Lab 1 Report

Changjun Lim, Sungwoo Park

For this lab, we have implemented a 32-bit ALU.

< Implementation >

We used “bit slice” approach to construct a 32-bit ALU capable of multiple operations. Following is the top-level diagram of the 32-bit ALU that consists of 32 1-bit ALU chained together.

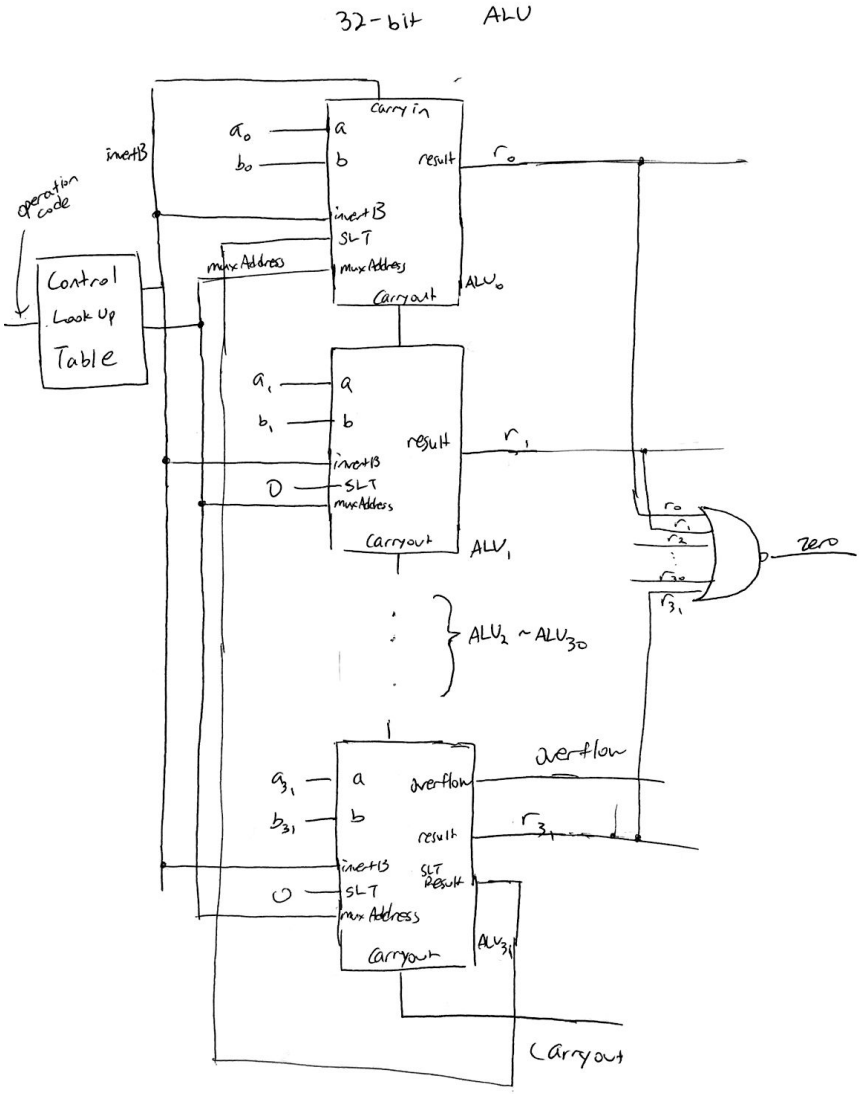


Figure 1: 32-bit ALU

Individual bits of 32-bit operands are fed into one of 32 1-bit ALU, along with appropriate control signals (invertB flag, muxAddress to choose appropriate mux output within the 1-bit ALU, SLT

input value). 1-bit results from all 1-bit ALUs are combined together to form a final 32 bit result of the operation. Notice that carryout from first 31 1-bit ALUs (denoted ALU_0 - ALU_{30}), are fed into carry-in of next 1-bit ALU to correctly handle carrys in ADD/SUB operations. Also notice that our implementation handles the zero flag by computing the result of combining all result bits by NOR gates.

In the below figure, we present the implementation of the 1-bit ALU.

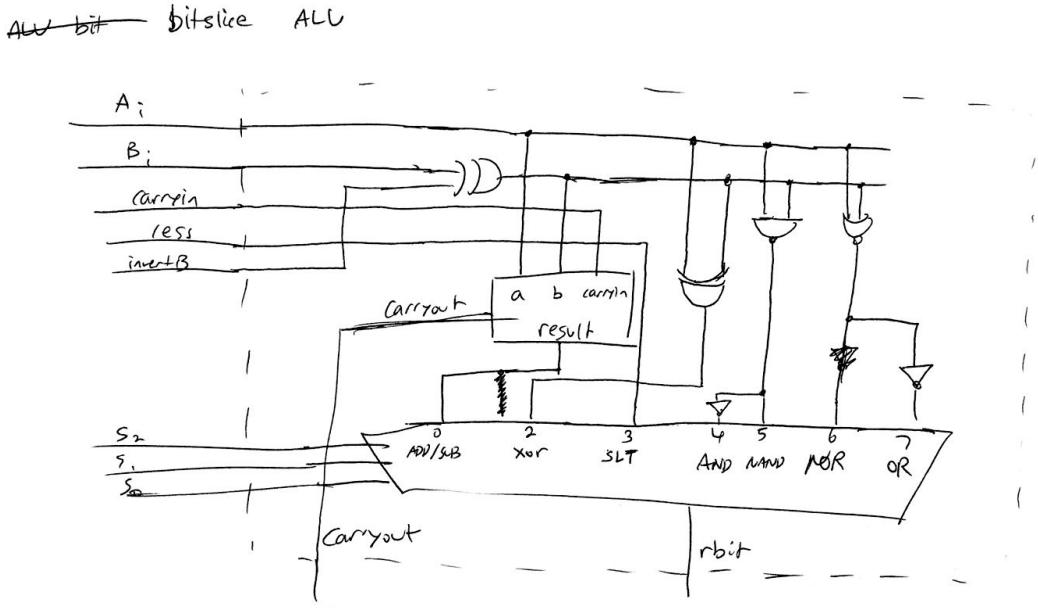


Figure 2: 1-bit ALU

The 1-bit ALU is basically a multiplexer, in which an input to the multiplexer is a result of different operations based on the given operand values. Notice that we used a single 1-bit adder for both outputs for ADD and SUB. This is made possible by selectively inverting B bit based on the given operation code. The 3-bit multiplexer used in this implementation was constructed by combining 2 2-bit multiplexers. The implementation of the 3-bit multiplexer is shown below.

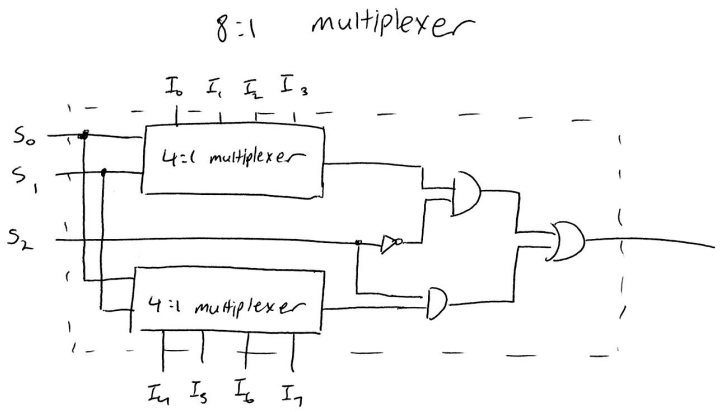


Figure 3: 3-bit multiplexer

< Testing >

(3-bit mux)

We started by writing simple test cases for 3-bit mux to make sure that we did not make any obvious errors. Following is the output from 3-bit mux test benches.

Addr	inputs		Output
000	00010000	|	0
000	10000001	|	1
000	00000000	|	0
011	01000001	|	0
010	00001101	|	1

Since 2-bit mux that I used to construct this 3-bit mux are already tested and working, I did not write very exhaustive test cases that cover every possible cases. I just chose few cases to make sure that the multiplexer is behaving as expected. All test cases that I tested worked.

(1-bit ALU)

Next step in testing was writing test benches for 1-bit ALU. Since this is a critical component of the ALU that we are constructing, I set out to create test cases that are exhaustive and covers most possible cases.

testing ADD

bitA	bitB	carryin	less	muxIndex	invertBFlag	bitR	carryout
1	1	0	0	100	0	1	1
1	0	0	0	100	0	0	0
1	0	0	0	000	0	1	0
0	0	0	0	000	0	0	0
0	1	0	0	000	0	1	0
1	1	0	0	000	0	0	1
0	0	1	0	000	0	1	0

```
1  0  1  0  000  0 |  0  1
1  1  1  0  000  0 |  1  1
testing SUB
0  0  0  0  000  1 |  1  0
1  0  0  0  000  1 |  0  1
0  1  0  0  000  1 |  0  0
1  1  0  0  000  1 |  1  0
0  0  1  0  000  1 |  0  1
1  0  1  0  000  1 |  1  1
0  1  1  0  000  1 |  1  0
1  1  1  0  000  1 |  0  1
testing XOR
0  0  0  0  010  0 |  0  0
1  0  0  0  010  0 |  1  0
0  1  0  0  010  0 |  1  0
1  1  0  0  010  0 |  0  1
testing SLT
0  0  0  0  011  1 |  0  0
0  0  0  1  011  1 |  1  0
testing AND
0  0  0  0  100  0 |  0  0
1  0  0  0  100  0 |  0  0
0  1  0  0  100  0 |  0  0
1  1  0  0  100  0 |  1  1
testing NAND
0  0  0  0  101  0 |  1  0
1  0  0  0  101  0 |  1  0
0  1  0  0  101  0 |  1  0
1  1  0  0  101  0 |  0  1
testing NOR
0  0  0  0  110  0 |  1  0
1  0  0  0  110  0 |  0  0
0  1  0  0  110  0 |  0  0
1  1  0  0  110  0 |  0  1
testing OR
0  0  0  0  111  0 |  0  0
1  0  0  0  111  0 |  1  0
0  1  0  0  111  0 |  1  0
1  1  0  0  111  0 |  1  1
```

The test cases shown above are all correct. Before getting this result, we encountered a test cases in which a result returns ‘z’ as a value. We realized that this was happening because we forgot to include a delay after setting register values.

(32-bit ALU)
The final step for test is making test benches for 32-bit ALU. We express operand A and B as hexadecimal values in convenience. And since there is no meaning for flags except ADD and SUB, we do not consider the value of flags for XOR, SLT, AND, NAND, NOR, OR cases.

(operandA	operandB	expected outputs	carryout	zero	overflow)
testing ADD					
00000002	00000001	00000000000000000000000000000011	0	0	0
FFFFFFFF	FFFFFFFF	11111111111111111111111111111110	1	0	0
00000000	00000000	00000000000000000000000000000000	0	1	0
7FFFFFFF	00000001	10000000000000000000000000000000	0	0	1
testing SUB					
00000003	00000001	00000000000000000000000000000010	1	0	0
80000000	00000001	01111111111111111111111111111111	1	0	1
00000000	00000000	00000000000000000000000000000000	1	1	0
FFFFFFFF	FFFFFFFF	00000000000000000000000000000000	1	1	0
testing XOR					
AA550055	AAFF55AA	00000000101010100101010111111111	-	-	-
FFFF0000	00FF00FF	11111111000000000000000011111111	-	-	-
testing SLT					
555555AA	55AA55AA	00000000000000000000000000000001	-	-	-
555555AA	555555AA	00000000000000000000000000000000	-	-	-
00FF00FF	FF00FF00	00000000000000000000000000000000	-	-	-
FFFFFF00	0000FFFF	00000000000000000000000000000001	-	-	-
AAAA55AA	AA5555AA	00000000000000000000000000000000	-	-	-
FF55FF00	FFFF5500	00000000000000000000000000000001	-	-	-
testing AND					
FFFF0000	00FF00FF	00000000111111110000000000000000	-	-	-
FF00AA55	AAAA55AA	10101010000000000000000000000000	-	-	-
testing NAND					
FFFF0000	00FF00FF	11111111000000001111111111111111	-	-	-
FF00AA55	AAAA55AA	01010101111111111111111111111111	-	-	-
testing NOR					
55550055	AAFF55AA	00000000000000001010101000000000	-	-	-
FFFF0000	00FF00FF	00000000000000001111111100000000	-	-	-
testing OR					

55FFAA00	AAAA55AA	| 11111111111111111111111110101010	-	-	-
FFFF0000	00FF00FF	| 11111111111111110000000011111111	-	-	-

Reasoning behind choosing above test cases:

ADD/SUB: Interesting characteristics that we would like to see and design edge cases around are overflow and carry. We chose test cases that will result in neither overflow nor carry, one of them, and both of them.

SLT: We choose 6 cases for SLT. There are 3 parameter related to results, sign of A, sign of B, and A < B. It could be categorized to 8(=2^3) cases, but when A is negative and B is positive, A < B is always true and when A is positive and B is negative, A < B is always false. So there are only 6 categories.

Logic operators: For bitwise logic operations(XOR, AND, NAND, NOR, OR), we choose 2 cases and set each case to have 4 categories(in the truth table) in 32 bits.

Debugging through test cases

```
testing ADD
operandA operandB command |              result              carryout zero overflow| expected outputs
00000002 00000001 0       | xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx01    0     z    x       | 00000000000000000000000000000011  0  0  0
ffffffff ffffffff 0       | xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0    1     z    x       | 11111111111111111111111111111110  1  0  0
00000000 00000000 0       | 00000000000000000000000000000000    0     z    0       | 00000000000000000000000000000000  0  1  0
7fffffff 00000001 0       | x0101010101010101010101010101010    0     z    0       | 10000000000000000000000000000000  0  0  1
testing SUB
00000003 00000001 1       | xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0    x     z    x       | 00000000000000000000000000000010  0  0  0
80000000 00000001 1       | x1010101010101010101010101010101    1     z    0       | 01111111111111111111111111111111  1  0  1
00000000 00000000 1       | 10101010101010101010101010101010    0     z    1       | 00000000000000000000000000000000  0  1  0
ffffffff ffffffff 1       | 10101010101010101010101010101010    0     z    1       | 00000000000000000000000000000000  1  1  0
testing XOR
aa550055 aaff55aa 2       | 0x0x0x0010101010xxxxxxx1x1x1x1x1    1     z    1       | 00000000101010100101010111111111  0  0  0
ffff0000 00ff00ff 2       | x1x1x1x1xxxxxxx00000000xx1x1x1x1    1     z    0       | 11111111000000000000000011111111  0  0  0
testing SLT
555555aa 55aa55aa 3       | x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0    0     z    0       | 00000000000000000000000000000001  0  0  0
555555aa 555555aa 3       | x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0    0     z    0       | 00000000000000000000000000000000  0  0  0
00ff00ff ff00ff00 3       | 0000000xxxxxxxx00000000xxxxxxxx0    0     z    0       | 00000000000000000000000000000000  0  0  0
ffffff00 0000ffff 3       | xxxxxxxxxxxxxxx00000000000000001    1     z    1       | 00000000000000000000000000000001  0  0  0
aaaa55aa aa5555aa 3       | 0x0x0x0x0x0x0x0xxxxxxxxxxxxxxxx    x     z    x       | 00000000000000000000000000000001  0  0  0
ff55ff00 ffff5500 3       | x0x0x0x0x0x0x0xxxxxxxxx0x0x0x0x0    0     z    0       | 00000000000000000000000000000000  0  0  0
ff55ff00 ffff5500 3       | x0x0x0x0x0x0x0xxxxxxxxx0x0x0x0x0    0     z    0       | 00000000000000000000000000000001  0  0  0
testing AND
ffff0000 00ff00ff 4       | xxxxxxxx1111111x0000000000000000    x     z    x       | 00000000111111110000000000000000  0  0  0
ff00aa55 aaaa55aa 4       | xxxxxxx0000000000000000000000000    1     z    0       | 10101010000000000000000000000000  0  0  0
testing NAND
ffff0000 00ff00ff 5       | x1x1x1x1xxxxxxx0xxxxxxx1x1x1x1x1    1     z    0       | 11111111000000001111111111111111  0  0  0
ff00aa55 aaaa55aa 5       | x1x1x1x1x1x1x1x1x1x1x1x1x1x1x1x1    1     z    1       | 01010101111111111111111111111111  0  0  0
testing NOR
55550055 aaff55aa 6       | x0x0x0x0x0x0x0x0x0x0x0x000000000    0     z    0       | 00000000000000000101010100000000  0  0  0
ffff0000 00ff00ff 6       | xxxxxxxxxxxxxxx0xxxxxxxx00000000    x     z    x       | 00000000111111110000000000000000  0  0  0
testing OR
55ffaa00 aaaa55aa 7       | x1x1x1x1x1x1x1xxxxxxxxxxx0x0x0x0    1     z    0       | 11111111111111111111111110101010  0  0  0
ffff0000 00ff00ff 7       | 111111111111111x0000000xx1x1x1x1    1     z    0       | 11111111111111110000000011111111  0  0  0
```

There were few noticeable bugs that we realized after running test cases as shown above. First, we didn't include zero flag in our implementation (as shown in our "z" value). Second, we didn't properly wire carry in and carry out values, resulting in lot of glitch values in our result. Lastly, we got incorrect result for overflow because we used incorrectly values for a xor gate that we used to compute overflow value.

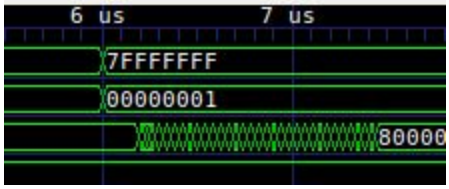
We also realized that our SLT operations are not working properly so we went back to fix SLT operation implementation.

< Timing Analysis >

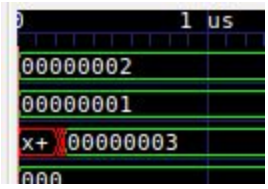
Worst case propagation delay for each operation:
ADD: When bit addition at each bit results in carry (About 50ns for each adder operation * 32 bit ALUs)
SUB: Similar to ADD case. When bit subtraction at each bit results in carry.
XOR, AND, NAND, NOR, OR: Fairly fast operations. Worst propagation delay is just a gate delay for a corresponding operation gate (10, 20, or 30ns depending on the gates).
SLT: Similar to SUB case.

Simulation result

First row is operand A, second row is operand B, third row is the result



This is the ADD operation between 7FFFFFFF and 00000001. Notice that this operation results in big propagation delay due to many carrys.



On the other hand, this ADD operation between 00000002 and 00000001 does not have big propagation delay because it does not have many carrys.

< Work Plan Reflection >

Scheduled work plan

- ALU verilog implementation(alu.v) (3 hours) ~10/6

- Test bench implementation(alu.t.v) (2 hours) ~10/6
- Revising the code (2 hours) ~10/7
- Analyzing the result (2 hours) ~10/9
- Writing Report (2 hours) ~10/10

Actual time spent

- ALU verilog implementation(alu.v): 10/10 (0.5 hours), 10/11 (2 hours) 10/12 (0.5 hours)
- Test bench implementation(alu.t.v): 10/10 (2 hours) 10/11 (0.5 hours)
- Revising the code 10/11 (0.5 hours) 10/12 (3.5hours)
- Analyzing the result 10/12 (1 hour)
- Writing Report 10/12 (1.5 hours)

The total time(12 hours) we have spent for Lab 1 is quite similar to scheduled time(11 hours). It took more time than the schedule to revise the code. The actual date we finished each part is written after the proposed deadline.

Reference:

http://web.cse.ohio-state.edu/~teodorescu.1/download/teaching/cse675.au08/Cse675.02.F.ALUDesign_part2.pdf
(Referenced this ALU implementation document while working on the lab)