Lab1 Report

Anna Buchele, Apurva Raman

1. Implementation

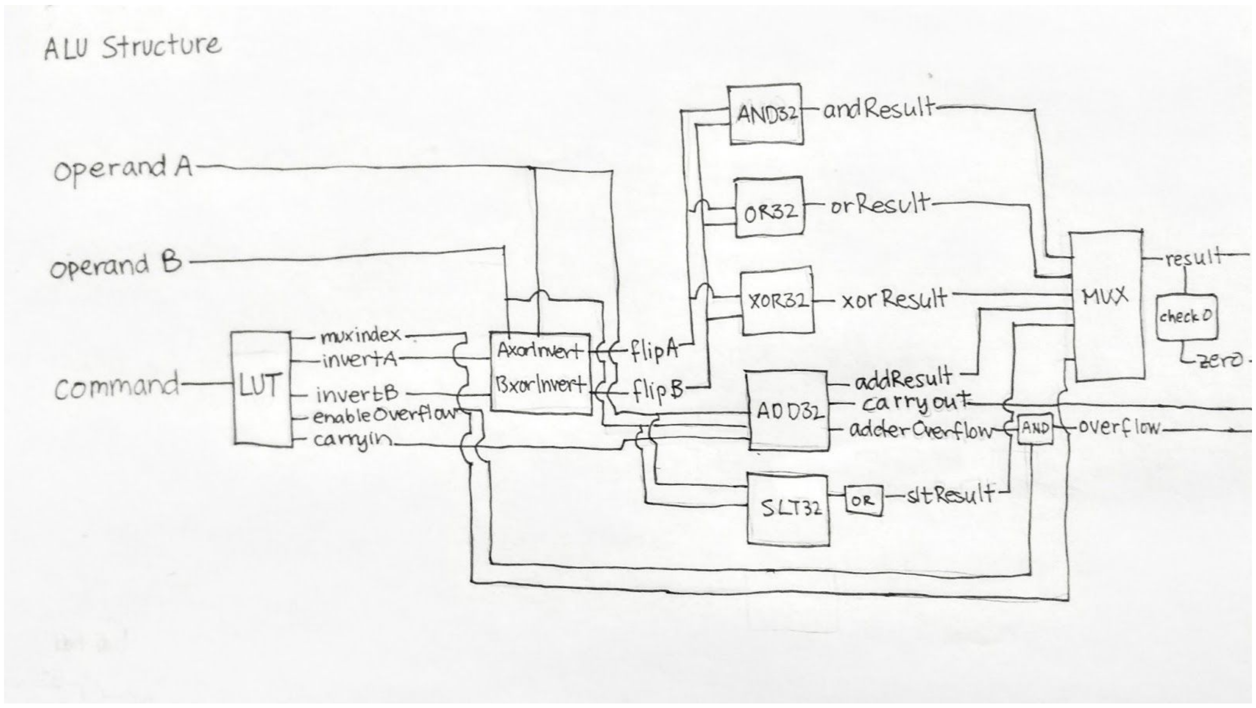


Figure 1. Structure of the ALU

To implement the ALU, we started by implementing a module for each 32-bit operation the ALU would perform (adder/subtractor, and, or, nand, nor, xor, and the SLT). When we modified our 4 bit adder from Lab0 to be a 32 bit adder, we also added the subtractor functionality by inverting the second input and making the carryin value 1. We then noticed that we could make nand and nor from inverting both inputs of or and and, respectively.

We then implemented a lookup table which took the 3 bit command value as input and output the muxindex, flags for whether or not to invert operandA and operandB, a flag for allowing overflow (essentially indicates to suppress overflow for the SLT), and whether or not to have a carryin of 1 (for the subtraction and SLT operations). We condensed the number of muxindexes to 5: index 0 for ADD and SUB, 1 for XOR, 2 for SLT, 3 for AND and NOR, and 4 for NAND and OR.

The command goes to the LUT, and the muxindex and the flags are output. Based on the flags, the values for A and B were inverted or kept intact and then each operation was performed. The results from each operation are then input into the mux, which chooses which one to use based on the muxindex value. The overflow is determined based on the overflow from the adder and whether enableOverflow is true. The carryout is the carryout from the adder. The zero is calculated from checking if any of the bits of the result are 1.

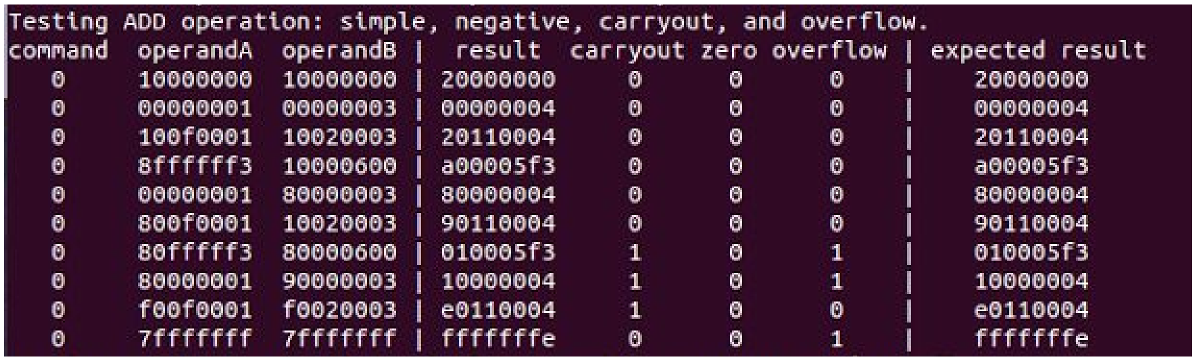
2. Test Results

We wrote many test benches during the course of the lab, and used their results to assess the functionality of our program. We tested each module individually, and then tested several modules together as the program reached higher operations.

Once we had finished the program, we wrote a test bench that tests the full program, inputting only a command signal and two operands. We ran multiple tests for each of the operations, and with the results of the test bench, we were able to make the necessary modifications to our program for correct performance.

Below is the output of our completed test bench, sectioned with explanations for why we implemented it the way we did.

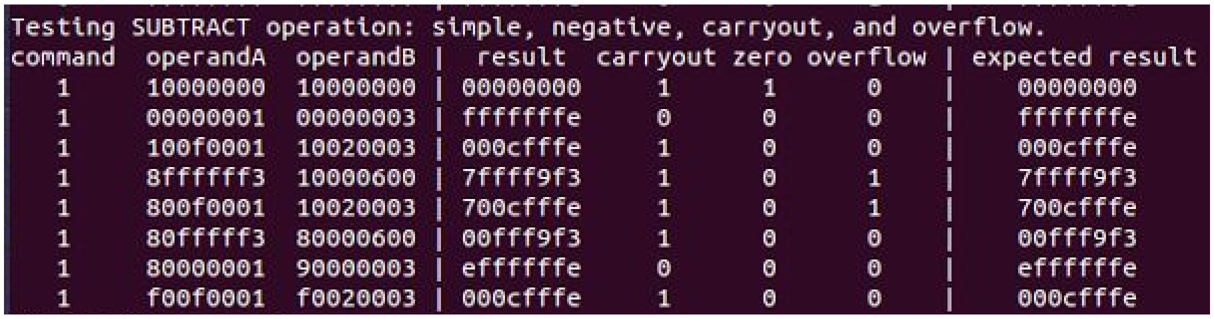
Section 1: ADD operation:



```
Testing ADD operation: simple, negative, carryout, and overflow.
command  operandA  operandB |  result  carryout zero overflow | expected result
   0     10000000  10000000 | 20000000     0      0      0    |    20000000
   0     00000001  00000003 | 00000004     0      0      0    |    00000004
   0     100f0001  10020003 | 20110004     0      0      0    |    20110004
   0     8ffffff3  10000600 | a00005f3     0      0      0    |    a00005f3
   0     00000001  80000003 | 80000004     0      0      0    |    80000004
   0     800f0001  10020003 | 90110004     0      0      0    |    90110004
   0     80fffff3  80000600 | 010005f3     1      0      1    |    010005f3
   0     80000001  90000003 | 10000004     1      0      1    |    10000004
   0     f00f0001  f0020003 | e0110004     1      0      0    |    e0110004
   0     7fffffff  7fffffff | fffffffe     0      0      1    |    fffffffe
```

We chose these test cases because we wanted to test both simple adding operations, as well as more complicated ones. The more complicated adding operations include operations resulting in a carryout or overflow, and operations involving negative numbers. For the ADD operation we ran four simple tests, three tests involving a negative and a positive, three tests of two negatives, and one positive overflow case. The testbench showed us in this case that our overflow and carryout operations were not working, and allowed us to fix them. It ended up that the carryout and overflow were outputs of multiple modules that were being run in one module. We had set the output and carryout to zero for several operations that did not involve a carryout or overflow, and that was confusing the result of the ADD operations' carryout and overflow.

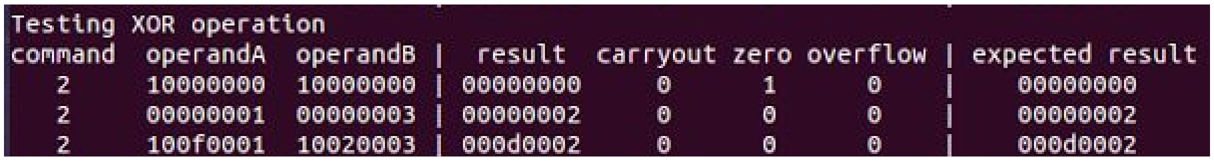
Next is our SUBTRACT operation:



```
Testing SUBTRACT operation: simple, negative, carryout, and overflow.
command   operandA  operandB |  result  carryout zero overflow | expected result
   1      10000000  10000000 | 00000000     1      1      0    |    00000000
   1      00000001  00000003 | fffffffe     0      0      0    |    fffffffe
   1      100f0001  10020003 | 000cfffe     1      0      0    |    000cfffe
   1      8ffffff3  10000600 | 7ffff9f3     1      0      1    |    7ffff9f3
   1      800f0001  10020003 | 700cfffe     1      0      1    |    700cfffe
   1      80fffff3  80000600 | 00fff9f3     1      0      0    |    00fff9f3
   1      80000001  90000003 | effffffe     0      0      0    |    effffffe
   1      f00f0001  f0020003 | 000cfffe     1      0      0    |    000cfffe
```

We chose these test cases because we wanted to test both simple subtraction operations, as well as more complicated ones. The more complicated subtraction operations include operations resulting in a carryout or overflow, operations where the output is negative, and operations with a negative input. We also chose the first test to make sure that it was outputting a zero result correctly. For the SUB operation we ran 2 simple test, 5 tests with a negative input, 2 tests with a negative output, and six tests with a carryout or overflow. We also included a zero case to check our zero operation. In this case, the test bench helped us locate a bug that prevented the subtraction operation from being performed correctly. In the ADD module, we had an operation to flip operandB (and add 1) if the input flag for subtraction was true. However, we also had an operation in the ALU to flip both operandA and operandB if the corresponding invert flag was true. This led to operandB getting inverted twice, leaving us with the original operandB plus/minus 1. After we fixed this bug, and many others, when we ran the test bench again we noticed it was now performing addition (plus 1). When we went to check the code, we found that in our removal of other bugs, we had removed both the operandB invert operations. We were then able to re-include the original invert operation in the addition module.

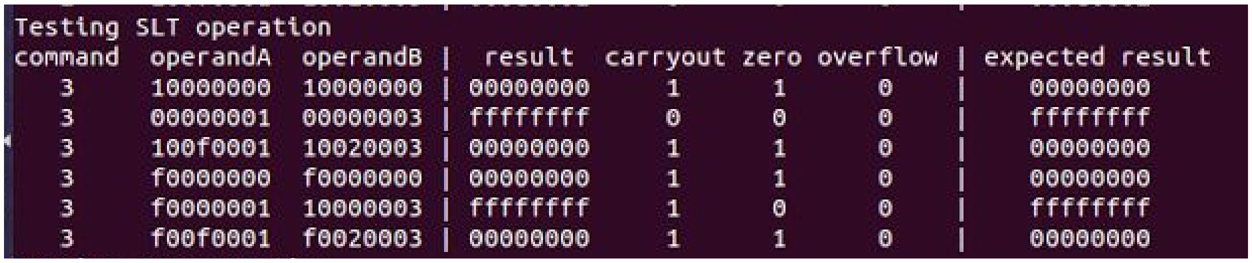
Next is our XOR operation:



```
Testing XOR operation
command   operandA  operandB |  result  carryout zero overflow | expected result
   2      10000000  10000000 | 00000000     0      1      0    |    00000000
   2      00000001  00000003 | 00000002     0      0      0    |    00000002
   2      100f0001  10020003 | 000d0002     0      0      0    |    000d0002
```

For our XOR operation, and for the other simpler modules, we used fewer test cases. This is because these operations were much simpler to implement, and once we implemented them correctly we did not have more problems with the majority of them. In this case, we simply included three tests of simple XOR operations. An input being negative or large would not affect the way and XOR is run in our program, so we saw no need to implement more test cases and decided that simplicity would be best for quickly recognizing errors.

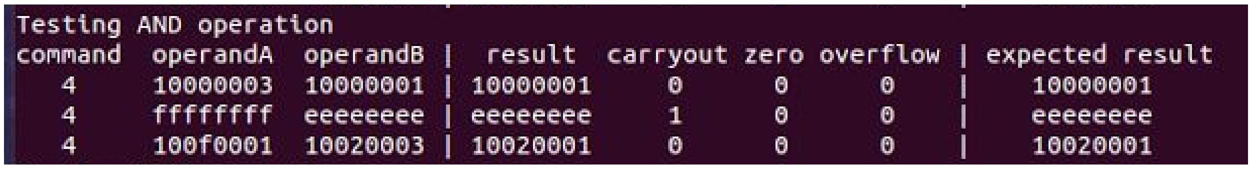
Next is our SLT operation:



```
Testing SLT operation
command   operandA  operandB |  result   carryout zero overflow | expected result
   3      10000000  10000000 | 00000000     1       1     0     |    00000000
   3      00000001  00000003 | ffffffff     0       0     0     |    ffffffff
   3      100f0001  10020003 | 00000000     1       1     0     |    00000000
   3      f0000000  f0000000 | 00000000     1       1     0     |    00000000
   3      f0000001  10000003 | ffffffff     1       0     0     |    ffffffff
   3      f00f0001  f0020003 | 00000000     1       1     0     |    00000000
```

For our SLT operation, we chose to include six test cases- three of positive numbers and three with negative numbers. For each, we included first, a case where the two inputs are equal; second, a case where operandA is less than operandB; and third a case where operandA is larger than operandB. We chose this because we knew that negative numbers would affect our SLT and we wanted to make sure we would perform the operations correctly. We also included SLT tests with two equal inputs because we knew that equal inputs might have an effect on our program. We included the unequal inputs for obvious reasons- to test if the SLT could correctly identify whether operandA was less than operandB. For our SLT, the test bench helped us identify a bug in which the majority of the result was undeclared. For example, our result for every SLT test was (in hex) “xxxxxxxX”. When we put it in binary, we could see that the entirety of the 32 bit result expected by the multiplexer was undeclared except for the last bit. After discovering this, we were able to modify the SLT and the SLT results so that it would be output as a 32 bit of either all ones or zeros. We decided that this would be the least misleading way to represent the SLT result.

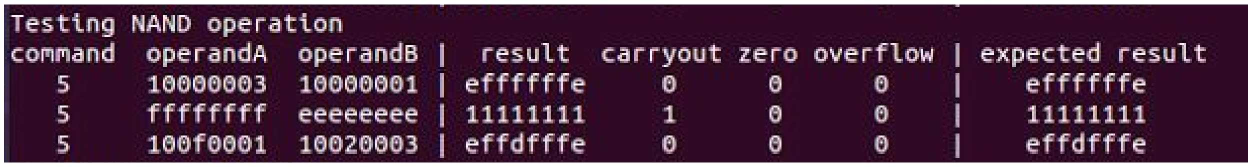
Next is our AND operation:



```
Testing AND operation
command   operandA  operandB |  result   carryout zero overflow | expected result
   4      10000003  10000001 | 10000001     0       0     0     |    10000001
   4      ffffffff  eeeeeeee | eeeeeeee     1       0     0     |    eeeeeeee
   4      100f0001  10020003 | 10020001     0       0     0     |    10020001
```

For our AND operation, we chose to include three test cases- one with negative numbers, and two with positive numbers. We chose so few test cases because, similar to the XOR function, this operation was very simple to implement and we did not have any trouble within the module after implementation. One bug that the test bench helped us uncover was that the AND output was not wired correctly to the multiplexer- Although it had muxindex 3, in implementation it was being wired to muxindex 2- the same mux index as the SLT. Once we discovered this, we were able to quickly fix it and continue with our implementation.

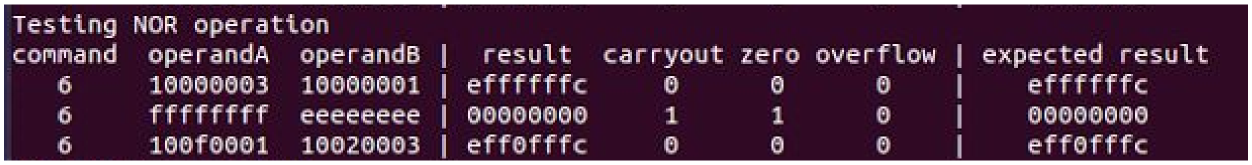
Next is our NAND operation:



```
Testing NAND operation
command   operandA  operandB |  result  carryout zero overflow | expected result
   5      10000003  10000001 | effffffe     0       0      0    |    effffffe
   5      ffffffff  eeeeeeee | 11111111     1       0      0    |    11111111
   5      100f0001  10020003 | effdfffe     0       0      0    |    effdfffe
```

For our NAND operation, we used the same tests as in our AND operation. We did this because the NAND operation should have the same outputs as the AND operation, only inverted. In this case, we included two positive-only cases and one negative case. The test bench for this operation output the expected results the first time we ran it, and every time afterwards. It did not help us catch any bugs, but it did confirm that the module was working correctly.

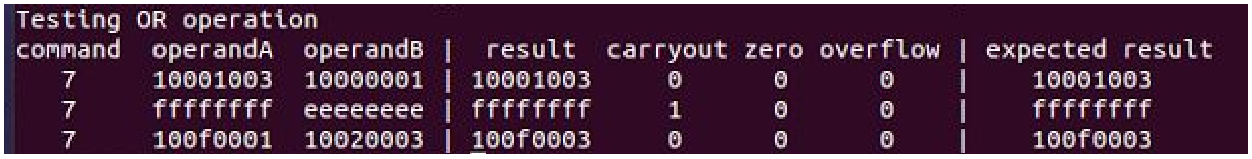
Next is our NOR operation:



```
Testing NOR operation
command   operandA  operandB |  result  carryout zero overflow | expected result
   6      10000003  10000001 | efffffffc    0       0      0    |    effffffc
   6      ffffffff  eeeeeeee | 00000000     1       1      0    |    00000000
   6      100f0001  10020003 | eff0fffc     0       0      0    |    eff0fffc
```

For our NOR operation, we again did three simple tests, as the NOR operation was simply the AND operation with inverted inputs, per De Morgan's law. This test bench helped us uncover the same bug as in the AND operation, where the AND output was not wired correctly to the MUX. As the AND output contained either the output of an and function or a nor function, depending on the invert signals, when the AND was miswired it affected both the NOR operation and the AND operation. The NOR test bench was actually our first signal that something was wrong, and it wasn't until we had run both the AND test bench and the NOR test bench that we were able to isolate the problem to the MUX.

Next is our OR operation:



```
Testing OR operation
command   operandA  operandB |  result  carryout zero overflow | expected result
   7      10001003  10000001 | 10001003     0       0      0    |    10001003
   7      ffffffff  eeeeeeee | ffffffff     1       0      0    |    ffffffff
   7      100f0001  10020003 | 100f0003     0       0      0    |    100f0003
```

For our OR operation, we used the same tests as our NAND operation- since our NAND operation was actually the same as our OR operation, only with inverted inputs. We chose this because the OR was a relatively problem-free implementation and we did not have any problems with it after implementation. We chose three tests, two with positive inputs and one with negative inputs. Since it was the same gate as the NAND operation, we also did not catch any bugs using the test bench- the test bench output the expected results the first time, and every time afterwards. Although it did not help us catch any bugs, it did confirm to us that the module was working correctly.

3. Timing Analysis

The first part of the timing analysis was figuring out the gate delays. Based on the model of having the timing be proportional to the input, we determined that the NOT gate had a delay of 10 units and the NAND and NOR gates had a delay of 20 units each. Since the AND and OR gates have a “hidden” inverter, they have an additional 10 units of delay. We found that the XOR gate, when written in terms of fundamental gates, had a worst-case path through 3 NAND gates, and so we set the gate delay to 60.

For the 32 bit AND and OR modules, the propagation delay is 960, or 30 for each bit.

For the 32 bit XOR gate and for the module that XORs a 32 bit number with a single bit, the propagation delay was 1920 (60*32).

32 bit NAND and NOR require inverting both inputs first (which uses 2 XOR gates) so the delay for these operations is (1920*2+960), which is 4800.
One way to improve the speed would be to base the 32 bit AND and OR off NAND and NOR, rather than the other way around.

The one bit adder (structuralFullAdder) has a propagation delay of 120 since the worst-case path is through an XOR gate, an AND gate, and an OR gate.

The 32 bit adder/subtractor uses the module that XORs each bit of a 32 bit number with a single bit (delay = 1920) and calls the one bit adder 32 times (delay = 120*32). It then has an additional XOR (delay = 60) gate for overflow. This gives us a worst-case propagation delay of 5820.

The SLT uses an adder/subtractor (delay = 5820) and a one bit XOR gate (delay = 60), so its propagation delay is 5880.

The worst-case operation for the ALU is therefore the SLT.

4. Work Plan Reflection

Our original work plan, shown here, allotted 9 hours and 25 minutes for completion of the lab.

-Test bench (1 hour)
-Structure for and, nand, or, nor, xor (45 minutes)
-Structure for adder,subtractor (1 hour)
-Structure for SLT (2 hours)

-Lookup table (1.5 hours)
-Multiplexer (30 minutes)
-Gate delays (15 minutes)
-Timing Analysis (30 minutes)
-Write lab report (1 hour)
-Debugging (45 minutes)
-Work plan reflection (10 minutes)

Total: 9 hours 25 minutes

Sunday: 3-6; test bench, adder, subtractor, and, nand, or, nor, xor
Monday: 7:30-10; SLT, plan for lookup table
Tuesday: 1-2; lookup table, multiplexer
Wednesday: 2:30-5:30; timing analysis debugging, gate delays, start lab report 8-9; finish lab report, work plan reflection,final bug fixes

Our actual times to complete the lab were much longer. We each counted up (using our calendars and github commit histories) how long we had spent working on this lab. Anna spent 18.5 hours working on the lab, while Apurva spent about 22 hours working on the lab. We spent 9 of those hours each working together, while the rest was time spent individually. Both of these times are far longer than we thought this lab would take. Most of the time was spent debugging hard-to-identify problems even when we had successfully understood the concepts and implemented logic that worked (an example of this was spending a long time testing when the module/file referenced was incorrect). The parts we ended up doing alone also ended up taking longer than we thought. We also didn't account for things like the mux and the zero checking in the plan. An estimated breakdown of what we spent our time on follows:

-Test benches: **~2 hours**
-Test bench debugging: **~3 hours**
-Structure for and, nand, or, nor, xor (together): **~30 minutes**
-Debugging gate reversal issues: **1 hour**
-Structure for adder, subtractor (together): **~90 minutes**
-Structure for SLT: **~30 minutes**
-Debugging SLT: **~1 hour**
-Structure for LUT, ALU module, and multiplexer: **~4 hours**
-LUT, ALU, MUX Debugging: **~4 hours**
-Gate delays and timing: **~45 minutes**
-Zero checking and debugging: **~4 hours**
-Lab report: **~2 hours**
-Total of above: 24 hours, 15 minutes
-Misc Debugging: **~ 16 hours, 45 minutes**

We’re DONE

Anna:
Sunday: 3-6, (3)
Monday: 11-12, 7:30-11, (4.5)
Tuesday: 1-2 (1)
Wednesday: 2:30-4:30, 5:30-6:30, 8-10 (5)
Thursday: 12-1, 8-12 (5)

Apurva
Sunday: 3:00-6:00 (3)
Monday: 7:30-11 (3.5)
Tuesday: 1:00-2, 5:30-6, 8:00-2:00 (8.5)
Wednesday: 2:30-5:30, 9:30-12:00 (5.5)
Thursday: 9:30-11:30 (2)
Total: 22.5