Lab 2: ??????????

# Design Explanation

## NAND Implement Lawrence

1. Design Concepts

We first define AND flag which maps to AND gate, OR flag maps to OR gate, etc… Next, we encode SEL to AND/OR/… flags via one hot encoding. That is, when AND flag is on, the circuit behave like AND gate.

We define the NAND binary operation as , that is, we define A NAND B as . is the gate output, is the flag and is the output. We have the following equations.

A simple MUX works as following.

We could prove the previous equations are equivalent by De Morgan’s law.

We know that . We have finished the construction of NAND operation.

We know that . We have finished the construction of AND operation.

We know that . We have finished the construction of OR operation.

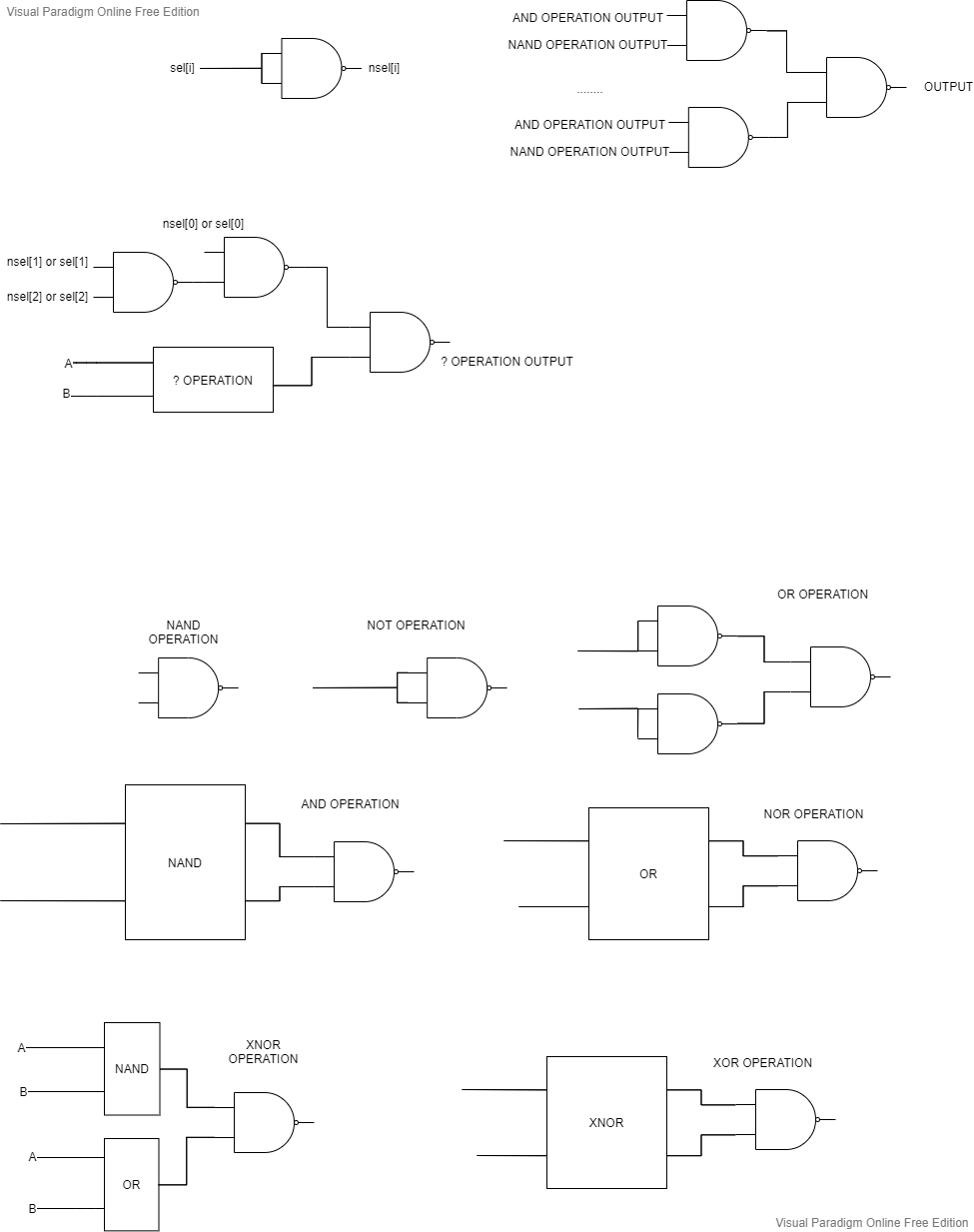
We know that . We have finished the construction of OR operation.

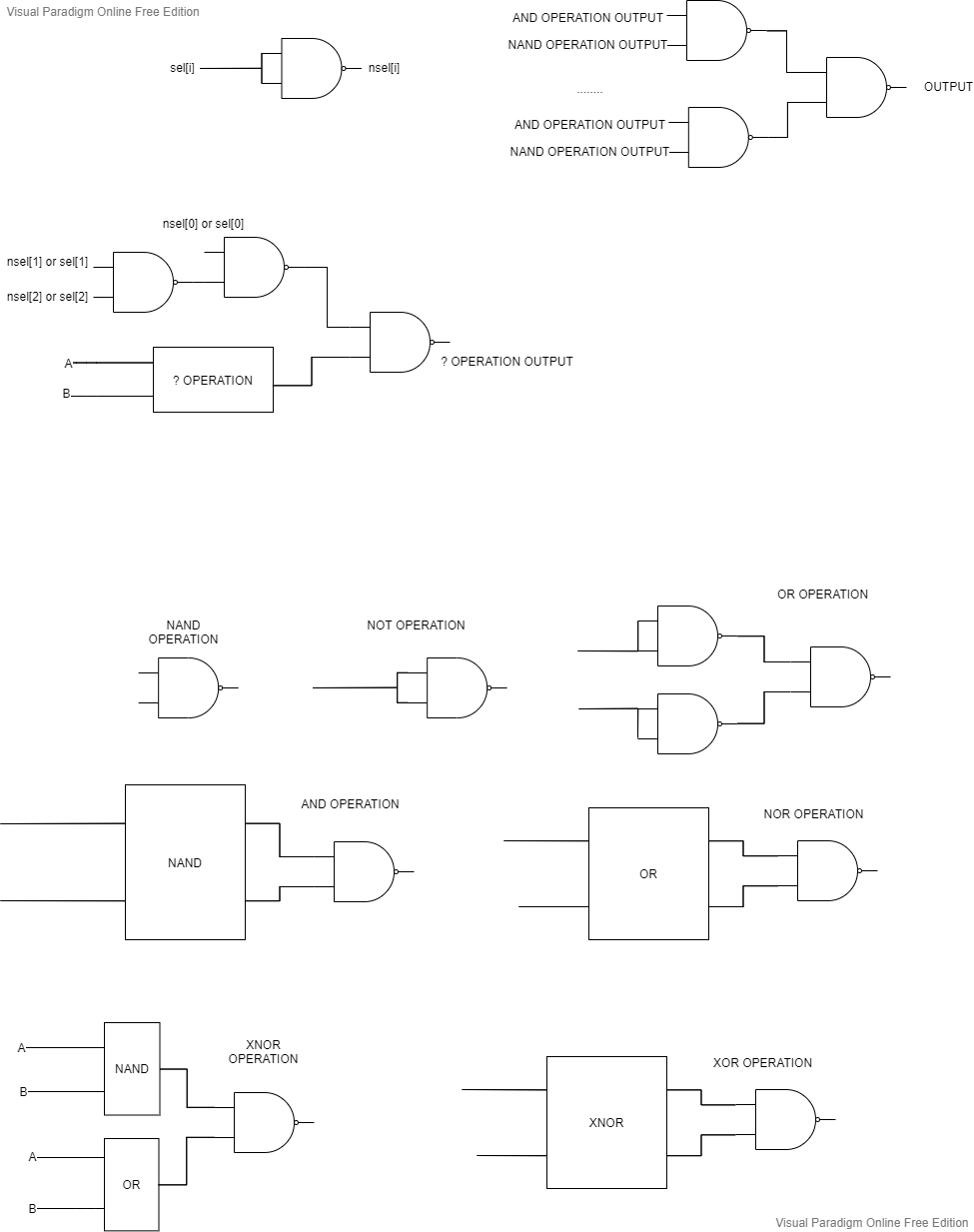
We know that . We have finished the construction of XNOR operation.

We know that . We have finished the construction of NOT operation.

We apply conjunction on XNOR operation and NOT operation, we have XOR operation.

1. Schematic Graph

The following is the basic gates in schematic graph.

The following is the circuits handles flags and mux.

## NAND Implement Ariel

## Ripple Adder

## Decode and Execute

## Carry Lookahead Adder

1. Circuits Explanation

We divide CLA (Carry Lookahead Adder) into 3 modules, adder unit, 4-bit CLA generator, 2-bit CLA generator.

First, the adder unit process inputs into p (propagate) flag and g (generate) flag. Second, 4-bit CLA process p flag and g flag into c (carry) flags. 4-bit CLA would also condense p flags and g flags. Last, 2-bit CLA generator process the condensed p flags and g flags into c flag.

Such procedure could significantly reduce the latency from gates to gates, we will dig into the benefits in following discussion.

1. Schematic Graph of 4-bit Carry Lookahead Generator

WIP.

1. How it works?

First, we define two flags, p (propagate) flag and g (generate) flag.

P flag means this bit would propagate a carry. That is, if previous bit sent a carry to this bit, this bit would send a carry to next bit. G flag means this bit would generate a carry. That is, no matter the previous bit sent a carry or not, this bit would always send a carry to next bit.

By the above equations, we know that and does not rely on carry signal, which made parallelization possible. We shall compute carry signals, by the equations below.

If and only if the p flag is turned on, . Thus, if is also turned on, the last digit of would be 0, otherwise the last digit would be 1. Such property can be expressed as where is the last digit of .

Output and the greatest bit of , the circuit is done.

1. Benefits

A ripple adder is a processing chain. We compute the first bit and carry the overflow to next bit. Next, we compute the second bit and carry the overflow to next bit, on and on.

Since ripple adder is combinatorial circuit, we shall reform the circuit to a DAG (Direction Acyclic Graph) by replacing gates with nodes and wires with edges.

Picking the greatest carry flag as root of the tree, the depth of the tree is if the ripple adder supports -bit addition. That is, it takes gates from input to output for a -bit addition. Since it takes to propagate through all gates in the tree, the latency from input to output is around .

A carry lookahead adder is a flattened processing graph. Picking the greatest carry flag as root, the depth of the tree is only if the CLA supports -bit addition. That is, the depth of the tree is not related to the bits of addition. Despite the propagate latency stays constant in respect to , CLA requires gates and wires. Therefore, we should always consider the space-time trade off on CLAs.

In practice, we divide CLA to smaller CLAs. Smaller CLAs leads to less gates thus less power consumption. In this case, we sacrificed time to consume less space.

In a nutshell, ripple adders require gates and takes to compute. In contrast, CLAs require gates and wires but takes only to compute. Also, divide CLA into multiple smaller CLAs could significantly reduce space in practice.

## 4-bit Multipler

1. How it works?

We know that where is the product of and . Reform the equation, we have

Since we are implementing a 4-bit multipler, the final product can always be stored with 8-bits. Thus, the 8-bit CLA previously designed could come handy.

We have only 4-bits to product, that is, there is only 16 terms of , which can be easily brute forced. The term is frankly shifting left bits and the term is shifting left bits.

We shall develop a circuit by the above knowledge. We keep as constant and as variable, we shall compute by placing at the corresponding bit. Next, we shift by bits. Using the 8-bit CLA previously designed, we shall compute .

1. Schematic Graph

WIP.

## Exhausted Testbench

# Contribution

## Lawrence Wu

## Ariel Chang

# What have we learned?

Layering up gates could build skyscrapers.

Circuits works like parallel programming.

When the objective is clear, building skyscrapers is not hard.

Word is horrible when it comes to mathematical equations.

Visio is your savior when you meet schematic graphs.