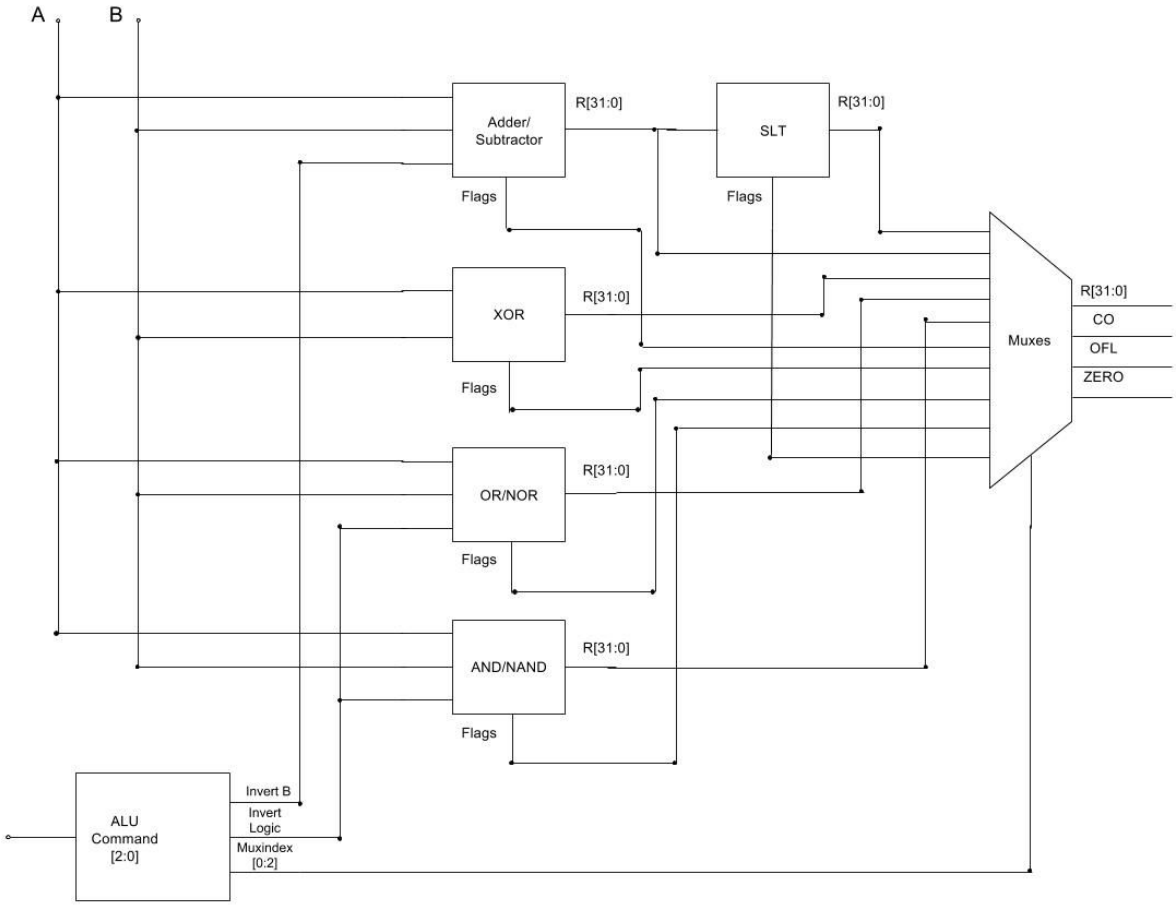
ALU Design

The following ALU design is complete, correct, and ready to be integrated into a CPU system. This ALU takes 32 bit binary numbers and can perform eight operations on them, depending on a command that is passed into them: ADD, SUB, XOR, SLT, AND, NAND, NOR, and OR. Output is the 32 bit result of the selected operation, as well as three flags that are set: carryout, overflow, and the zero flag. These flags are only set/significant when the ADD/SUB function is used, in which case carryout represents the carryout of the operation of the most significant bit, overflow determines whether the carryout is significant/problematic in a potential system, and the zero flag determines if the result is all zeros. In order to test the ALU, test cases were chosen for robust yet efficient testing on all different operations. Then, timing analysis was done in order to determine propagation delay of the ALU.

Implementation



We chose to implement the ALU by reusing as many components as possible along the way. Each module always performs an operation on the two operands, and the final result is chosen

using the mux index obtained from inputting the ALU command into the lookup table. The lookup table outputs a flag that determines whether or not the operand B is inverted (used by the adder/subtractor), a flag that determines whether or not the NAND and NOR gates are inverted at their outputs to become AND and OR gates, and an index indicating which module outputs should be selected by the muxes as the final result and flags. Each module sets flags, which are set to 0 except for the case of add/subtract operations. The block diagram shows one large mux at the end, but in reality, there is one mux for each bit of the result and for each flag. Each of these muxes has 5 inputs, one for each operation module. The muxes use the address output by the ALU control lookup table to choose the correct bit.

Test Benches

Add

The add operation of the ALU is supposed to have the following capabilities:

1. Add 32-bit numbers accurately.
2. Set a Carryout flag appropriately.
3. Set an Overflow flag appropriately.
4. Set a Zero flag appropriately.

Test cases were chosen to test these capabilities.

Simply to test edge cases, we have inputs of all 0s and 1s to make sure our add function can handle this operation on either end (as these are the smallest and largest values that should be able to passed into our add operation.

The later tests we chose based on the Carryout and Overflow flag. We wanted all possible combinations of Carryout and Overflow and so chose four test cases that would output the following Carryout/Overflow combinations (these are commented in our code):

- Carryout = 0, Overflow = 0
- Carryout = 1, Overflow = 0
- Carryout = 0, Overflow = 1
- Carryout = 1, Overflow = 1

Later, we realized that an important part of the add operation was making sure that the Zero flag was being set correctly. The prior tests had cases where there were Zero flags of 0 and 1, but in order to add another case where the inputs weren't all 0s but it would still set another flag, another test case was added in order to test that aspect.

Subtract

The subtract operation of the ALU is supposed to have the following capabilities:

1. Subtract 32-bit numbers accurately.
2. Set a Carryout flag appropriately.
3. Set an Overflow flag appropriately.
4. Set a Zero flag appropriately.

Test cases were chosen to test these capabilities.

Much like the adder, we wanted to test edge cases - which is inputs of all 0s and all 1s, as they represent the smallest and the largest values that can be sent into the ALU.

Because the Sub command of the ALU also sets Carryout, Overflow, and Zero flags, we made sure that that there were cases that had it so the following cases were tested:

- Carryout = 0, Overflow = 0
- Carryout = 1, Overflow = 0
- Carryout = 0, Overflow = 1
- Carryout = 1, Overflow = 1
- Zero = 0
- Zero = 1

We also made sure, in this subtraction test module, that we had the following combinations represented:

- positive - positive = positive
- positive - positive = negative
- negative - negative = positive
- negative - negative = negative

This was so that there would be more robust test cases with values types (e.g. negative, positive) that are likely. Also, a subtractor would have to be able to perform all of these operations correctly, so there is also that aspect of creating a functioning ALU.

Xor

The xor operation of the ALU should perform the following tasks:

1. Accurately xor two values.
    a. 0 xor 0 is 0
    b. 0 xor 1 is 1
    c. 1 xor 0 is 1
    d. 1 xor 1 is 0

We started with the edge cases of all 0s and 1s.

Xor is a bitwise operation, so each bit is xor-ed with each other for the definition above. The ALU needs a 32-bit input to work. The three input cases we chose were:

- two numbers that are the same → xor would return all 0s
- two numbers that are completely different → xor would return all 1s
- two numbers that correspond for some of the bits → xor would return 0s and 1s, depending in the actual bits

This is so we could test the individual bit operations robustly, but also be able to test large numbers that envelop all possible combinations of 32 bit binary numbers.

SLT

The SLT operation of the ALU should perform the following tasks:

1. Determine if an operand A is less than an operand B.
    a. A < B is 1
    b. A = B is 0
    c. A > B is 0

To begin our testing, we started off with the edge case of A being all 0s and B being set to all 1s, and vice versa. This would represent the two largest differences between the operands.

SLT is not a bitwise operation, and instead returns a 32 bit 0 or a 32 bit 1. The cases we used to test out our SLT are the following:

- Positives
    - A > B
    - A = B
    - A < B
- Negatives
    - A > B
    - A = B
    - A < B
- Combination of positives and negatives
    - +A > -B
    - -A > -B
    - -A < -B

These would not only allow us to test out the three ‘cases’ for any less than statement (less than, equal to, or greater than), but the variety of numbers going into it means that it will test if it is able to would with all possible numbers - thereby proving that it is working.

AND

The AND operation of the ALU should perform the following tasks:

1. Accurately and two values.
    a. 0 and 0 is 0
    b. 0 and 1 is 0
    c. 1 and 0 is 0
    d. 1 and 1 is 1

As per most of our tests, we AND-ed operands with the two edge cases.

AND is yet another bitwise operation. Our tests, therefore, were:

- two numbers that are the same

- two numbers that are completely different
- two numbers that correspond for some of the bits

Within these cases, we made sure to have plenty of the following operations:
- 0 and 0
- 0 and 1
- 1 and 0
- 1 and 1

This is so that not only the functionality of the AND can be tested, but it can also be stacked up against the different types of operands that could be passed into it.

NAND
The NAND operation of the ALU should perform the following tasks:
1. Accurately nand two values.
    a. 0 and 0 is 1
    b. 0 and 1 is 1
    c. 1 and 0 is 1
    d. 1 and 1 is 0

The test cases were the same as the AND, with the expected output simply being the inverse of the expected output for the AND. This provided yet another way of making sure both operations were working correctly - after all, one was based on the other operation.

NOR
The NOR operation of the ALU should perform the following tasks:
1. Accurately nors two values.
    a. 0 and 0 is 1
    b. 0 and 1 is 0
    c. 1 and 0 is 0
    d. 1 and 1 is 0

The two edge cases of all 0s and all 1s being passed in were two edge cases.

NOR is a bitwise operation. Our tests model the tests for the previous bitwise operations implemented in the ALU:
- two numbers that are the same
- two numbers that are completely different
- two numbers that correspond for some of the bits

Within these cases, we made sure to have plenty of the following operations:
- 0 and 0
- 0 and 1

- 1 and 0
- 1 and 1

Therefore, not only is the operation of the NOR being tested, but it also tests whether a range of values can be passed into it accurately.

OR

The OR operation of the ALU should perform the following tasks:

2. Accurately ors two values.
    a. 0 and 0 is 0
    b. 0 and 1 is 1
    c. 1 and 0 is 1
    d. 1 and 1 is 1

The test cases for the OR functionality were the same as for the NOR operation, with the expected output being the inverse of the expected output for the NOR.

The test cases were the same as the AND, with the expected output simply being the inverse of the expected output for the AND. This provided yet another way of making sure both operations were working correctly - after all, one was based on the other operation.

Errors

Throughout the course of this lab, we had many errors. However, a lot of our errors were pretty similar to each other and stemmed from the same issue. Please refer to the table below to get a comprehensive list of errors we encountered and how we managed to fix them.

Error	Example	Fixing
At times, we actually got the carryout and overflow flags wrong for certain values.	32'd0 - 32'd0 Cout = 1 Overflow = 0	This error was featured in our test bench, so all we had to do was change the actual conditions for the failure of the test. Although this wasn't an issue with the actual ALU and more so with our understanding, it still did happen for a fair amount of the tests where we actually wrote the tests incorrectly and so it seemed like our ALU was giving more errors than it should have.
Our nor/or operations were	32'd0 or 32'd1 → 0	This was essentially just a

flipped.	32'd0 or 32'd1 → 1	problem with the inverter signal. We had to set the inverter signal to 0 for nor and 1 for or, which we had reversed before.
Our outputLUT to actually perform the operations of the ALU was not working - it simply wouldn't do the operation we wanted and would simply propagate garbage.	32'd0 + 32'd0 = X	This was due to the instantiation of our LUT. We use an always block, having it start always @(muxindex). However, we really needed it to toggle whenever there was a change in the result, so we simply further added extra cases in the always block - the extra cases being the results of the operations, and or-ed them all together.
When testing our ALU, it kept propagating z's.	32'd0 + 32'd0 = Z	It turns out that although we had a delay, we needed a larger delay - like a factor of 100x larger delay.
Our addition and subtraction for 32 bit numbers wasn't working.	32'd50 - 32'100 != -50 (sorry, forgot what actual incorrect number it had)	This was due with an error with how we were actually adding and subtracting, specifically within our for loops. We were not indexing our intermediate carryout effectively, causing operational issues. We fixed it by changing the length of the wire (it is only 31 bits, not 32 bits) and indexing it correctly (not i, but i - 1).
The zero flag was not working.	For 32'd0 + 32'd0, the zero flag was set to zero.	This was due to the way we were setting the zero flag. Initially, we had xor-ed all bits with 32 bits that were all 1, and then we inverted that and took the least significant bit. We realized we were wrong in what we thought that was doing - we actually had to perform a bitwise operation of

		or, and or all the bits together. Then we could invert that in order to get the zero flag. This fixed our issue with the zero flag - with the exception of one case . . .
Random error with subtract zero flag when it is supposed to be asserted	For 32'd0 - 32'd0, the zero flag is set to 0, not 1.	The zero flag works for addition operations and subtraction operations - *except* when it is supposed to be 1. We spent a lot of time trying to fix this with no avail, especially since there is nothing in our implementation that makes setting a zero flag for the subtraction operation any different than the addition - though clearly, there is a difference. This is the only error that our test bench still has, and because it is a trivial one (subtraction is essentially just the same as addition with further modifications to operandB, ergo, if our zero flag for addition works, it's fine) we decided not to pursue this error any further than we already have.

Additions

We took the suggested approach to this lab, and started off by writing the ALU test bench. Even though we didn't finalize the test bench, we did manage to get a pretty good skeleton - ergo, all of the test cases we had originally thought to put in there, we put in. The difference was, with our fleshed out test bench, it was more organized and self-checking (whereas previously there were comments explaining the purpose of each case and some of the values filled in).

Because of this, we actually didn't have to add any test cases - the test cases we had originally ideated sufficed! In retrospect, this seems more due to the fact we had spent a lot of time initially on the test bench - more so than we actually should have. Next time, a better approach would be to actually time box making the test bench rather than try to ideate all possible test cases initially, which ended up slowing us down.

Timing Analysis

The full run of the testbench.

ADD/SUBTRACT
Each 1 bit full adder has a maximum propagation delay of 100 units, and the xor operation to invert B has a delay of 20 units. In total, there is a delay of 32 * 120 = 3840 units max before a sum can be measured.

XOR
XOR takes a delay of 20 for each bit, for a total of 640 units max before the result can be read.

SLT
The SLT operation takes as long as the subtraction operation, plus the delay of 20 that it must take to XOR the last bit of the subtraction result with its overflow.
AND/NAND and NOR/OR
Each AND/OR gate and each XOR gate has a delay of 20 units, so for 32 bits, the maximum delay is 1280 units.

Work Plan Reflection

Before implementing a 32-bit, eight-operation ALU, we first created a work plan to better manage our work/time. Here is our original workplan:

Task	Duration	Date Completed
Test cases of ALU and submodules	2 hours	10/6/17
Verilog implementation of ALU Simple implementation (non-optimized, separated operations) Optimized implementation (reducing silicon area and delay)	2.5 hours 2 hours	10/8/17 10/11/17 (stretch goal)

Uploading to FPGA (writing wrapper and uploading)	1 hour	10/10/17
Lab 1 Report	2 hours	10/11/17

Note that in the third task, we initially thought we had to upload our ALU to the FPGA. This turned out to be *false* and didn't fit within the scope of the lab.

The following table reflects the updated tasks, the duration, and the date at which it was completed.

Task	Duration	Date Completed
Test cases of ALU and submodules	3 hours	10/11/17 (added more)
Verilog implementation of ALU		
Simple implementation (non-optimized, separated operations)	7 hours	10/12/17 (random bugs kept appearing)
Optimized implementation (reducing silicon area and delay)	Didn't complete	Didn't complete
Lab 1 Report	3 hours	10/12/17

As can be seen by the the table, we had initially under-scoped the amount of time it would take to complete each task. This is primarily because we didn't account for small bugs. Though we accounted for bugs in the code, we were not expecting errors that would take hours to debug. This is why the Verilog implementation of the ALU took more than twice the amount of time we expected it too - because we kept on trying to fix very minor errors in the program for hours. The other tasks took only a little more time than we anticipated but not as severe as the Verilog implementation.

As for the date completed, it seems as if though everything was done last minute, but in reality this was just due to the fact that there were many random errors that we were very unsure how to fix (as we had never dealt with them before). This was especially true on the final day, because the code we had pushed the night prior had worked, and seemingly broke again after fixing another small error.

The takeaway from this is that in the future, we will spend more time in consideration of such things happening, and budget for them more carefully. Although we attempted to do that by leaving our schedules fairly free and timing our meetings around help hours, as well as doing a fair amount of work individually, it seems that still wasn't accurate to real life.