Lab 2: Advanced Gate-Level Verilog

Group 3 : 109060013 張芯瑜　109062328 吳邦寧

Design Explanation

1. NAND Implement

(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 $\circ$, that is, we define A NAND B as $A \circ B$. X is the gate output, F is the flag and O is the output. We have the following equations.

$$O = (F_{and} \circ X_{and}) \circ (F_{or} \circ X_{or}) \circ (F_{xor} \circ X_{xor}) \circ \dots$$

A simple MUX works as following.

$$O = (F_{and} \cdot X_{and}) + (F_{or} \cdot X_{or}) + (F_{xor} \cdot X_{xor}) + \cdots$$

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

$$O = \overline{\overline{(F_{and} \cdot X_{and})} \cdot \overline{(F_{or} \cdot X_{or})} \cdot \overline{(F_{xor} \cdot X_{xor})} \cdot \dots}$$

We know that $A \circ B = A \circ B$. We have finished the construction of NAND operation.

We know that $A \cdot B = (A \circ B) \circ (A \circ B)$. We have finished the construction of AND operation.

We know that $A + B = (A \circ A) \circ (B \circ B)$. We have finished the construction of OR operation.

We know that $\overline{A + B} = \big((A \circ A) \circ (B \circ B)\big) \circ \big((A \circ A) \circ (B \circ B)\big)$. We have finished the construction of OR operation.

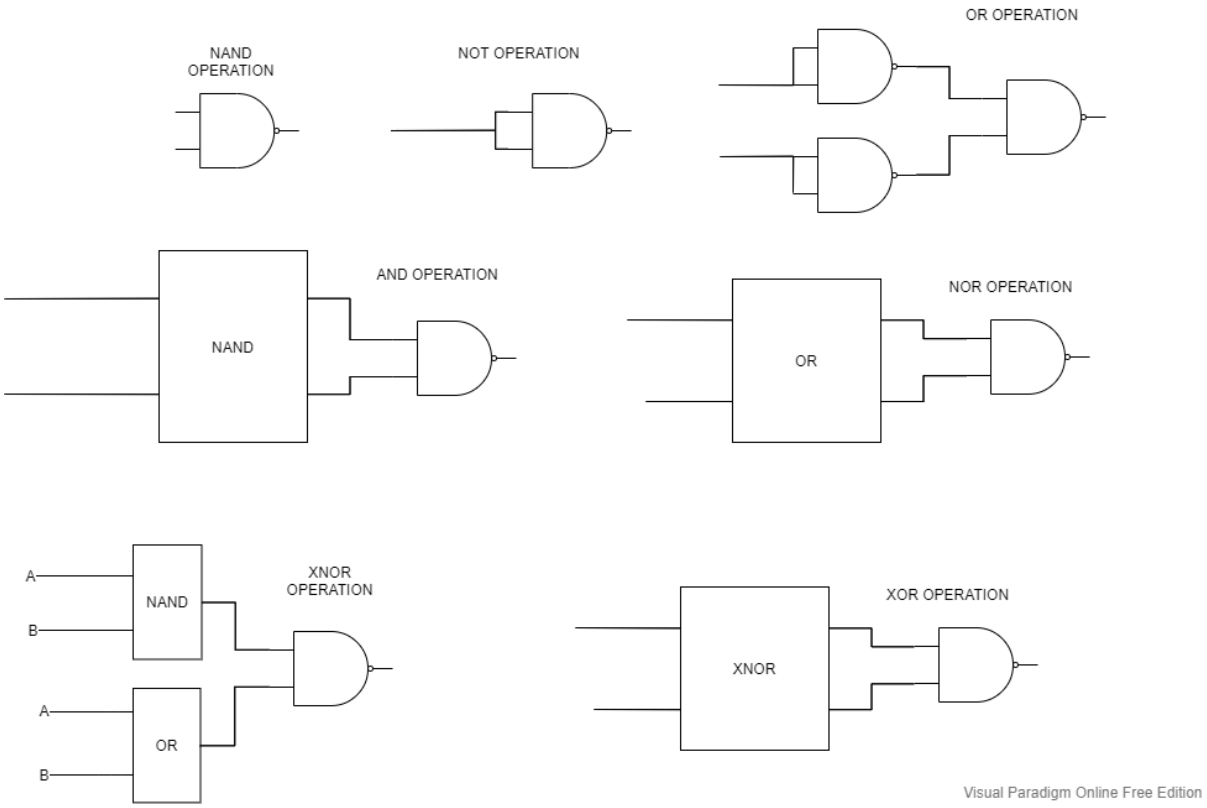
We know that $\overline{A \oplus B} = (A \circ B) \circ \big((A \circ A) \circ (B \circ B)\big)$. We have finished the construction of XNOR operation.

We know that $\bar{A} = A \circ A$. We have finished the construction of NOT operation.

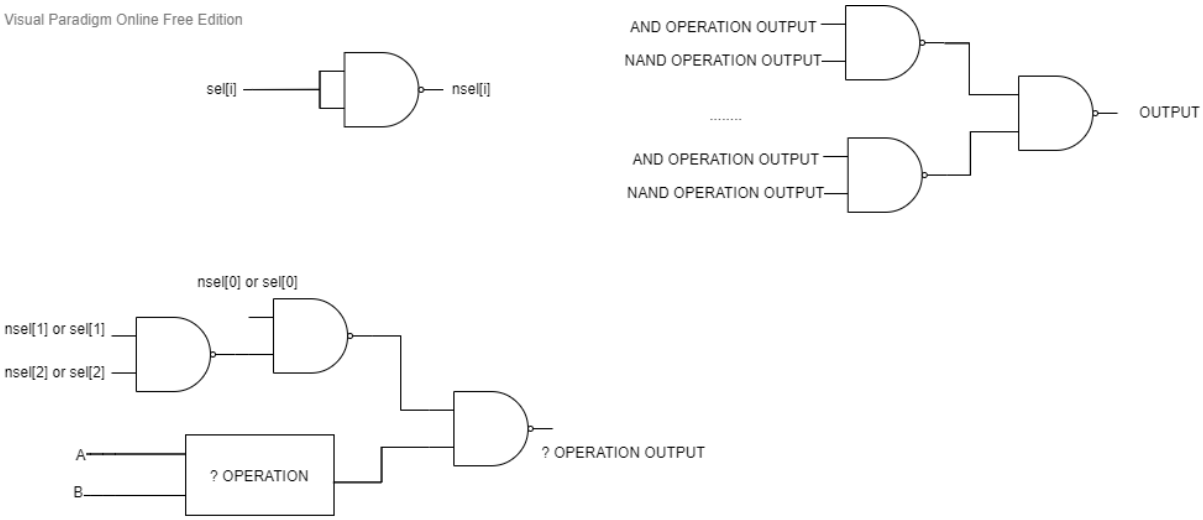
We apply conjunction on XNOR operation and NOT operation, we have XOR operation.

(2) Schematic Graph

The following is the basic gates in schematic graph.



The following is the circuits handles flags and mux.



(3) Validation

Apply the given testbench & manually check.

2. Basic Question 3

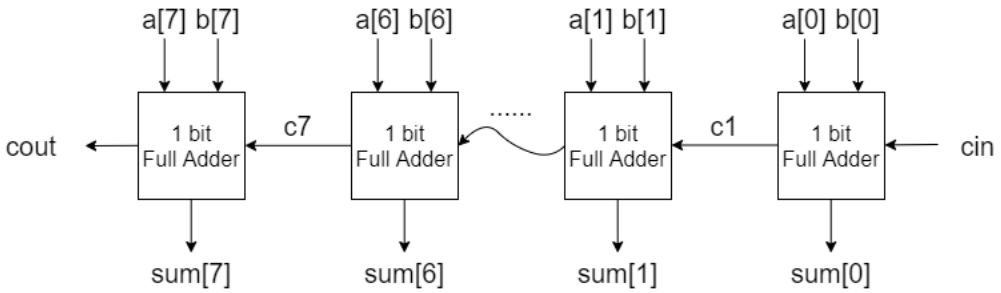
半加法器只有兩個輸入腳位，而全加法器有三個輸入腳位，所以全加法器可以接收前一位置的進位。

3. Ripple Adder

(1) Design Concepts

Since we have designed full adders, simply chain them together. We chain them by inputtin the previous cout into the next adder's cin, soit can satisfy doing carry out.

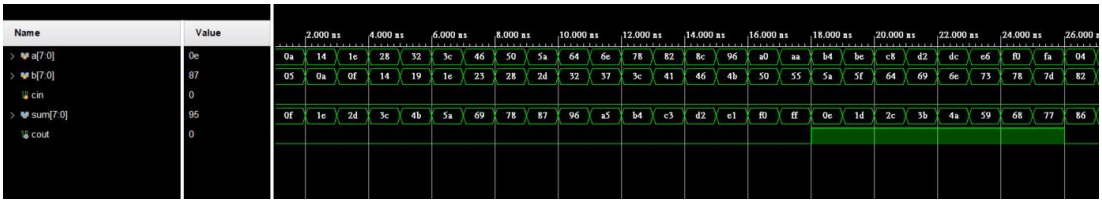
(2) Schematic Graph



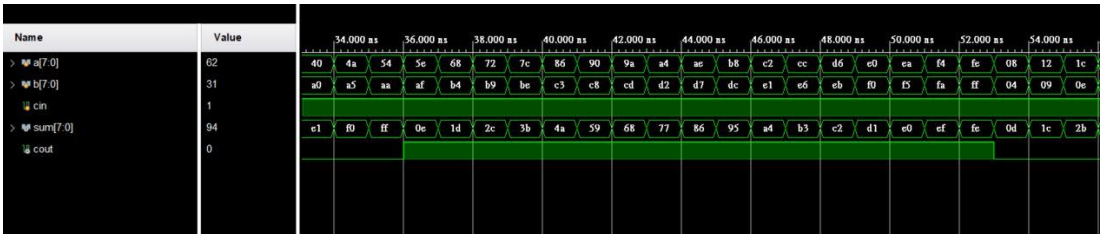
(3) Validation

In our testbench, we check both cin = 0 and cin = 1 conditions. We change a and b's signal in order by adding numbers to them. In order to check if the cout signal is going well, we run 32 groups for both conditions. Then we add them manually to check if the answer is correct.

- cin = 0 :



- cin = 1 :



4. Decode and Execute

(1) Design Concepts

First, we design the basic gates: AND, OR, NOT, XOR, XNOR. Then we design

each function.

1) ADD:

   We build a half adder first. By using it, we build a full adder and chain them together. The way we chain them together is same as the previous question.

2) SUB:

   First, invert each bit of 'rt' (the one which need to minus), and we will get one's complement of 'rt'. Then add one to it, and we will get two's complement of 'rs'. Then we use the first function 'ADD' to add 'rs' with two's complement of 'rt'.

3) BITWISE AND:

   We use AND gate, which we build first as a basic gate, to do bitwise and for every bit.

4) BITWISE OR:

   We use OR gate, which we build first as a basic gate, to do bitwise and for every bit.

5) RS LEFT SHIFT:

   We connect the signal of rs[n] to out[n+1], which make it left shift. For rs[3], we connect it to out[0].

6) RT RIGHT SHIFT:

   We connect the signal of rt[n] to out[n-1], which make it left shift. For out[0], we connect it to rt[3].

7) COMPARE EQ:

   We use XNOR to check if each bit of 'rs' and 'rt' are both 1 or 0. Then use OR to check that every bits are the same.

8) COMPARE GT:

   If it's greater, then it should satisfy one of these:

   - rs[3] > rt[3]
   - rs[3]=rt[3] and rs[2]>rt[2]
   - rs[3]=rt[3] , rs[2]=rt[2] and rs[1] >rt[1]
   - rs[3]=rt[3] , rs[2]=rt[2] , rs[1] =rt[1] and rs[0]>rt[0]

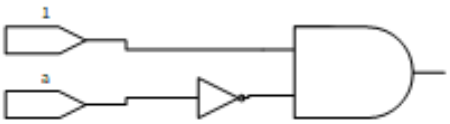
   We check these signal by OR, if one of the above condition appears, rs is greater then rt.

Then in our main module, we use gates to identify the selection signal and use SELECT module to select the output for each bit.

(2) Schematic Graph

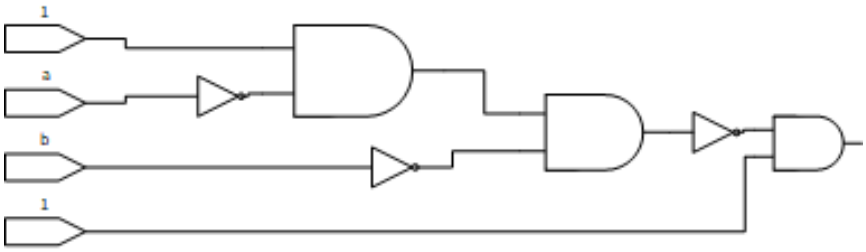
- Basic Gate:

NOT

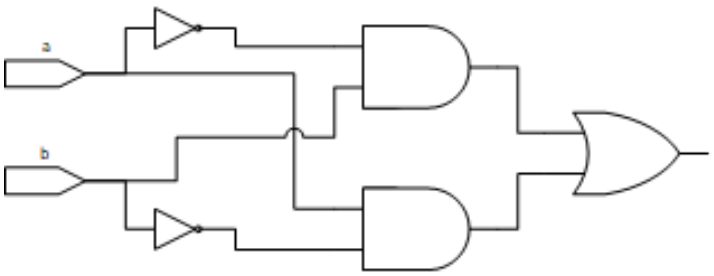


AND

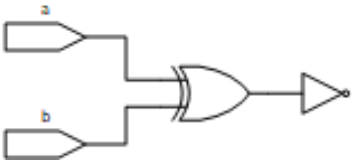
OR



XOR

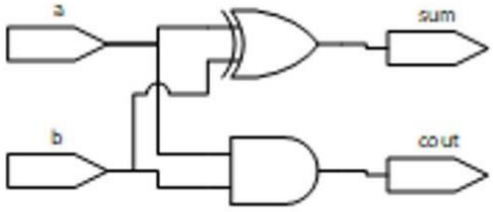


XNOR

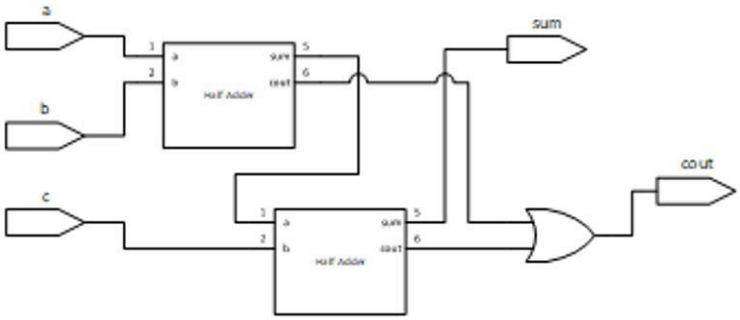


- ADD

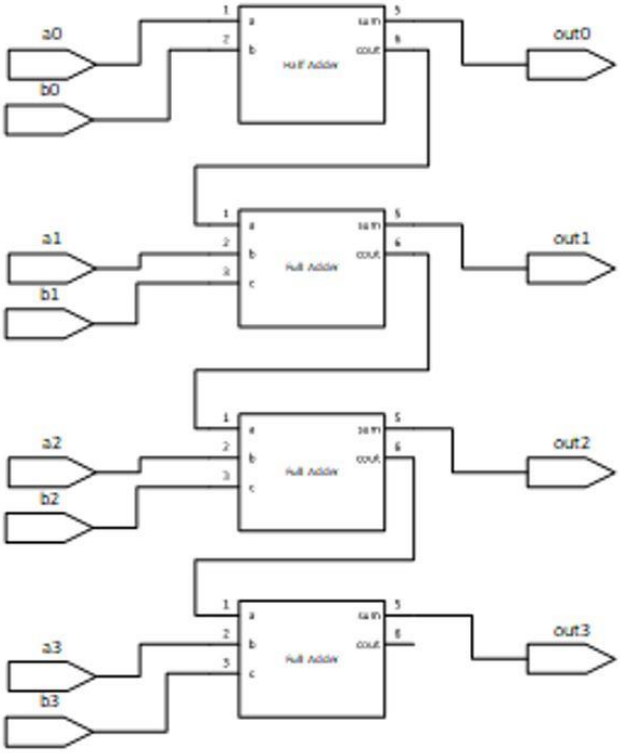
Half Adder



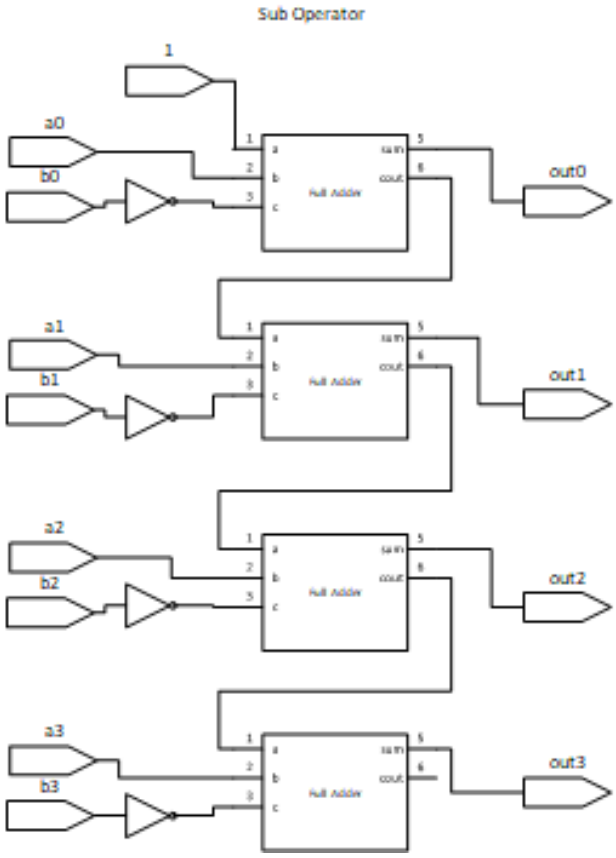
Full Adder



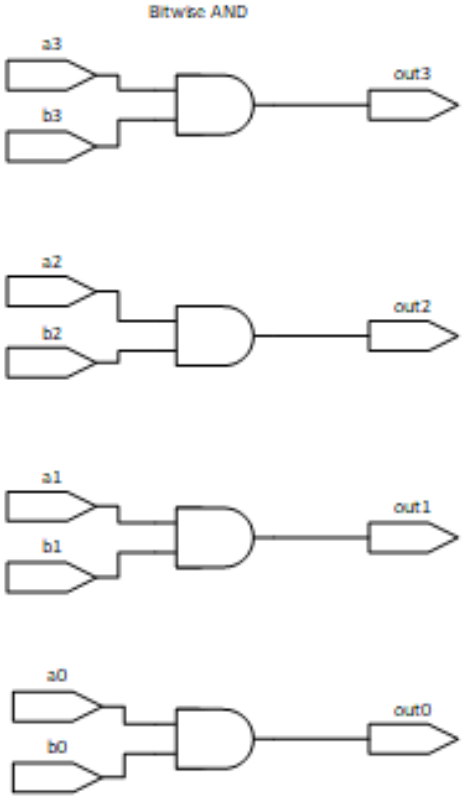
Add Operator



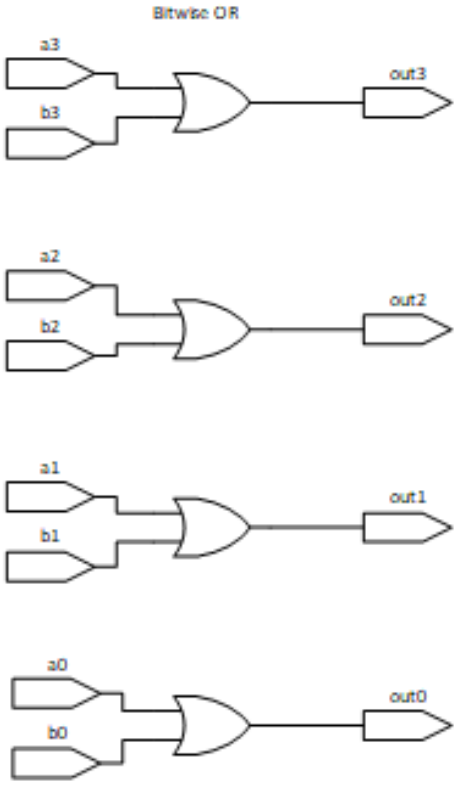
- SUB



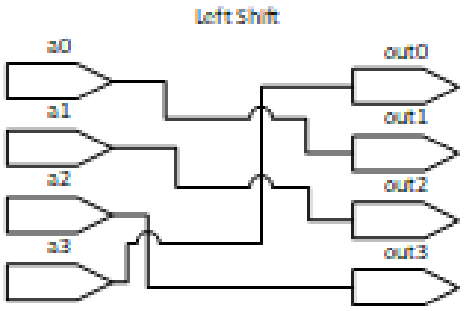
- BITWISE AND



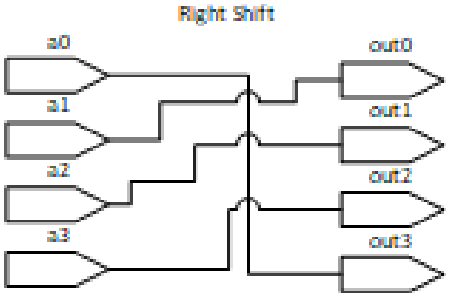
- BITWISE OR



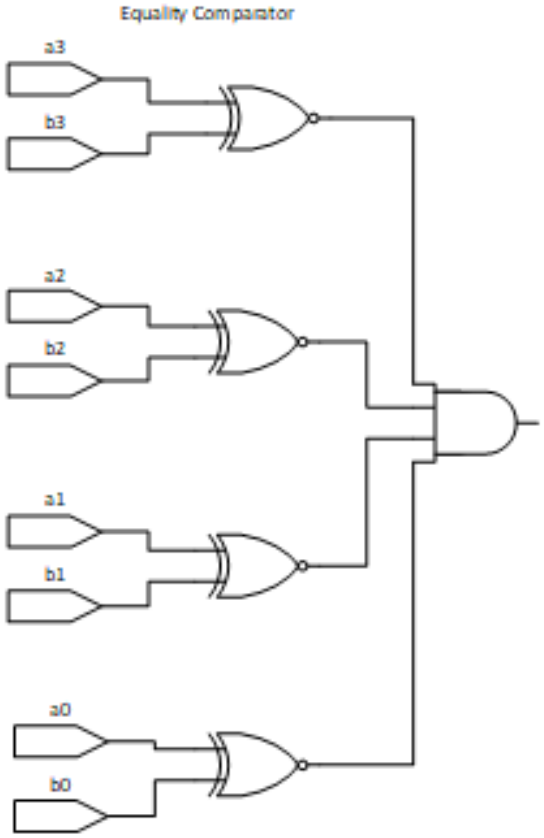
- RS LEFT SHIFT



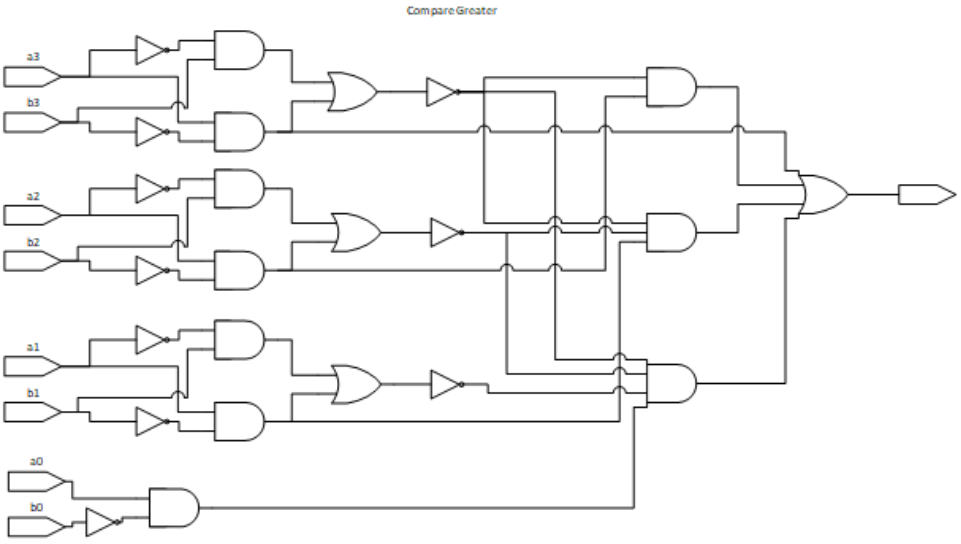
- RT RIGHT SHIFT



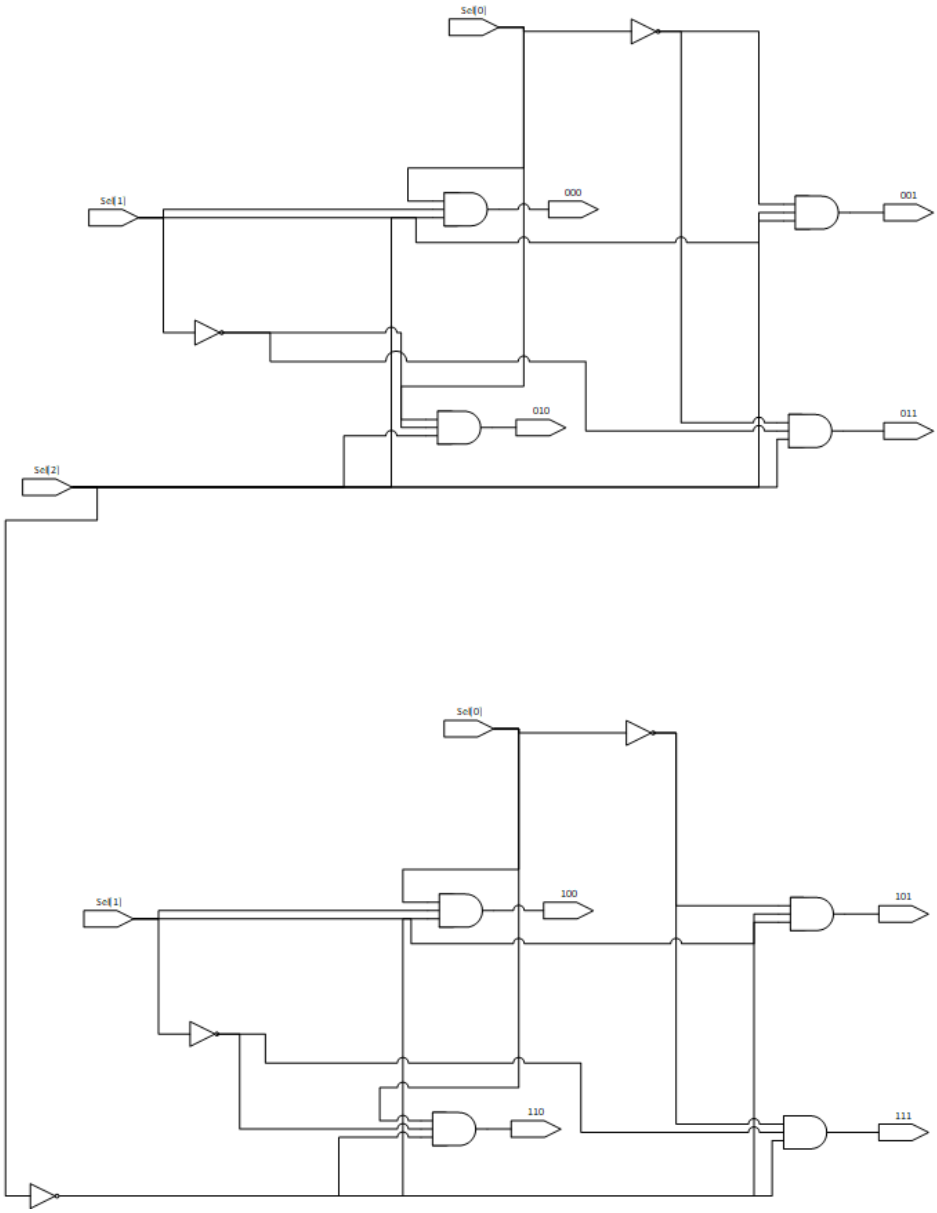
- COMPARE EQ



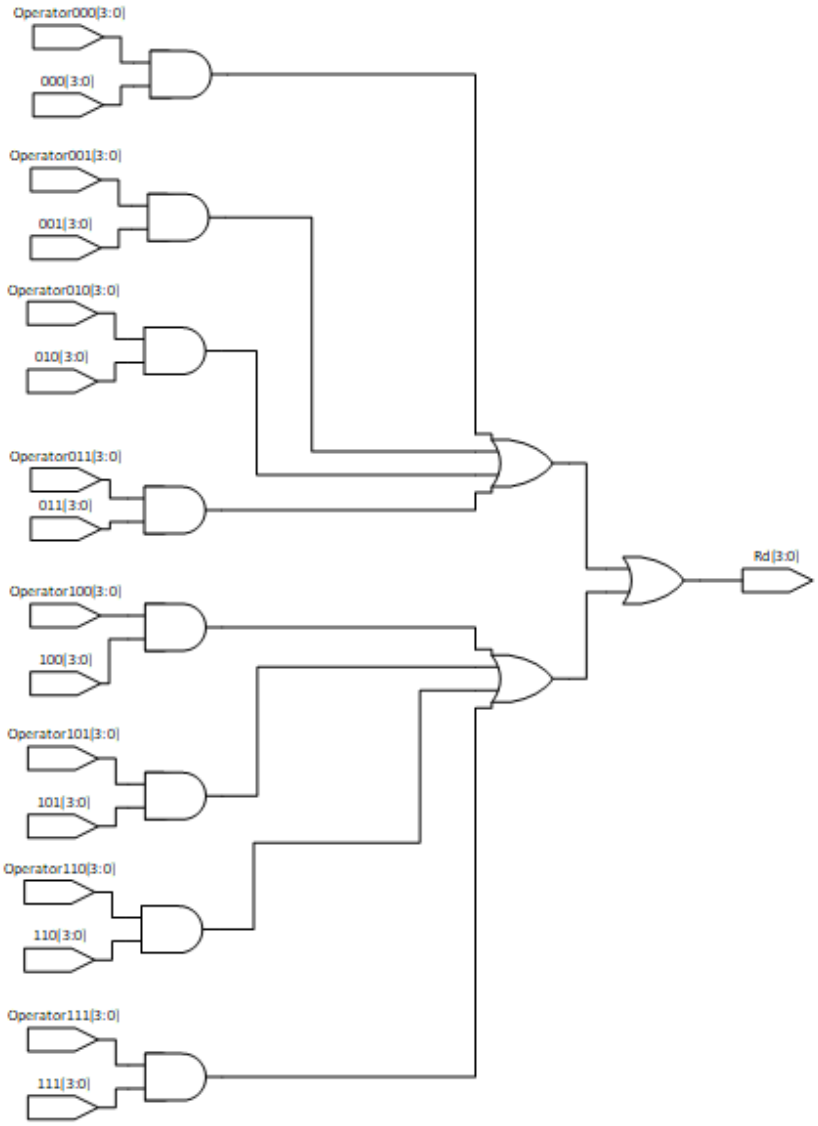
- COMPARE GT



- Identify the Selection Signal



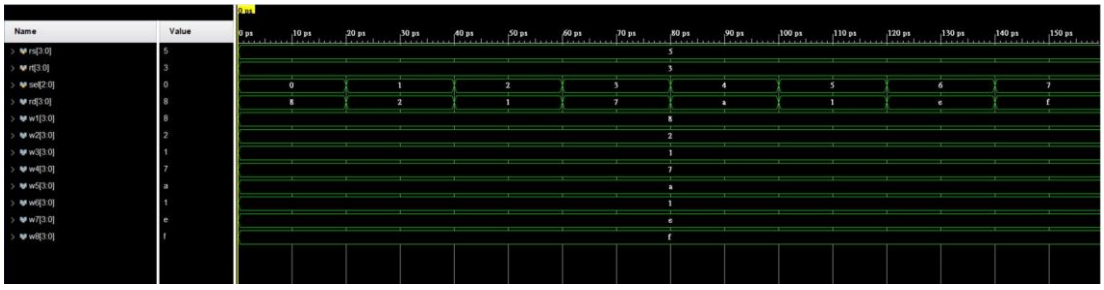
- SELECT



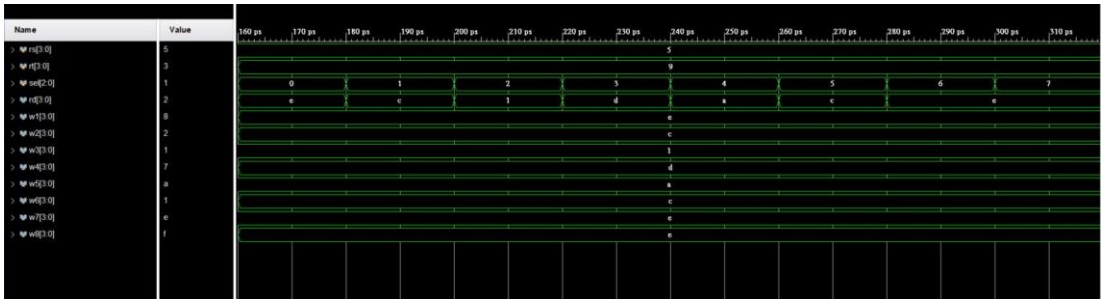
(3) Validation

We choose four groups of inputs of ‘rs’ and ‘rt’. Then for each ‘rs’ and ‘rt’, we change selection signal and check the output. Some of them are mainly for specific functions.

1) rs = 4’b0101, rt = 4’b0011

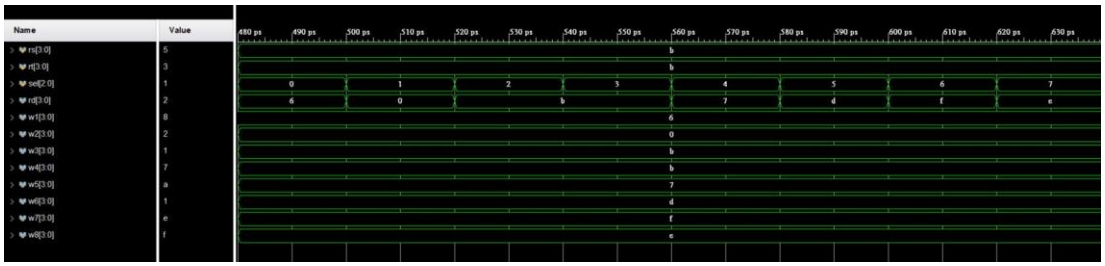


2) rs = 4’b0101, rt = 4’b1001



3) rs = 4’b1101, rt = 4’b1001
(Mainly to check function 2, 7)

4) rs = 4’b1011, rt = 4’b1011
(Mainly to check function 7, 8)



5. Carry Lookahead Adder

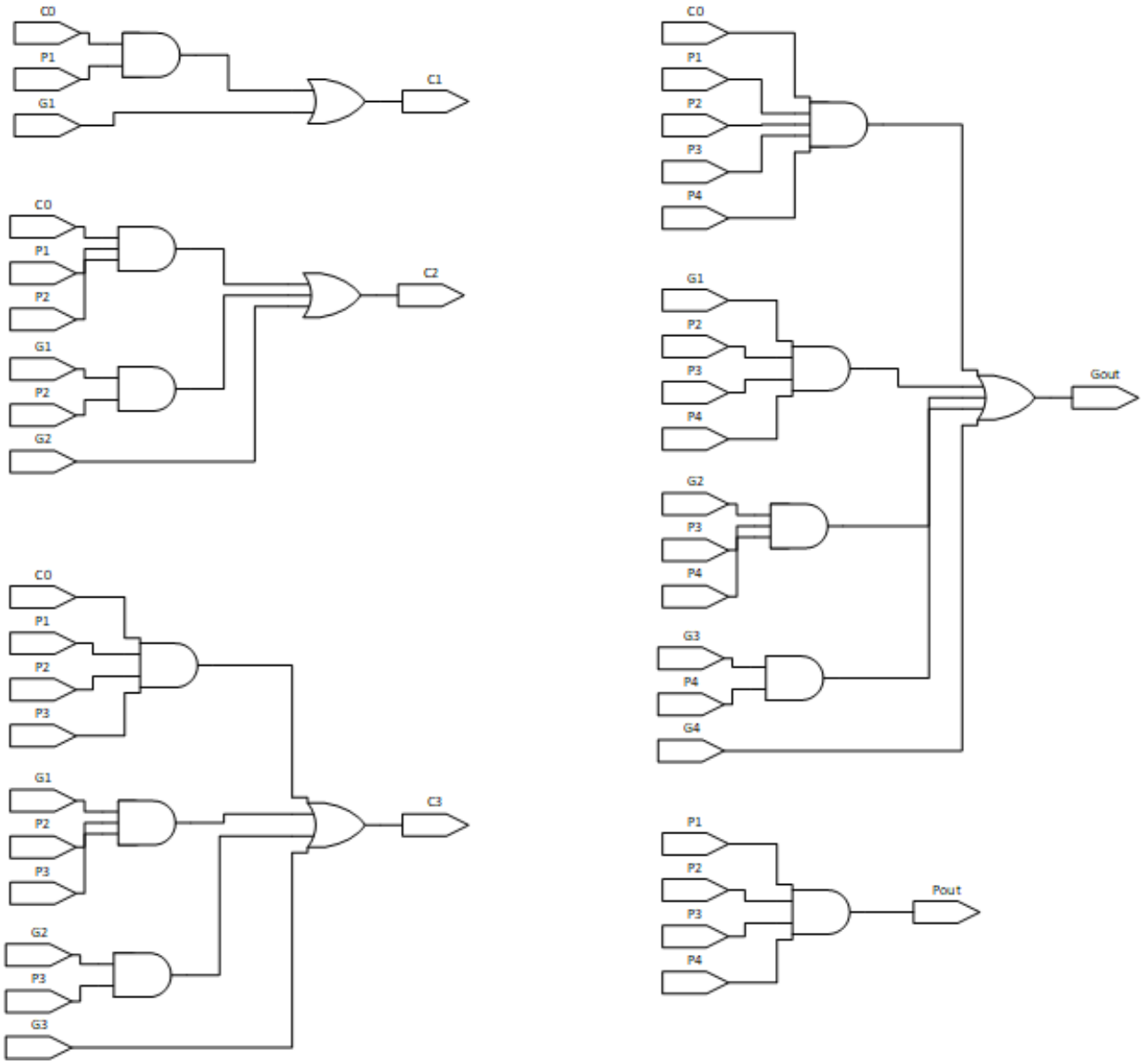
(1) Circuits Explanation

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.

(2) Schematic Graph of 4-bit Carry Lookahead Generator



(3) 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.

$$P_i = A_i + B_i$$
$$G_i = A_i \cdot B_i$$

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

$$C_i = C_0 \prod_{k=0}^{i-1} P_k + \sum_{j=0}^{i-1} G_j \prod_{k=j+1}^{i-1} P_k$$

If and only if the p flag is turned on, $A_i + B_i = 1$. Thus, if C_i is also turned on, the last digit of $A_i + B_i + C_i$ would be 0, otherwise the last digit would be 1. Such property can be expressed as $P_i \oplus C_i = S_i$ where S_i is the last digit of $A_i + B_i + C_i$.

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

(4) 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 $O(N)$ if the ripple adder supports N-bit addition. That is, it takes $O(N)$ gates from input to output for a N-bit addition. Since it takes $O(N)$ to propagate through all gates in the tree, the latency from input to output is around $O(N)$.

A carry lookahead adder is a flattened processing graph. Picking the greatest carry flag as root, the depth of the tree is only $O(1)$ if the CLA supports N-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 N, CLA requires $O(N^2)$ gates and $O(N^3)$ 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.

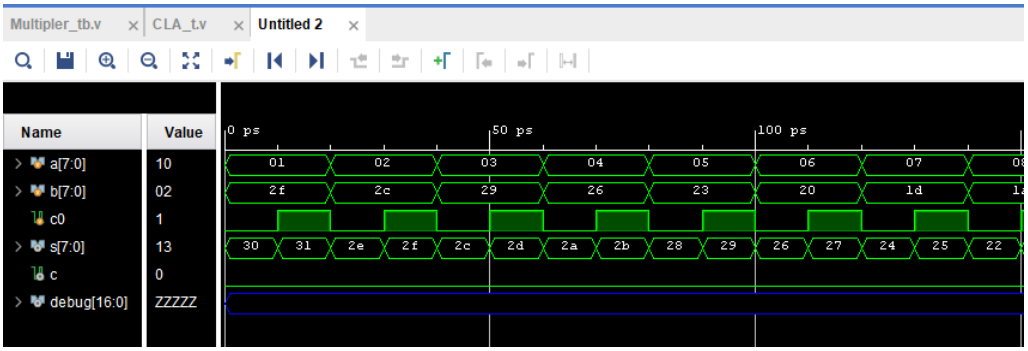
In a nutshell, ripple adders require $O(N)$ gates and takes $O(N)$ to compute. In contrast, CLAs require $O(N^2)$ gates and $O(N^3)$ wires but takes only $O(1)$ to compute. Also, divide CLA into multiple smaller CLAs could significantly reduce space in practice.

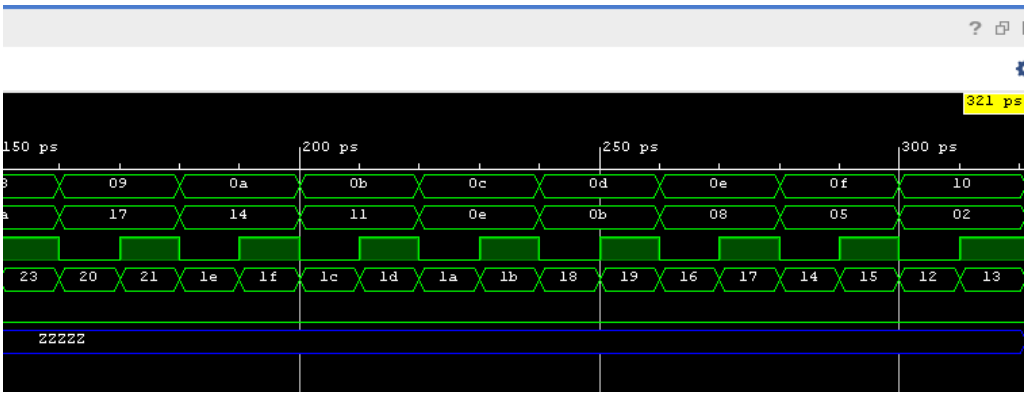
(5) Validation

Initially, we let a = 0 and b = 50.

Repeat the following procedure for 16 times.

First, we let a += 1, b -= 3 and c0 = 0.





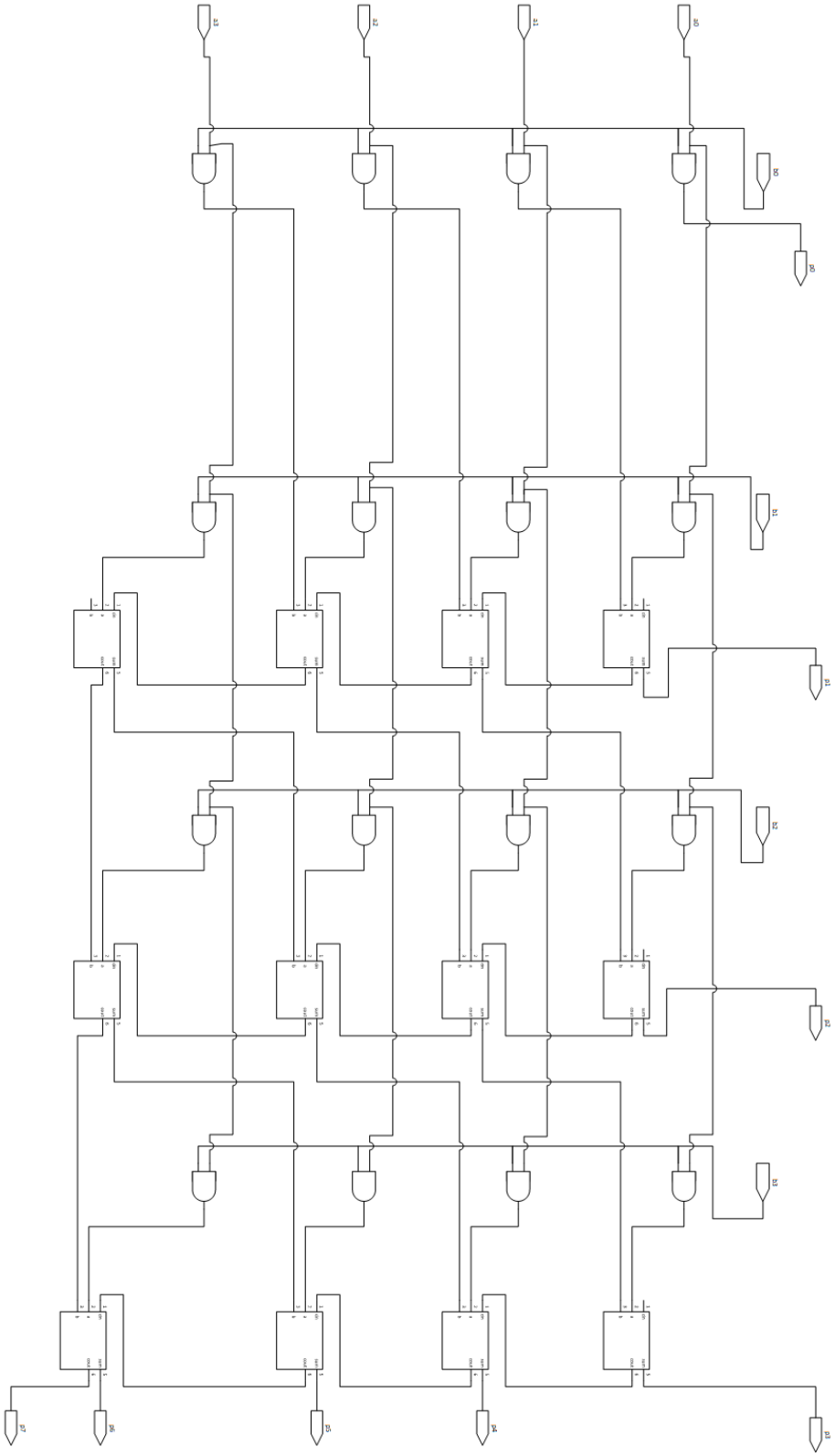
Second, we let c0 = 1.

6. 4-bit Multipler

(1) How it works?

Follow the given spec & brute force implementation, the design shall be made. Also, the 1-bit multiplication can be implemented by AND gate.

(2) Schematic Graph

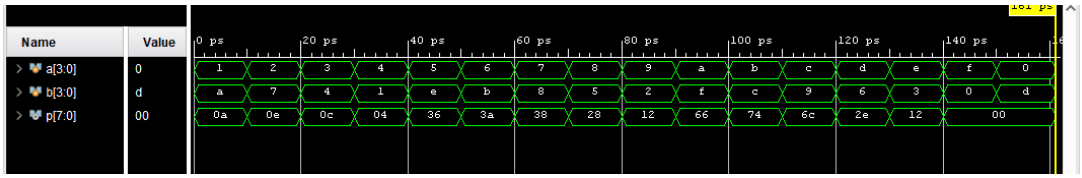


(3) Verification

Initially, we let a = 0 and b = 13.

And we repeat the following procedure 16 times.

Let a += 1 and b -= 3.



7. Exhausted Testbench

(1) Design Concepts:

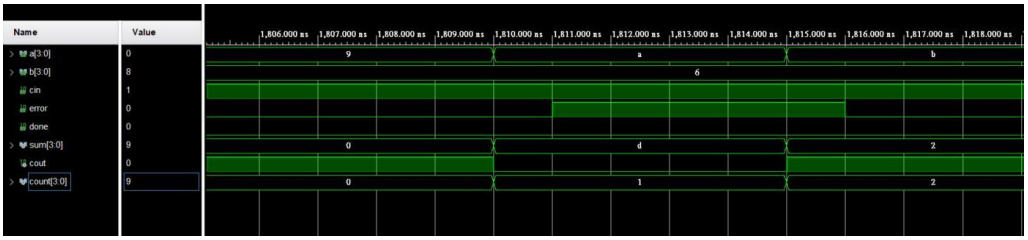
In order to run all over every possible inputs, we use repeats to make loops. In the loops, we use adding to change the inputs. To get the right signal, we check the inputs after it change for 1ns, and the error signal will keep up for 5ns if it's wrong. Last, we pull up the done signal after it done all of the possible inputs for 5ns, and it also keep up for 5ns.

(2) Verification

We use a adder to check our testbench.

First, we check if it goes well when adder is correct. In this condition, error should never be pulled up. We can check if the done signal pulled up correctly at the last.

Second, we check if it goes well when adder is not correct. In this condition, error should be pulled up several times. We choose some of them to check if it is really wrong or not. We also check does the signal pulled up at the right time.



8. Decode and Excute on FPGA

(1) Design Concepts:

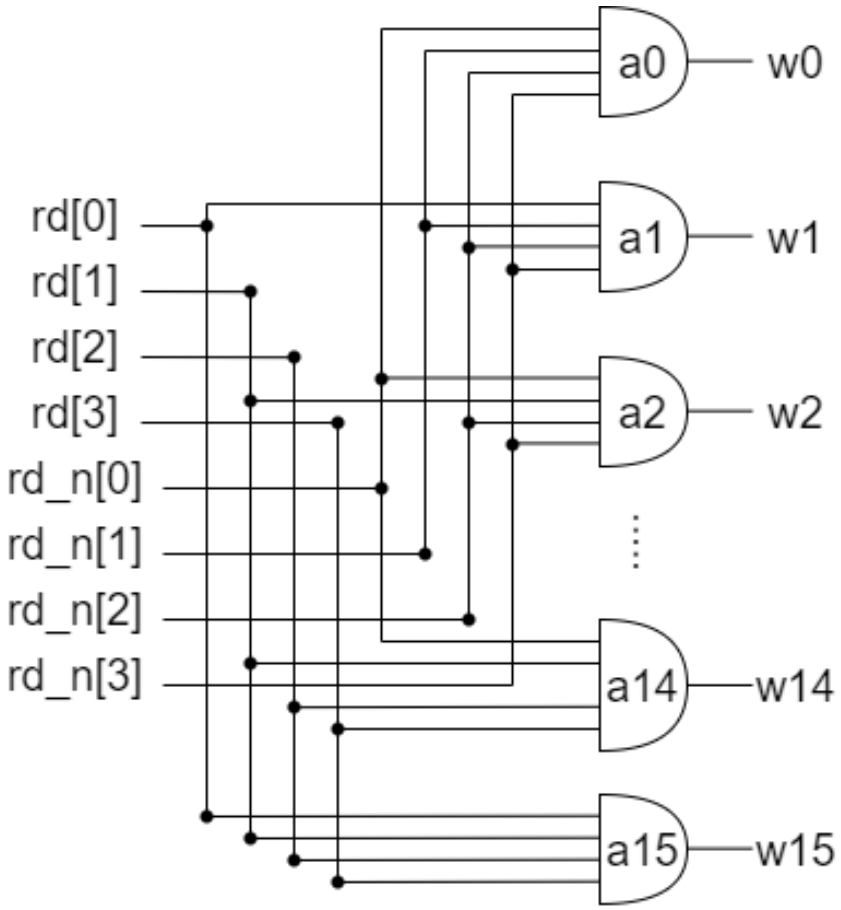
Using the module in question2, but in order to fit the FPGA demonstratio, we make a little bit change.

The output of the module is a 4 bit signal ‘rd’. We first invert all bit of ‘rd’. Then we use 16 ANDs to check what is the number it should show now. Next, for all 7 lightbars, we use OR to check does it should light up. Last, because of the signal should be 0 if it need to light up, we invert the lights signal again.

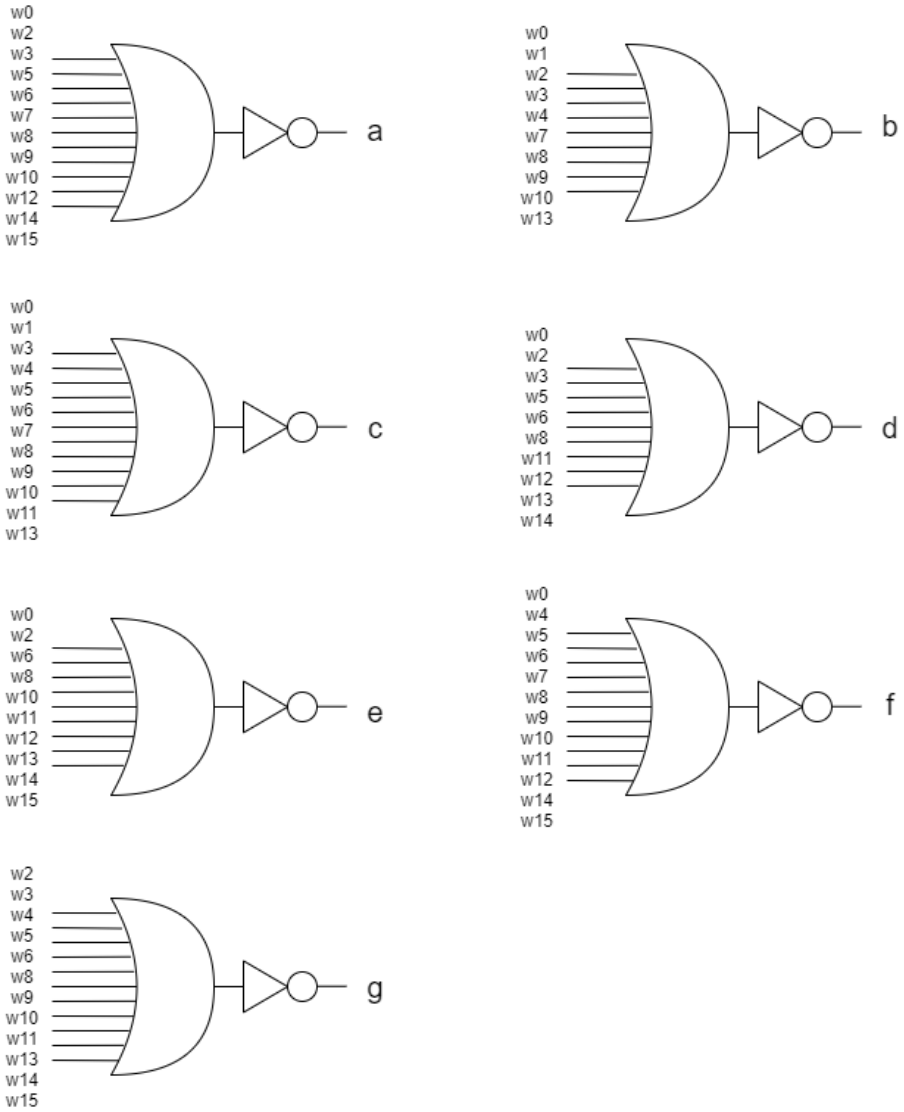
We also use gate levels to make sure the point don’t light up and the number only show up at the rightmost 7-segment display.

(2) Schematic Graph:

- Identify the ‘rd’ signal



■ Light Control



(3) I/O Pin Assignment:

I/O	sel[0]	sel[1]	sel[2]	rs[0]	rs[1]	rs[2]
LOC	V17	V16	W16	W17	W15	V15
I/O	rs[3]	rt[0]	rt[1]	rt[2]	rt[3]	AN[0]
LOC	W14	W13	V2	T3	T2	U2
I/O	AN[1]	AN[2]	AN[3]	a	b	c
LOC	U4	V4	W4	W7	W6	U8
I/O	d	e	f	g	h	
LOC	V8	U5	V5	U7	V7	

Contribution

1. Lawrence Wu

Problem 3 and Problem 4.

2. Ariel Chang

Problem 1, 2, 5 and FPGA demo.

What have we learned?

How to control a 7-segment display.

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.

Write your report before coding.

$display is not equivalent to output wires.

[3:0] means four wires. Such representation is not friendly to C/C++ users.