Computer Architecture Lab 1 : 32-Bit ALU

Yoonyoung Cho, Haozheng Du, Jee Hyun Kim

October 6 2016

1 Introduction

For this lab, we implemented 32-bit ALU, which supports 8 operations, using Control Logic LUT. We then verified our implementation for each building block against different test cases and modified the code accordingly when the test cases failed.

2 Implementation

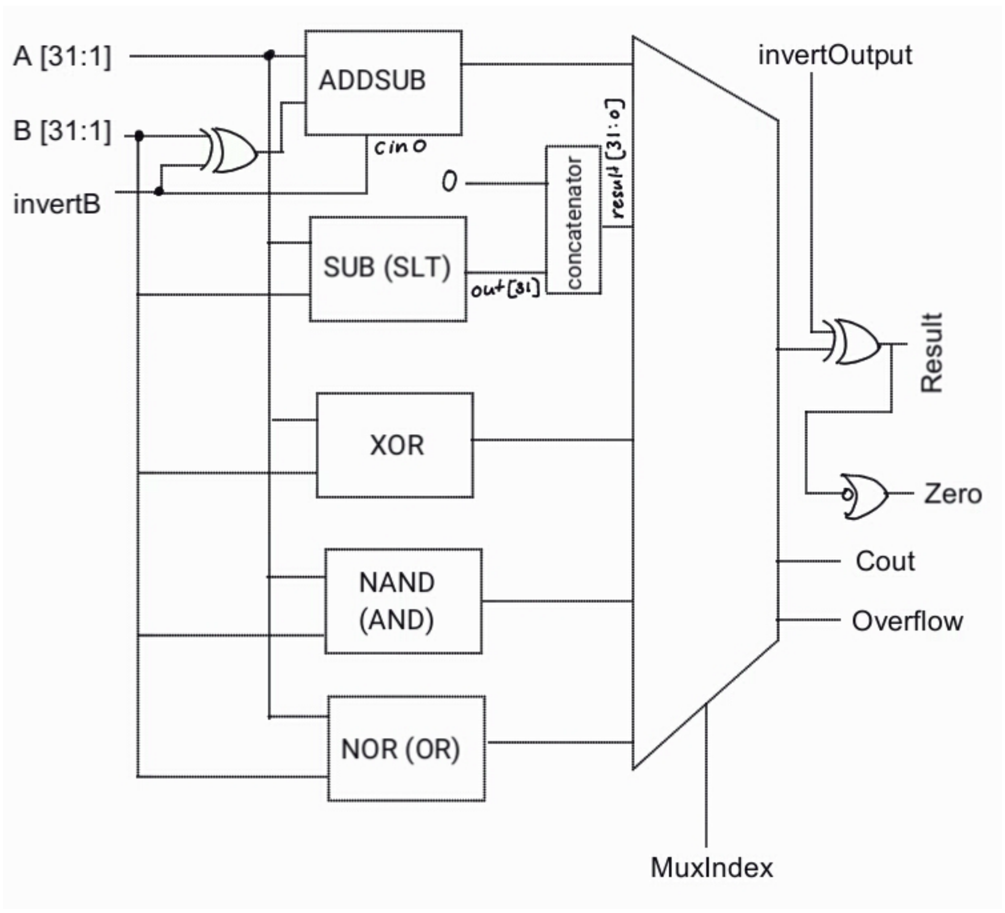

Figure 1: ALU block diagram

We implemented the ALU with modules internally slicing the bits. This decision was primarily based on the fact that because of the asymmetry that occurs in the addition/subtraction modules when dealing with carryout and overflow; we made the modules so that each module completes its function and outputs a full result, which an array of multiplexers would then choose bit by bit.

2.1 Modules

Initially, we built all the modules - ADD, SUB, XOR, SLT, AND, NAND, NOR, OR - for one bit and iterated through 32 bits using a for loop. Then, we combined the inverses (AND with NAND, OR with NOR, SUB with ADD) to form one module each. The output of AND and OR was found by inverting the output of NAND and NOR respectively, and The SUB was found by inverting operandB and passing in 1 as 0th carry in. The SLT was computed by subtracting the B from A and taking the most significant bit (1 for A<B and 0 for A>B).

2.2 Adder-Subtractor

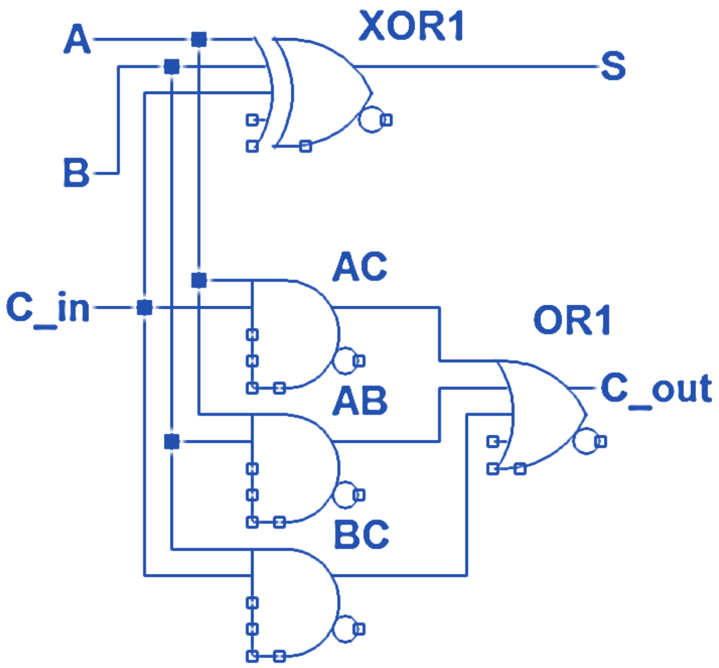

Figure 2: 1-bit full adder

We used the full adder from previous work (where its implementation details are also listed)

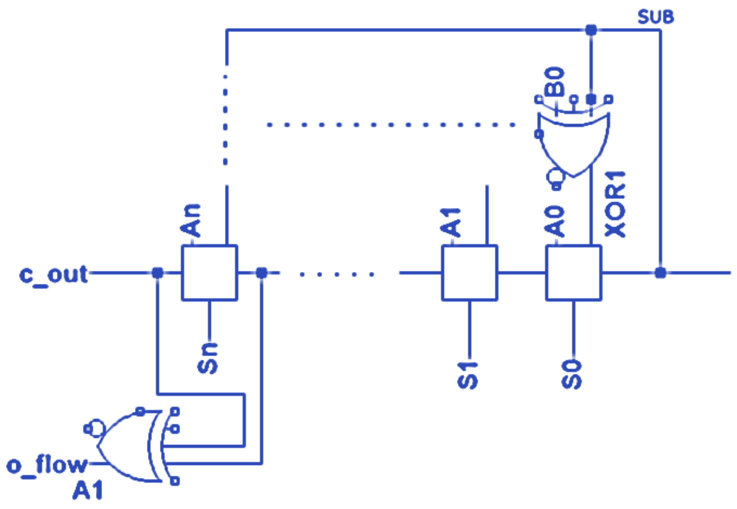

Figure 3: n-bit full adder/subtractor

The implementation of the n-bit full adder is largely equivalent to the one in the previous lab; we take the exclusive-or value of the carry-in and the carry-out values for the last module to determine whether or not the module has overflown. In order to account for the subtraction, an xor circuit with the subtraction flag as input flips b-input accordingly, as well as supplying the carry-in for the first module to increment the value by one (2's complement)

2.3 Multiplexer

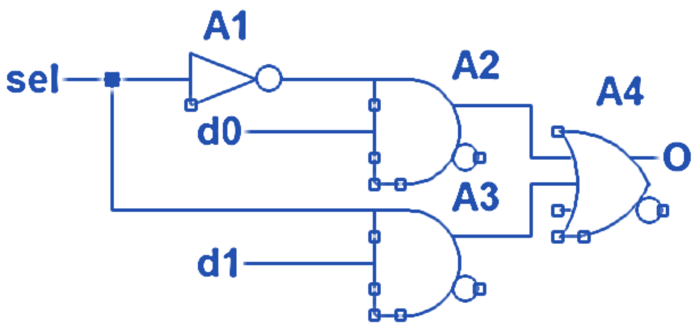

Figure 4: 1-bit multiplexer

We implemented an n-bit multiplexer by recursively iterating 1 bit multiplexers (Figure 4) as shown in Figure 5.

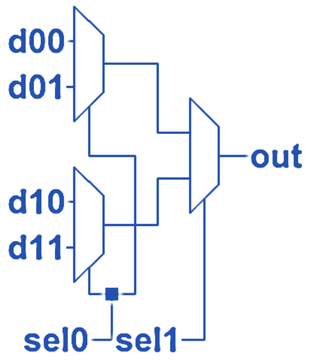

Figure 5: 2-bit multiplexer example built from recursion; as such, an n-bit multiplexer would be composed of $(2^n - 1)$ 1-bit multiplexers.

, with 3 layers where each layer selects each *address bit* of the output.

	Delay	**Area**
1-bit mux	70	8
n-bit mux	$70n$	$8(2^n - 1)$

Table 1: Performance Analysis of our mux implementation.

Because the mux is composed of a series of NOT, AND, and OR gates, the worst propagation delay would be the sum of the delay for the respective gates, i.e.

$$10ns_n + 30ns_a + 30ns_o = 70ns_t$$

While implementing, we were introduced to the concept of parametrization in Verilog, which allowed us to write generic modules without explicitly limiting the width of the module's buses.

2.4 Parametrization

```
// declaration
module my_module #(parameter n=32) ( args... );

// usage
my_module #(.n(4)) module_name( args... );
```

Listing 1: Example of a parametrized module.

Naturally, we applied this newly learned technique to our core modules; because it was difficult to keep track of the entire 32 bits during the test phase, we parametrized a generic module that could work for any width of input/output bits for the ALU. In this way, most of our verifications were done in 4-bits, the results of which were then extrapolated to 32-bit cases.

3 Test Results

For ADD and SUB Module, we selected 5 test cases for each of them. The first 4 tests are performed on a 4-bit base:

- Overflow with Carryout
- Overflow without Carryout

- No Overflow with Carryout
- No Overflow without Carryout

After the 4-Bit test, we also included a Full-bit (32 bit) test for each of the module with $A = 2^{32} - 1$

3.1 ADD

Table 2: ADD Test Cases Results

A	B	CMD	RES	COUT	ZERO	OVERFLOW	EXPLANATION
-5	-6	0	5	1	0	1	Overflow with Carryout
6	4	0	-6	0	0	1	Overflow without Carryout
-1	-6	0	-7	1	0	0	No Overflow with Carryout
1	5	0	6	0	0	0	No Overflow without Carryout
-1	1	0	0	1	1	0	Full bits test

We selected the ADD test cases such that we could test a zero case and all the 4 possible combinations of overflow and carryout.

3.2 SUB

Table 3: SUB Test Cases Results

A	B	CMD	RES	COUT	ZERO	OVERFLOW	EXPLANATION
-5	4	1	7	1	0	1	Overflow with Carryout
7	-6	1	-3	0	0	1	Overflow without Carryout
-1	2	1	-3	1	0	0	No Overflow with Carryout
5	6	1	-1	0	0	0	No Overflow without Carryout
-1	-1	1	0	1	1	0	Full bits test

The way we selected SUB test cases is similar to how we selected the ADD test cases.

3.3 SLT

Table 4: SLT Test Cases Results

A	B	CMD	RES	COUT	ZERO	OVERFLOW	EXPLANATION
7	2	2	0	0	1	0	A>B (Positive)
1	4	2	1	0	0	0	A<B (Positive)
-4	-7	2	0	0	1	0	A>B (Negative)
-6	-3	2	1	0	0	0	A<B (Negative)
-1	-1	2	0	0	1	0	A=B Full bits test

For SLT, we wanted to test representative cases which covers all the combinations of when A is equal to, greater or less than B, and when the result is negative or positive.

3.4 XOR, NAND, AND, NOR, OR

We performed 4 full-bit tests on each of the XOR, NAND, AND, NOR, and OR Module:

- $A = 2^{32} - 1$, $B = 2^{32} - 1$

- $A = 2^{32} - 1$, $B = 0$
- $A = 0$, $B = 2^{32} - 1$
- $A = 0$, $B = 0$

Table 5: XOR Test Cases Results

A	B	CMD	RES	COUT	ZERO	OVERFLOW
-1	-1	3	0	0	1	0
-1	0	3	-1	0	0	0
0	-1	3	-1	0	0	0
0	0	3	0	0	1	0

Table 6: NAND Test Cases Results

A	B	CMD	RES	COUT	ZERO	OVERFLOW
-1	-1	4	0	0	1	0
-1	0	4	-1	0	0	0
0	-1	4	-1	0	0	0
0	0	4	-1	0	0	0

Table 7: AND Test Cases Results

A	B	CMD	RES	COUT	ZERO	OVERFLOW
-1	-1	5	-1	0	0	0
-1	0	5	0	0	1	0
0	-1	5	0	0	1	0
0	0	5	0	0	1	0

Table 8: NOR Test Cases Results

A	B	CMD	RES	COUT	ZERO	OVERFLOW
-1	-1	6	0	0	1	0
-1	0	6	0	0	1	0
0	-1	6	0	0	1	0
0	0	6	-1	0	0	0

Table 9: OR Test Cases Results

A	B	CMD	RES	COUT	ZERO	OVERFLOW
-1	-1	7	-1	0	0	0
-1	0	7	-1	0	0	0
0	-1	7	-1	0	0	0
0	0	7	0	0	1	0

4 Timing Analysis

4.1 ADD, SUB, SLT

The propagation delay for ADD, SUB and SLT are the same because they are computed using the same ADDSUB module and the concatenation part for SLT was assumed to have 0 delay, as it is merely a wiring connection.

```
    `XOR xorGate(sum, a, b, carryin);
    `AND and0(AandB, a, b);
    `AND and1(AandC, a, carryin);
    `AND and2(BandC, b, carryin);
    `OR orGate(carryout, AandB, AandC, BandC);
```

Listing 2: structural part of FullAdder module

The 1 bit Full Adder has propagation delay of 1AND2 + 1OR3 which translates to 90 units. The xor for computing the sum, 1XOR3, is ignored since it takes less than 90 units to compute.

```
  generate
    genvar i;
    for (i=0; i<n; i = i+1) begin: addsubgenblk
      `XOR xoraddsub(xorB[i], operandB[i], sub);
      FullAdder fa(result[i], c[i+1], operandA[i], xorB[i], c[i]);
    end
   endgenerate

   assign carryout = c[n];
  `XOR xorGate(overflow, c[n], c[n-1]);
```

Listing 3: structural part of ADDSUB module

The ADDSUB module should take $1XOR2 + Delay(FullAdder)n + 1XOR2$ to reach a steady state for the worst case scenario when the XOR of operandB is computed concurrently. Using the delays we have defined for the gate, the ADDSUB module has propagation delay of $90n+100$, which is 2980 units for 32 bit module.

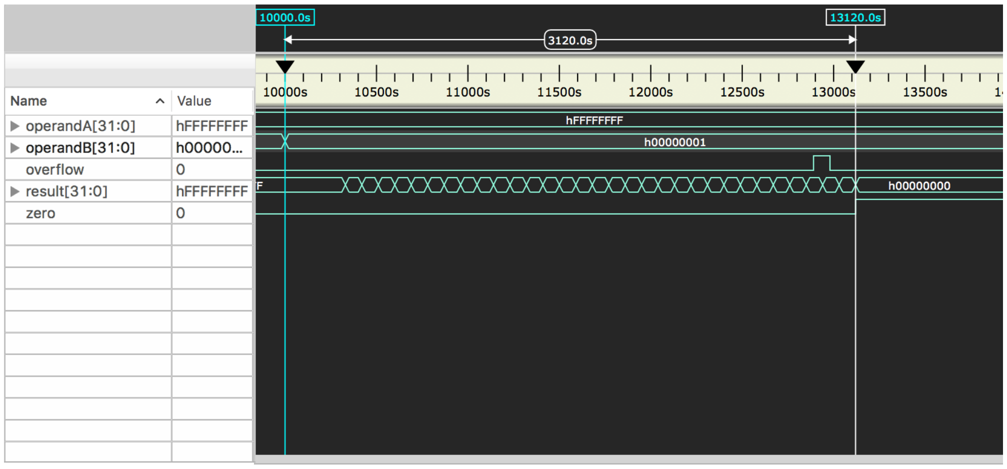

Figure 6: Worst Case Delay for ADD

One of the worst case delay for 32 bit ALU selecting ADD operator happens when A is kept at -1 (111...111) and B is changed from 0 to 1.

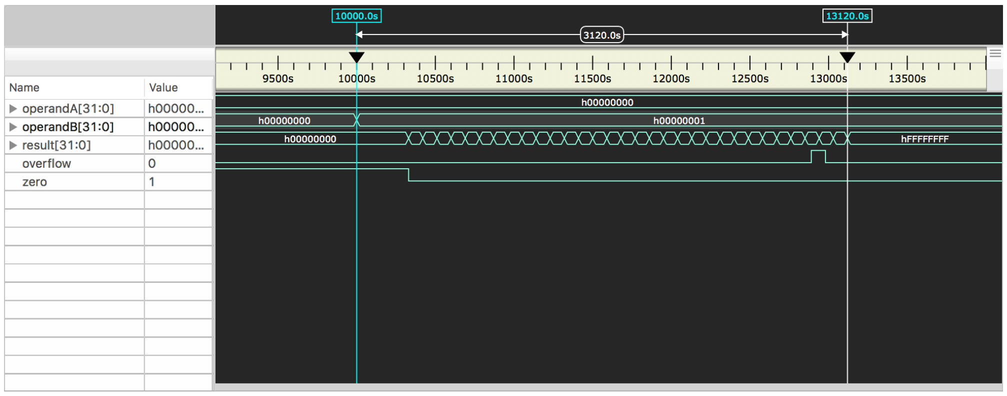

Figure 7: Worst Case Delay for SUB

The worst case delay used for generating the waveform of the 32bit ALU using SUB ooperator happens when A is kept at -1 (111...111) and B is changed from 0 to -1.

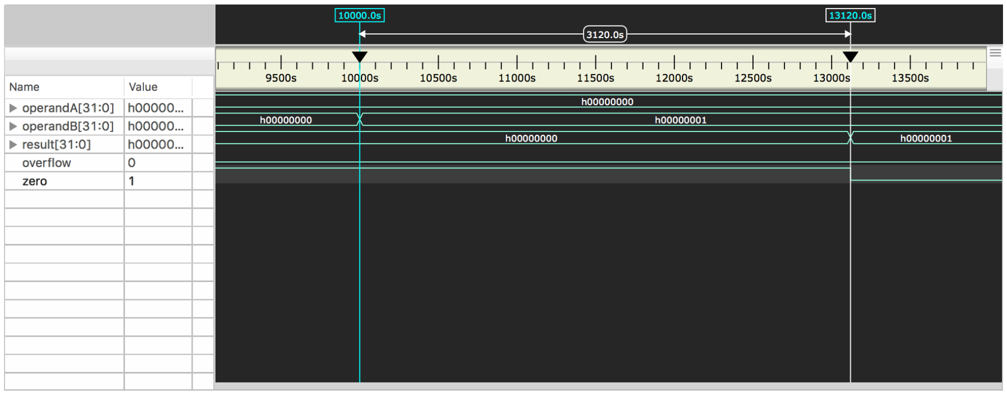

Figure 8: Worst Case Delay for SLT

Because the SLT makes use of the ADDSUB module, same case as SUB was used to generate the waveform of the worst case delay.

The worst case delay happens for ADDSUB when all the result bit change. The propagation delay is 3120s which is 120s off from the expected 3240s (2980 + 260, 260 is from final processing module after the mux). We tried to find the source of the error and checked how long each Full Adder took by diving into each component in wave generator. However, each full adder contributed to delay of 90 which match our expected value and we are confident that our calculation for ADDSUB module delay are right.

4.2 XOR

Each XOR gate was assumed to have a delay of 50 units. For our 32 bit ALU, the XOR module takes 50 units to compute since they are all computed concurrently.

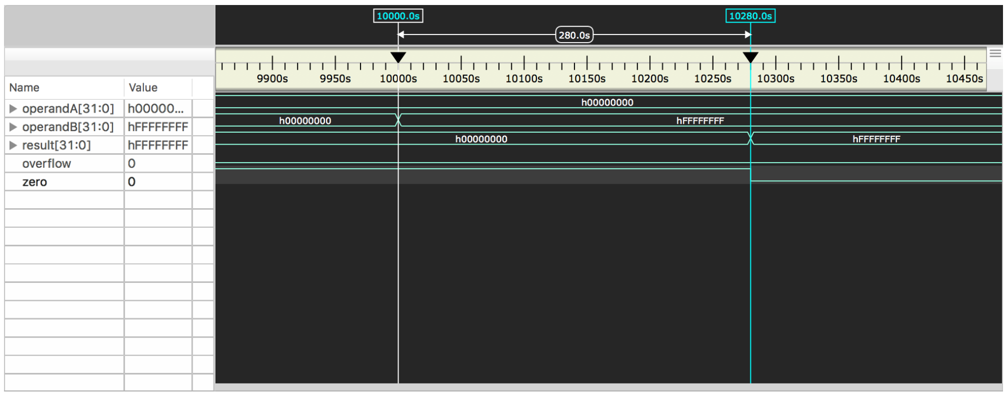

Figure 9: Worst Case Delay for XOR

For the XOR module, we kept the A at -1 and switched the B from 0 to 1. From the waveform in Figure 9, we can read that the worst case delay took 280s. This is off by 30s from the expected 310s (50+260).

4.3 NAND, AND, NOR and OR

s Each NAND and NOR gate was assumed to have a delay of 20 units. For our 32 bit ALU, the NAND and NOR module takes 20 units to compute since they are all computed concurrently. The AND and OR also uses NAND and NOR module. They are computed afterwards by inverting the output from the mux.

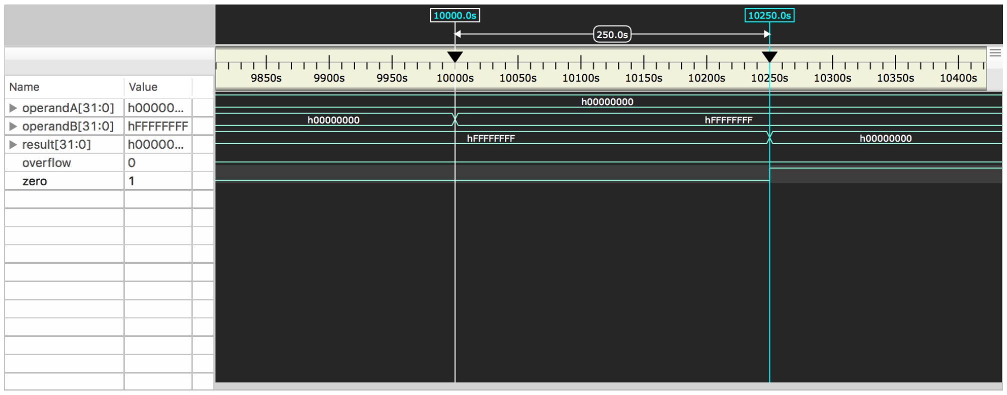

Figure 10: Worst Case Delay for NAND, AND, NOR and OR

The NAND, AND, NOR and OR all have the same worst case delay because the NAND and NOR gate has the same delay. The case selected for 10 was A remained at -1 and B switched from 0 to -1. The delay from the wave generation was 250s which is off by 30s from the expected 280s(20+260).

4.4 ALU

The ALU is composed of the core modules, the multiplexer, and the final processing modules for setting the flags. The ADDSUB, XOR, NAND and NOR module has worst case delay of 2980, 50, 20, 640 respectively for our system. The 3 bit multiplexer has delay of 210 ($70 * 3$). The final processing module consist of XOR of mux output with invertOutput to compute the result, and a NOR of result to check for zero flag. The XOR takes 50 units because it is computed cocurrently, and the NOR takes 0 unit for our system.

The worst case delay happens SUBADD module is used

$$(90n + 100) + 210 + 50 = 90n + 360$$

when n = 32,

$$2980 + 210 + 50 = 3240 units$$

We suspect that there are errors from outside the core modules because the offset for NAND/AND/NOR/OR gates match with the offset for XOR. There might be some computation we didn't account for going on outside the core modules.

We can reduce the delay by 50 units by moving the XOR after the mux (for inverting the output) into the NAND and NOR module. If we are not concerned about area, we can also create separate module for SUB and use more time efficient NOT gate on the operandB instead of the XOR gate.

5 Work Plan Reflection

The following is a copy of our initial Work plan:

- ALU Implementation: 3 Hr. 20 min. By Oct. 4th
  - High-Level Design : 1 Hr.
    Figure out how the modules connect, emit flags, etc. and draw the high-level block diagram
  - Gate Delay : 20 min.
    Decide how long each of the gates will take to execute.
  - Modules : 2 Hr.
    Complete all the 8 modules in 80 min. with 10 min. for each module
    Debug Buffer : 40 min.
- Test Benches : 3 Hr. By Oct. 4th
  - Implementation : 1 Hr.
  - Test Cases : 126 min.
    For each module, come up with 4 test cases. 3 min for each test case. 3 x 4 x 8 = 96 min.
    Debug Buffer : 30 min.
- Analysis : 1 Hr. By Oct. 5th
  - Worst Propagation Delay Calculation : 30 min.
  - Worst Propagation Delay Simulation and Identification : 30 min.
- Report : 1 Hr. By Oct. 5th
  - Wrap up and finish lab report : 1 Hr.

Our actual implementation took longer than what we have initially expected.
The time used for writing out the code for ALU and test benches were within our expectation. The debugging process, however, took much longer than we expected. We could debug syntax error using the error output. However, when we got logic errors (from the test bench failures), we had to go through the logical flow of the code and that took a while to debug.
Also, some implementation did not work the way we expected it to. At first, we computed the zero (zero flag) by nor (zero, result) and expected the nor to iterate through all the bits in result such that it was equivalent to nor(zero, result[0], result[1], ... result[n]). However, it was equivalent to nor(zero, result[0]). Because this was a simple nor logic, we did not realise that it was giving erroneous outputs until quite late into the lab.
In hindsight, we should have allocated more time for debug buffer for implementation and testing the test cases.
We have planned for the report to take an hour from our Lab 0 experience. However, even though the required sections are similar, the ALU requires more explanation because it is made up of different components. Generating tables and figures and writing out the analysis took longer than we planned. In the future, we will allocate more time to write the report and to go through the report afterwards.