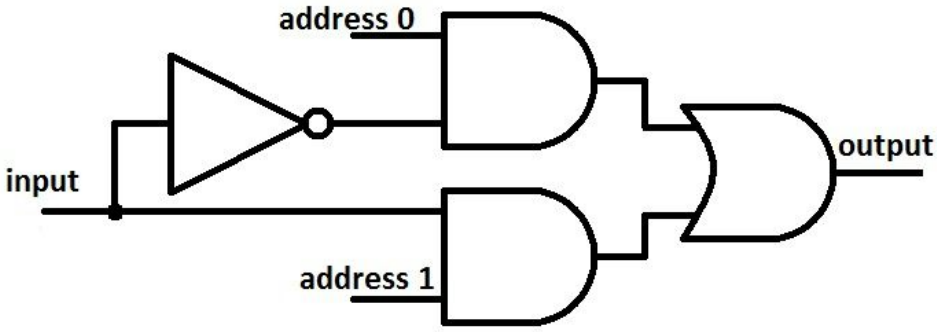


For ZERO we input the final 32bit bus from the ALU through a NOR gate. If the result is 1 all bits of the result were zero.



For the LUT, we instantiated 32 multiplexers and used those to select for each bit. Each MUX was a combination of 3 NOT gates and 6 AND gates with one final OR gate. We took a bit slice approach and reused a lot of our code structure from previous assignments. We wrote a python script to generate all 32 multiplexers instead of writing them out by hand, which also allowed us to go back and make changes to all of them quickly if necessary.

One bit of three-bit MUX:



Test Results

For each ALU operation, include the following in your report: 1. A written description of what tests you chose, and why you chose them. This should be roughly a paragraph or two per operation. 1. Specific instances where your test bench caught a flaw in your design. 1. As your ALU design evolves, you may find that new test cases should be added to your test bench. This is a good thing. When this happens, record specifically why these tests were added.

In order to test the different operations of our ALU, we decided to use a combination of traditional test benches as well as comparing our operations to their behavioral counterparts. We tested each operation extensively outside of the ALU first and then, once we put the code into the ALU, we did some smaller testing to make sure our previously working code was working within the structure of the ALU.

For AND, OR, NAND, NOR, and XOR, we chose four different test benches. As each of the 32 bit slices didn't interact with each other (as in the other operations), we only needed to test differences in input bits.

Our four cases:

- both inputs were all ones
- both inputs were all zeros
- the first input was all ones and the second was all zeros
- the first input was all zeros and the second was all ones

For each operation, we caught no errors in intial testing. We believe the simplicity of these operations structurally caused the ease of implementation. We did these test benches in structural verilog because of how easy it was to determine the expected outputs.

Once we began integrating them into the larger ALU system, we got errors. There was an error on the SLT because we were testing if b was less than a instead of a being less than b, so our behavioral code didn't match.

Test Benches from ModelSim:

```
# this is the SLT operation
# operandA   operandB  | muxindex  invertB  zero | expected   actual
# 0000000f   00000001  |    2         0      0   | 00000001  00000001
# ffffffff   fffffff0  |    2         0      0   | 00000001  00000001
# 0fffffff   f0000000  |    2         0      1   | 00000000  00000000
# cfffffc    cfffffc   |    2         0      0   | 00000001  00000001
# f0000000   f0000000  |    2         0      0   | 00000001  00000001
# ffffffff   ffffffff  |    2         0      0   | 00000001  00000001
# this is the AND operation
# operandA   operandB  | muxindex  invertB  zero | expected   actual
# ffffffff   ffffffff  |    3         0      0   | ffffffff  ffffffff
# 00000000   00000000  |    3         0      1   | 00000000  00000000
# 00000000   ffffffff  |    3         0      1   | 00000000  00000000
# ffffffff   00000000  |    3         0      1   | 00000000  00000000
# this is the OR operation
# operandA   operandB  | muxindex  invertB  zero | expected   actual
# ffffffff   ffffffff  |    6         0      0   | ffffffff  ffffffff
# 00000000   00000000  |    6         0      1   | 00000000  00000000
# 00000000   ffffffff  |    6         0      0   | ffffffff  ffffffff
# ffffffff   00000000  |    6         0      0   | ffffffff  ffffffff
# this is the NAND operation
# operandA   operandB  | muxindex  invertB  zero | expected   actual
# ffffffff   ffffffff  |    4         0      1   | 00000000  00000000
# 00000000   00000000  |    4         0      0   | ffffffff  ffffffff
# 00000000   ffffffff  |    4         0      0   | ffffffff  ffffffff
# ffffffff   00000000  |    4         0      0   | ffffffff  ffffffff
# this is the NOR operation
# operandA   operandB  | muxindex  invertB  zero | expected   actual
# ffffffff   ffffffff  |    5         0      1   | 00000000  00000000
# 00000000   00000000  |    5         0      0   | ffffffff  ffffffff
# 00000000   ffffffff  |    5         0      1   | 00000000  00000000
# ffffffff   00000000  |    5         0      1   | 00000000  00000000
# this is the XOR operation
# operandA   operandB  | muxindex  invertB  zero | expected   actual
# ffffffff   ffffffff  |    1         0      1   | 00000000  00000000
# 00000000   00000000  |    1         0      1   | 00000000  00000000
# 00000000   ffffffff  |    1         0      0   | ffffffff  ffffffff
# ffffffff   00000000  |    1         0      0   | ffffffff  ffffffff
```

(continues on next page)

```
# this is the ADD operation
# operandA   operandB  | muxindex  invertB  zero | expected  actual   | carryout carryoutB | overflow
# 0000000f   00000001  |    0         0       0  | 00000010 00000010 |     0        0     |    0
# ffffffff   fffffff0  |    0         0       0  | fffffffef fffffffef |     1        1     |    0
# 0fffffff   f0000000  |    0         0       0  | ffffffff ffffffff |     0        0     |    0
# cffffffc   cffffffc  |    0         0       0  | 9ffffff8 9ffffff8 |     1        1     |    0
# f0000000   f0000000  |    0         0       0  | e0000000 e0000000 |     1        1     |    0
# ffffffff   ffffffff  |    0         0       0  | fffffffe fffffffe |     1        1     |    0
# this is the SUB operation
# operandA   operandB  | muxindex  invertB  zero | expected  actual   | carryout carryoutC | overflow
# 0000000f   00000001  |    0         1       0  | 0000000e f800000e |     0        0     |    0
# ffffffff   fffffff0  |    0         1       0  | 0000000f fc00000f |     0        0     |    0
# 0fffffff   f0000000  |    0         1       0  | 1fffffff 1fffffff |     0        1     |    0
# cffffffc   cffffffc  |    0         1       0  | 00000000 ff800000 |     0        0     |    0
# f0000000   f0000000  |    0         1       0  | 00000000 ff000000 |     0        0     |    0
# ffffffff   ffffffff  |    0         1       0  | 00000000 ffc00000 |     0        0     |    0
```

* Subtraction, and Overflow for Addition aren't working. They work in their own program but not within the ALU.

ZERO: The zero gate testing ran into initial problems when first running it (no image was saved) as the Zero Flag was showing up as long as the 1st bit was 0. This was corrected by calling each bit of the bus individually into the NOR gate instead of just calling the bus into the NOR gate as had previously been done. After this change the Zero Flag works as intended. Here is an example of it working within the LUT:

```
the 2 operands are 00000000 and 00000000
muxindex = 011
invertB: 0
the result of the ALU operation is: 00000000
If adding or subtracting the carryout is: 0 and the overflow is: 0
The Zero Flag is: 1
```

Timing Analysis

AND: 20 - for each bit there's one AND gate with two inputs. 2 * 10 = 20. All 32 gates function simultaneously.

OR: 20 - for each bi there's one OR gate with two inputs. 2 * 10 = 20. All 32 gates function simultaneously.

NAND: 30 - for each bit there's one AND gate with two inputs and one NOT gate. 2 *10 + 10 = 30. All 32 of the bit systems function simultaneously.

NOR: 30 - for each bit there's one AND gate with two inputs and one NOT gate. 2 *10 + 10 = 30. All 32 of the bit systems function simultaneously.

XOR: 50 - for each bit there's two levels of timing. The first layer is an OR gate and an AND gate which happen simultaneously. 2 *10 = 20. The AND gate output is put through a NOT gate. 20 +10 = 30. Then the OR gate and the NOT gate output are put into an AND gate. 30 + 2*10 = 50. All 32 of the bit systems function simultaneously.

ADD: 1430 - The first section of timing occurs simultaneously through all 32 bits. The control and b are put into an XOR gate and the result of that and a is put into another XOR gate. 50 + 50 = 100. Then the cascading adding begins. Each bit puts the carryin from the previous bit slice and the output of this second XOR gate into an AND gate and simultaneously puts the outputs of the first XOR gate and a into an AND gate. 100 + 20 * 32 = 740. Then these two outputs are put into an OR gate. 740 + 20 *32 = 1380. Then finally the carryin and carryout of the last adder are put into an XOR gate to determine overflow. 1380 + 50 = 1430.

SUB: 1430 - The SUB is the same timing as the ADD because the control input is the only thing that changes between the two (not the structure).

SLT: 1480 - The SLT is the SUB plus an additional XOR gate that takes the most-significant bit of the final output and the overflow as inputs. 1430 + 50 = 1480.

ZERO: 330 - The Zero Flag is a NOR gate taking a 32-bit bus input into an OR gate NORing the result (32*10)+10 = 330

LUT (MUX): 110 - The LUT is a three level gate system. It begins with NOT gates. 10 = 10. The output of the NOT gates is put into AND gates with three other inputs. 10 + 3 *10 = 40. The output of the AND gates are put into a final OR gate that has seven inputs. 40 + 7 *10 = 110. This is the time delay for each MUX. Since the MUX selection for each bit happens simultaneously and there are no other parts in the LUT, this is the total time delay for the LUT.

Sidenote: Each of the AND, OR, NAND, NOR, XOR, ADD, SUB, and SLT's worst case propagation delay doesn't include the time needed for the ALU to select which operation is occurring. That time is the time delay of the LUT (MUX). Each operation would actually be increased by 110, but to see the differences between operations easily, we didn't add that amount to the results above.

Work Plan Reflection

Task	**Time Estimate**	**Date Estimate**	**Time Actual**	**Date Actual**
Make work plan	1 hour	October 6	45 min	October 6
Think about logic and plan design	2 hours	October 9	2.75 hours	October 13
Decide test cases	15 min	October 10	15 min	October 9
Make test benches	30 min	October 10	7 hours	October 16
Make OR	30 min	October 11	30 min	October 13
Make AND	30 min	October 11	10 min	October 13
Make NOR	30 min	October 11	10 min	October 13

Make NAND	30 min	October 11	10 min	October 13
Make XOR	30 min	October 11	10 min	October 13
Make ADD	1 hour	October 12	1.5 hours	October 14
Make SUB	1 hour	October 12	1 hour	October 14
Make SLT	1 hour	October 12	1 hour	October 14
Make Zero Flag	--------	----------	30min	October 15
design and test control logic LUT	1 hour	October 13	5 hours	October 15
integrating into ALU	---------	----------	12 hours	October 16
timing analysis	1 hour	October 13	1 hour	October 14
write up lab	1 hour	October 13	5 hours	October 16
work plan reflection	15 min	October 14	15 min	October 16

Thoughts:

We thought that we'd be able to fully design the whole system initially and then split up the work based on that. However, we realized the design process was circular and time needed to spend on it throughout the process. This added a lot more time than we initially thought.

We also didn't think about how we would implement our test benches. We have no idea where we got our original 30 minutes estimate from, but whether implemented structurally (typing out 32 bits for each output and expected output) or behaviorally (learning how behavioral code works), test bench implementation would take a while.

Writing up the lab took a lot longer than thought of previously. We had to type up all our reasonings, get photos of relevant tests and such, and draw circuit diagrams.

The design and testing of the control logic LUT took a lot longer because it also included integrating all the aspects we had already made together.

There was some small variation in implementation for various other aspects, but not very much. It would have been hard to guess that before beginning the work. Most of our guesses except those described above were somewhat accurate.