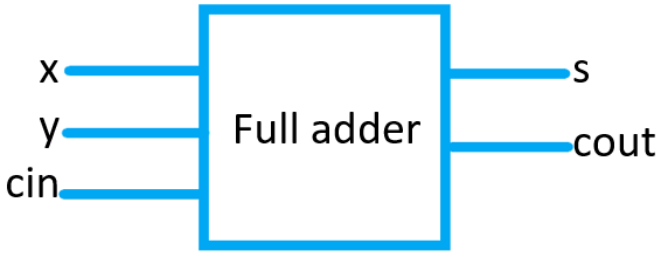
Experiment1

Design Specification

Input: x, y, cin
Output: s, cout

Block diagram:



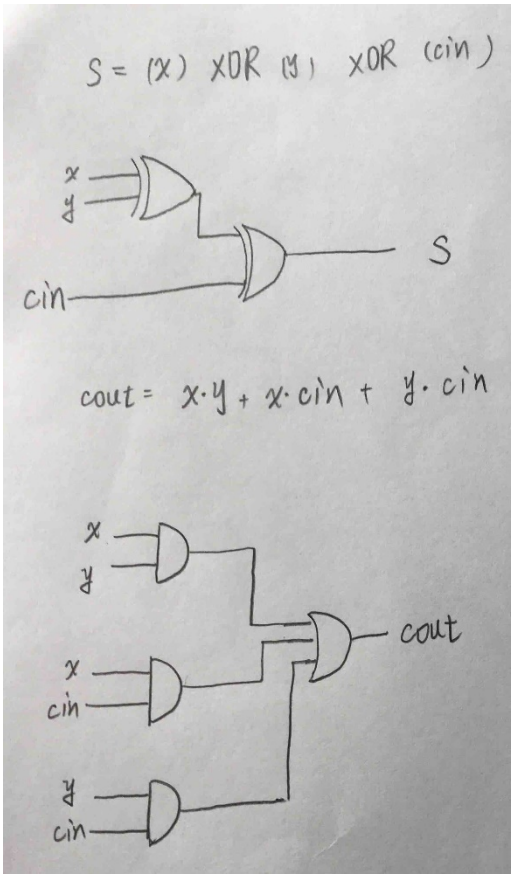
Design Implementation

Logic equation:

因為 s 表示個位數，所以當 3 個 input (x, y, cin)中奇數個 1，s 則會為 1。所以 s = (x) XOR (y) XOR (cin) = s ^ y ^ cin。

cout 表示進位，當 3 個 input (x, y, cin)中有 2 個或 2 個以上為 1，則 cout 為 1。所以 cout = x·y + x·cin + y·cin。

Logic diagram:

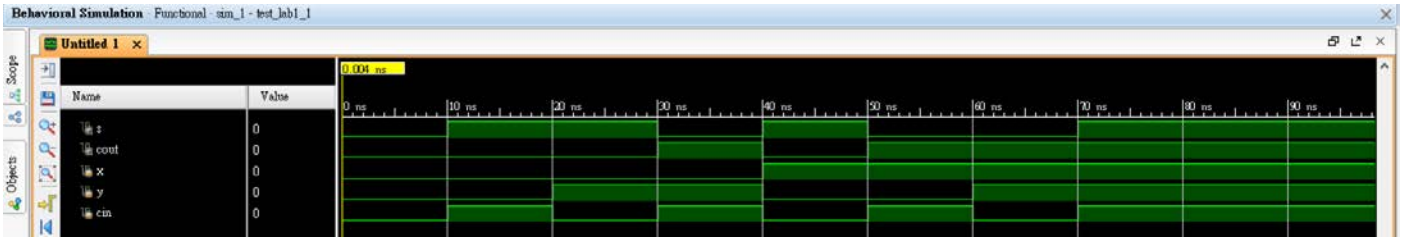


Discussion:

As mentioned above, if there are odd number of 1's in 3 inputs (i.e. one of inputs is 1 or all of inputs are 1) , 's' will be equal to 1. Therefore, I used XOR to check.

As for cout (carry out), at least two of inputs are 1 (i.e. two of inputs is 1 or all of inputs are 1), cout will be equal to 1. Therefore, I used AND gate to check if there are two of inputs equal to 1. Then, I used OR gate to combine the 3 AND gate results.

Simulation waveform:



It's single bit full adder with 3 inputs and 2 outputs. So there are $2^3 = 8$ types of input pattern showed on above waveform and they add each other to produce correspond output. Minimum is 00_2 maximum is 11_2 .

Conclusion:

This experiment made me understand the detail of full adder and review the content of logic design and rules of Verilog as well.

Reference:

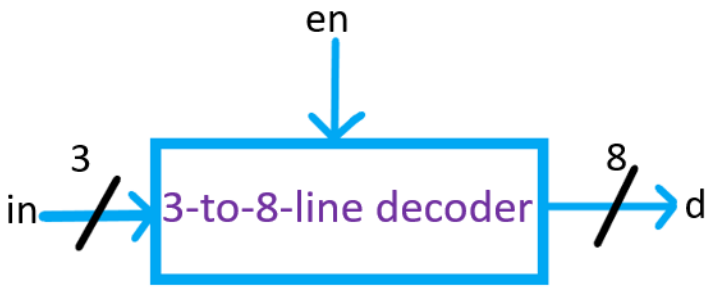
Handout given by the professor (01_introduction.pdf). It introduces some examples of Verilog. I know how to express XOR gate in Verilog and that how to assign output using 'assign'

Experiment 2

Design Specification

Input: in[2:0], en
Output: d[7:0]

Block diagram:



Design Implementation

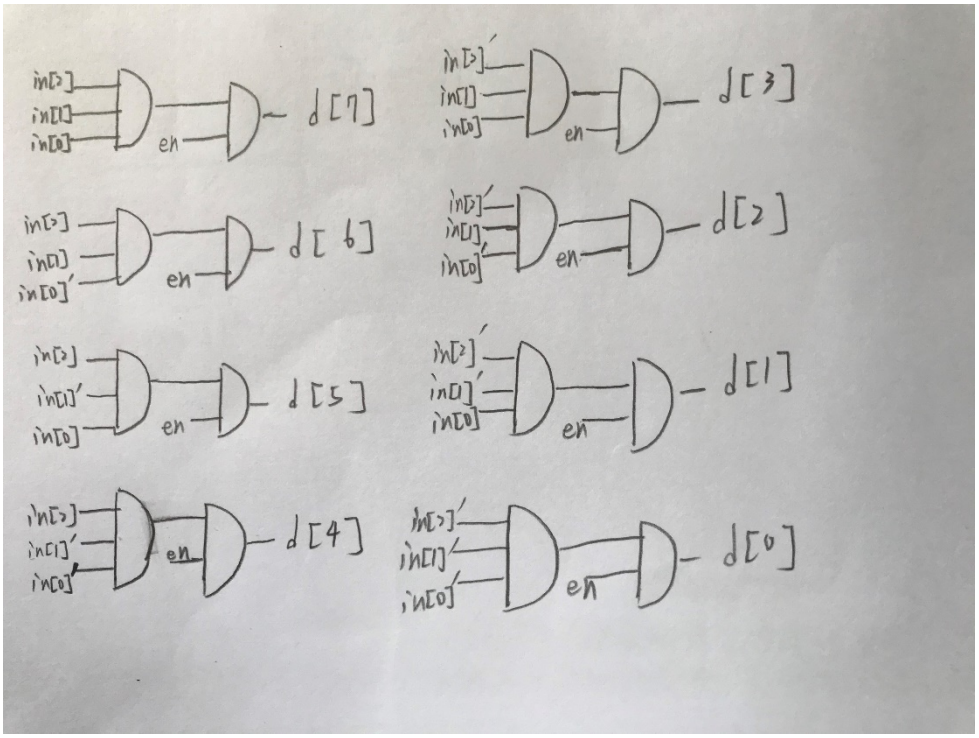
Logic equation:

Truth table:

en	in[2]	in[1]	in[0]	d[7]	d[6]	d[5]	d[4]	d[3]	d[2]	d[1]	d[0]
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0

d[7] = in[2]·in[1]·in[0]·en
d[6] = in[2]·in[1]·in[0]‘·en
d[5] = in[2]·in[1]‘·in[0]·en
d[4] = in[2]·in[1]‘·in[0]‘·en
d[3] = in[2]‘·in[1]·in[0]·en
d[2] = in[2]‘·in[1]·in[0]‘·en
d[1] = in[2]‘·in[1]‘·in[0]·en
d[0] = in[2]‘·in[1]‘·in[0]‘·en

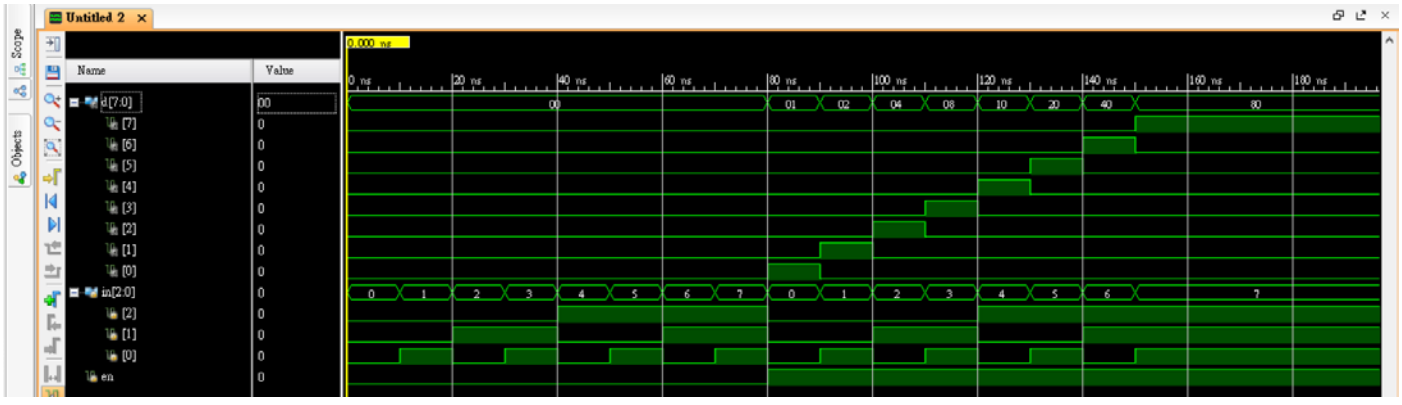
Logic schematic:



Discussion:

First of all, I drew the truth table. I found it’s one-hot relation with the inputs. And only when en (enable) = 1, d (output) is non-zero, otherwise, it’s each bit of d (output) is 0. Therefore, I wrote down the logic equation in Verilog.

Simulation waveform:



The decoder inputs are in[2:0], en. There are 4 bits in total. So, there are 2^4 = 16 types of input pattern showed on the waveform. When en = 0, d = 0. When en = 1, d is corresponded one-hot pattern according to input as showed above. That's because according to logic function, each bit of output is [(3 inputs possible patterns) **AND** en (enable)].

Conclusion:

From this experiment, I know how to handle multi-bit input/output.

Reference:

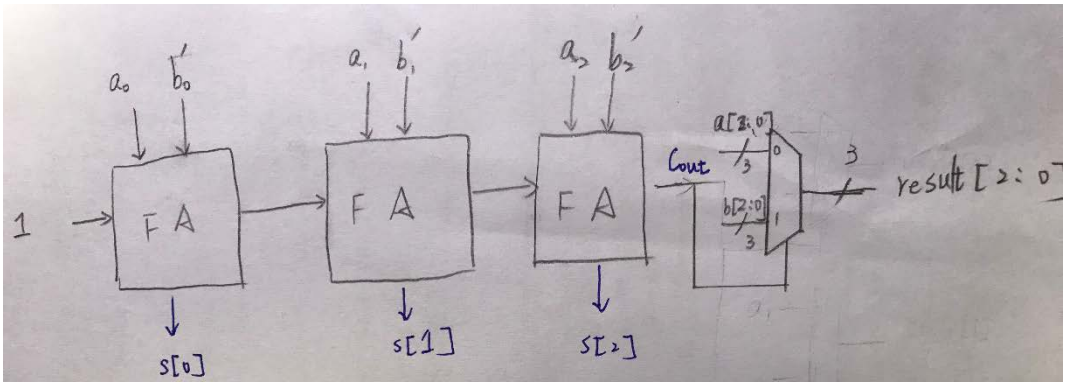
Handout given by the professor (01_introduction.pdf). From it, I know how to access each bit of input/output.

Experiment 3

Design Specification

Input: [2:0]a, b
Output: [2:0]result

IBlock diagram:



Detail of FA (full adder) has been shown at the **Experiment 1**!

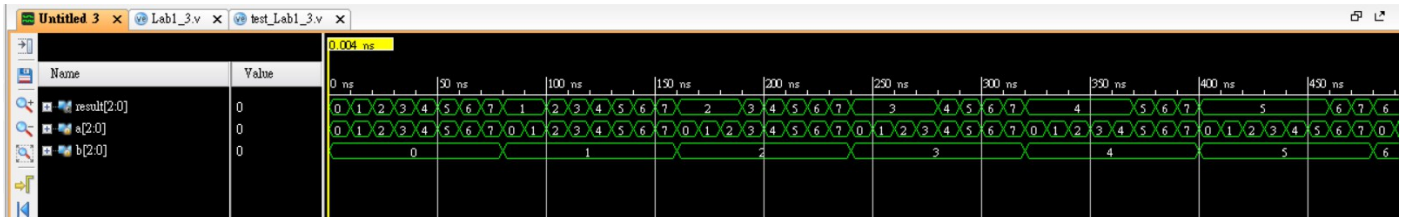
Design Implementation

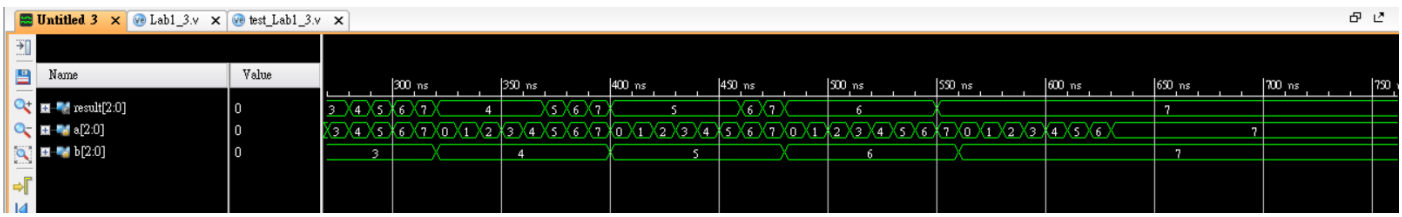
I used 3 full adder to do a[2:0] – b[2:0]. Then using Cout to decide which one is larger. If Cout = 0, a is larger. If Cout = 1, b is larger. So I use Cout to select my final output. (result[2:0])

Discussion:

As I only need to write RTL, I can just if…else… statement to check if a > b or the opposite. If a > b, I set result (output) = a. If a <= b, I set result = b. (when a = b, it doesn't matter which one I output)

Simulation waveform:





The Verilog code is straightforward. But, I think the hardware behind it is more complicated than the code looks like.

Conclusion:

I know the I need to write if…else… statement inside always procedure block. And inside always procedure block, output needs to be declared as ‘reg’

Reference:

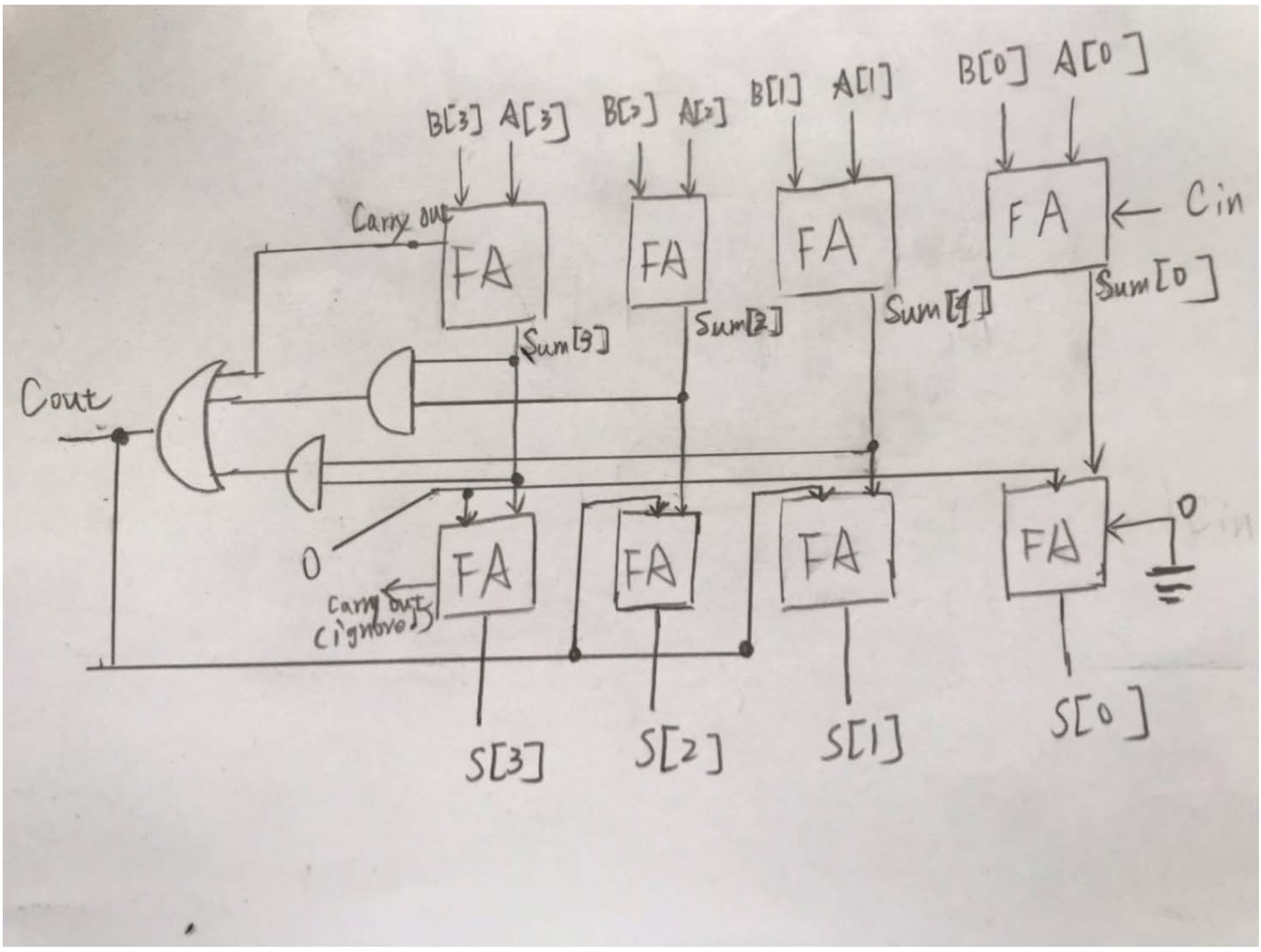
Handout given by the professor (01_introduction.pdf). From it, I know the basic rule of if…else… statement.

Experiment 4 (bonus)

Design Specification:

Input: [3:0]A, B
Input: Cin
Output: [3:0]S
Output: Cout

Block diagram:



Detail of FA (full adder) has been shown at the **Experiment 1!**

Design Implementation

First of all, I check if the A[3:0] + B[3:0] + Cin is less or equal than 9 or not using a comparator. If it is, I just assign its value to S[3:0]. And Cout is obviously equal to 0.

On the other hand, if the result is greater than 9, I need to add **6** to Sum[3:0] to get correct result S[3:0]. And Cout is 1.

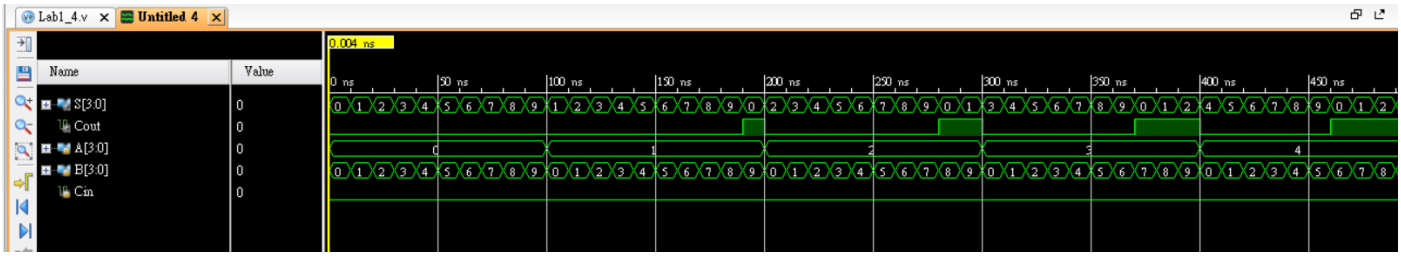
In the block diagram, if carry out is produced from upper 4-bit adder or Sum[3] & Sum[2] are both equal to 1 (i.e. Sum = $11xx_2$ >= 12_{10}) or Sum[3] & Sum[1] are both equal to 1 (i.e. Sum = $1x1x_2$ >= 10_{10}, we need to add six (0110_2) to 'Sum' to produce correct BCD number as showed at the bottom 4-bit adder in block diagram (Cout = 1). Otherwise, I just add 0 to 'Sum' (Cout = 0).

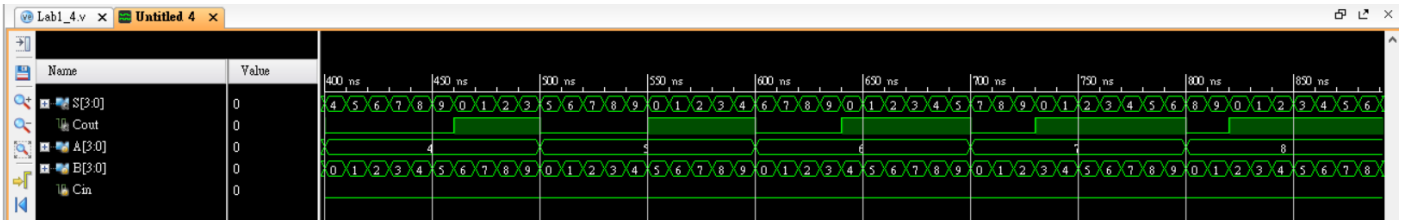
Discussion:

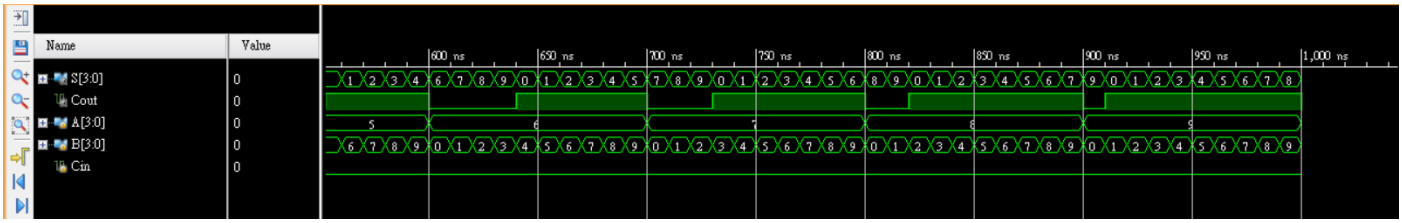
Decimal Adder – The digital systems handles the decimal number in the form of binary coded decimal numbers (BCD). When BCD numbers add together and result is greater than 9. It should produce carry out. (ex: 12 in BCD expression is 0001 0010 instead of 1100 in binary expression). And I need to add **six** to result if the result is greater than 9 to transform it to the correct BCD expression.

So, I need to know all cases whose result will be greater than 9. I google it and found a solution, which is to check [the carry out of the upper 4-bits full adder] or [Sum[3] & Sum[1]] are both equal to 1 or [Sum[3] & Sum[2] are both equal to 1] or not. Then I can correct the upper 4-bits result if it's great than 9.

Simulation waveform:







A, B have 10 possible number (0~9), Cin has 2 possible number (0, 1). There are 10·10·2 = 200 possible input patterns. And I check each corresponded output, it's all correct.

Conclusion:

Through this experiment, I am more familiar with BCD expression. I know how to deal without output when added result is greater than 9. And I learned that in Verilog, I need to use begin…end… to contain multi-line statements in if…else… statement.

Reference: https://www.eeeguide.com/decimal-adder-bcd-adder/

From this website, I know what BCD is, how to express BCD numbers, how to produce the correct result (**plus six**) when doing BCD adder and how to detect if I need add extra **six** to get the correct result.