LAB 2 REPORT

NAND Gate Implementation:

8-bits Ripple Carry Adder, 8-bits Carry Look Ahead Adder, and 4-bits Multiplier

Custom Universal Gate Implementation:

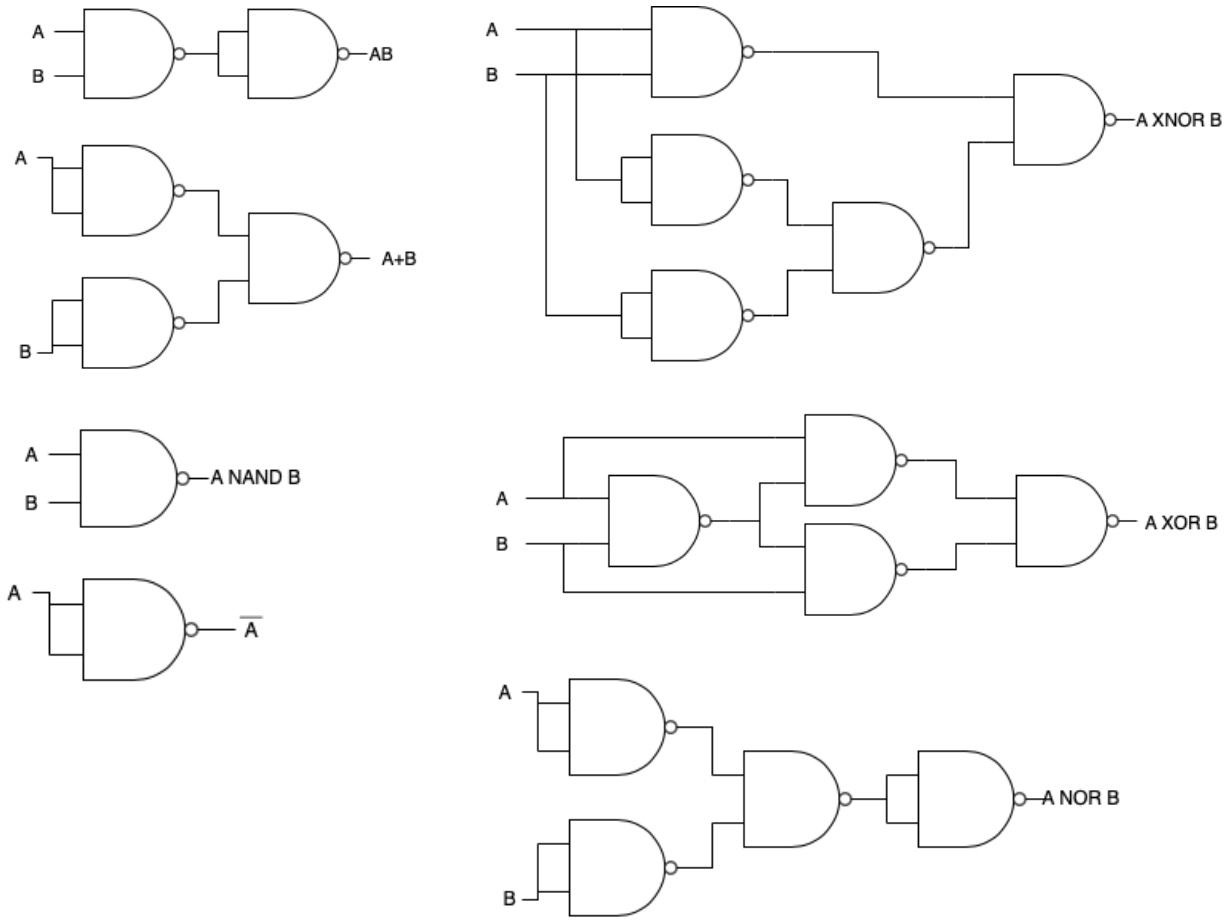
4-bits Decode and Execute (Adder, Subtractor, Bitwise, Shifter, Comparator)

FPGA Implementation on 7-segment Display Output and Toggle Input

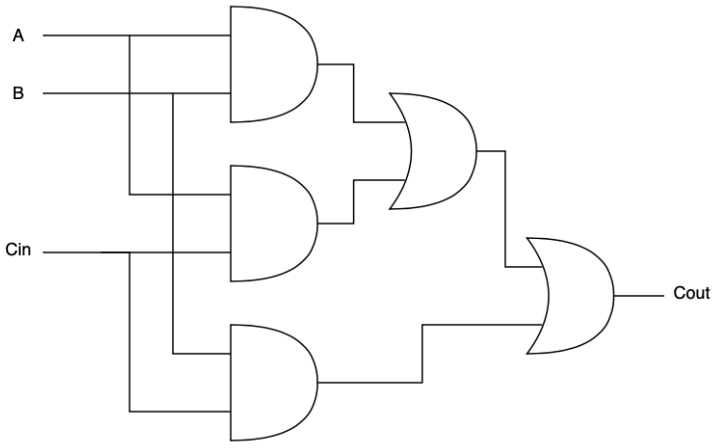
Team 37:

1. 徐美妮 Mary Madeline Nicole 109006205
2. 林之耀 Kevin Richardson Halim 109006277

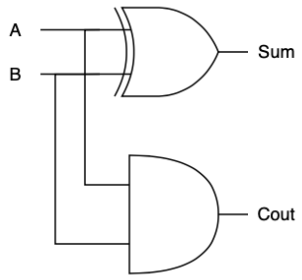
I. Gate-Level Circuits



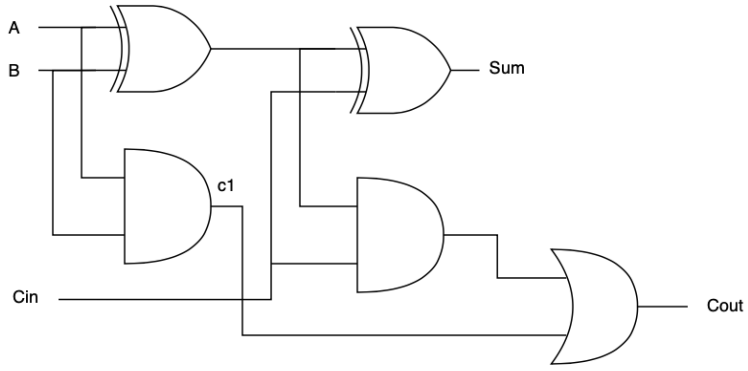
1.1 (Gate Level) NAND gates only



1.2 (Gate Level) 3-input majority gate

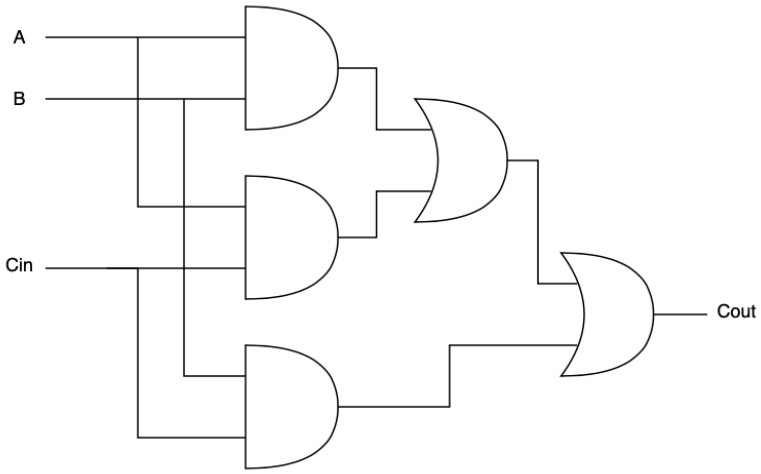


a. Half adder

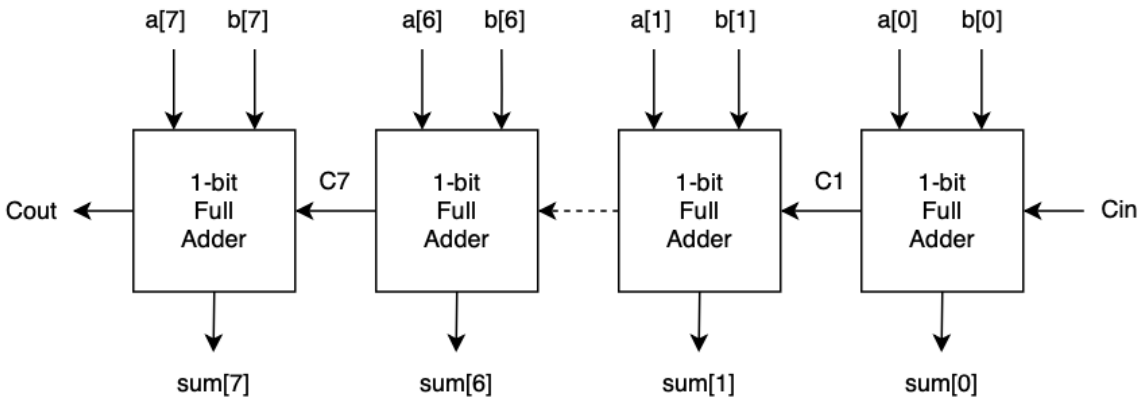


b. Full adder

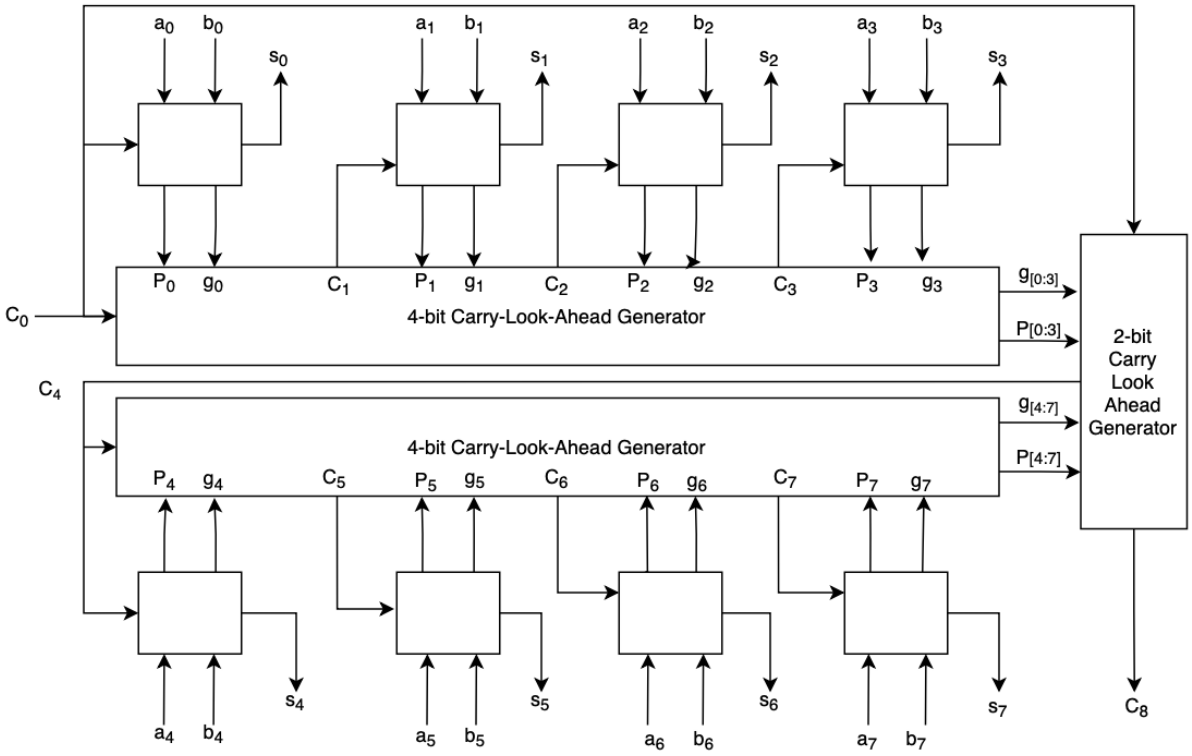
1.3 (Gate Level) 1-bit full adder and half adder



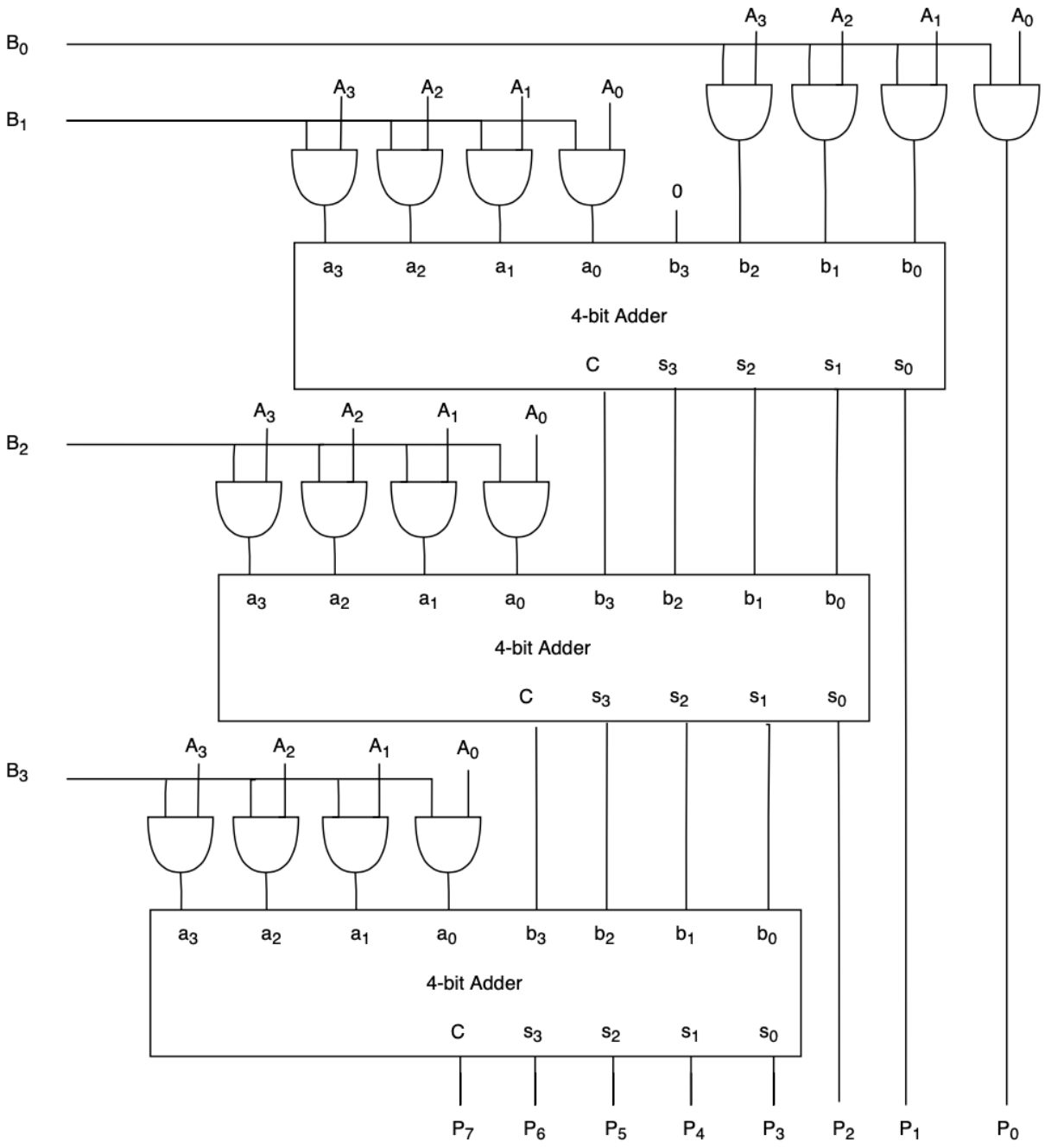
1.4 Majority gate



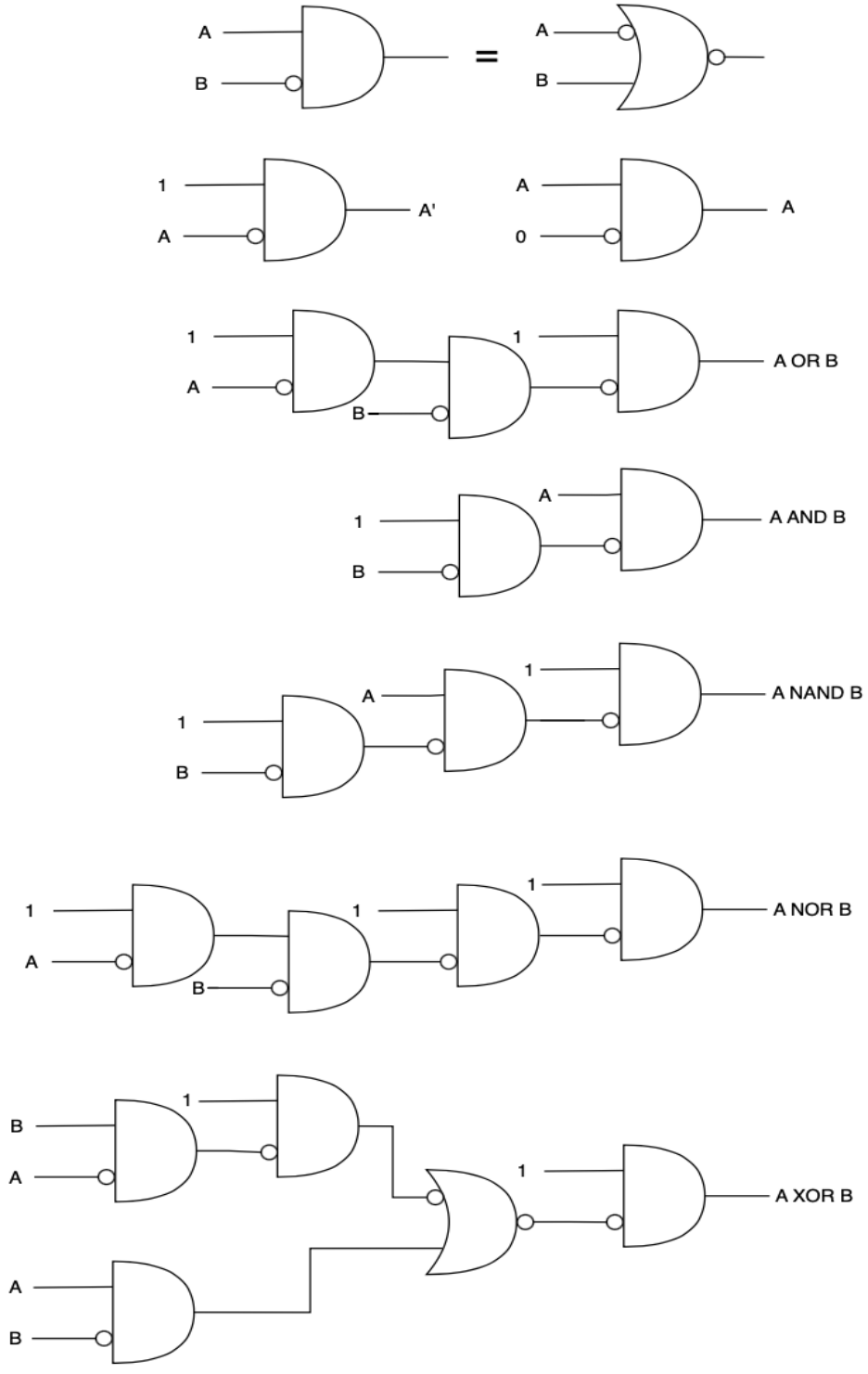
1.5 (Gate Level) 8-bit ripple-carry adder



1.6 (Gate Level) 8-bit carry-lookahead (CLA) adder



1.7 (Gate Level) 4-bit multiplier



1.8 Decode and execute

II. Design Process

The behaviour of each assignment was implemented by using a gate-level description function. All of the modules use NAND gate implementations for its basic gates. First, start by drawing/sketching the design. We slice the codes into smaller modules to create a bigger one hierarchically.

Second, the modules are written in verilog. Each gate and each wire or/and register on the file was implemented to imitate the real gate behavior. Then we check all the spelling, syntax, and errors.

Third, the testbench is built. The testbench is very useful to check whether or not the module works as it should. Moreover, by using the clock it is easier to manage and simulate timing. The use of display and task when making the testbench makes it easier to spot mistakes at our module, by using the CAD server. The testbench that we made covers all possible inputs. Specifying the timescale of the simulation which are the time unit and precision enables it to run fully on vivado. Critical conditions, or edge cases were also made, relying on when the module would break.

Finally, debugging and reiterating each line to fix some bugs that were faced. Such as wrong I/O, swapped variables, wrong naming, wrong behavior, typos, etc.

Modules:

1. (Gate-level) 1-bit full adder & half adder

Half adder has two inputs and two outputs, it is used for 2-bit addition. One XOR gate and one AND gate are used. The output is the sum of two signals. Full adder has three inputs and two outputs, and can be used for multiple bit addition. It is made up of two half adders. A full adder can be utilized as a half adder. To put it simply, a full adder can handle incoming input from one bit carry from one less significant bit adder, which half adder can not do.

2. (Gate-level) 8-bit ripple carry adder (RCA)

A smaller module of a 4-bit ripple carry adder is made to create an 8-bit ripple carry adder. In ripple carry adder, each full adder has to wait for its carry-in from its previous stage full adder. So, the nth full adder (n: number of bits) has to wait until all the previous full adders (n-1) have completed their operations. This

creates a delay which makes ripple carry adder very slow compared to the carry look ahead adder. As the number of bits increases (value of n), the situation will worsen and make it slower and slower.

We formed the ripple carry adder by using full adders that used half adders and majority.

3. Universal Gate

The universal gate is an AND gate with an inverse input, which is similar to nand. Creating a NOT gate by using this universal gate can create a NAND gate, then the other basic gates could be created.

2.4.1 Truth table of universal gate

A	B	B’	out
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Proof of the gates:

- A & B’ = (A & B’) ’ ’ = (A’ & B’ ’) ’ = (A’ + B)
- NOT A = 1 & A’ = A’
- A AND B = A & (B’) ’
- A OR B = (((A’) & B’)) ’
- A NAND B = (A & ((B’) ’)) ’
- A NOR B = (((A’) & B’) ’) ’
- A XOR B = (((B & A’) ’) ’ + (A & B’)) ’

4. (Gate-level) Decode and execute

Utilize universal gate to create basic operation according to operation code (op_code). The requested universal gate is [A & ~(B)] (A as the first input and B as the second input and got negated before both got the reduction by AND gate).

Using De Morgan's law, Universal Gate can be manipulated as AND, OR, and NOT Gate. Thus, the gate could be used as all logical possible gates.

To tackle this problem, we build from ground to up. Start from simple gates such as XOR, Adder, Assign ,etc. Working our way to tackle more complex problems.

5. (Gate-level) 8-bit carry-lookahead (CLA) Adder

The 8 bit CLA circuit was built by creating 2 4-bit carry lookahead generators.

The benefits of carry-lookahead adder is that it generates the carry-in for each adder simultaneously, so the carry output at any stage is dependent only on the initial carry bit of the beginning stage. This reduces the propagation delay.

Carry-lookahead adder is an improved version of the ripple carry adder. It generates the carry-in of each full adder simultaneously without causing any delay. The carry-in of a full adder depends on two parameters, which are the bits being added in the previous stages and the carry-in provided in the beginning. In CLA, both parameters of the full adder are always known from the beginning. So, the carry-in of any stage full adder can be evaluated at any instant of time. Thus, it does not need to wait until its carry-in is generated by its previous state full adder, unlike the RCA.

CLA works by manipulating the boolean expression of the full adder. The propagate (P) and generate (G) in a full adder is:

Pi = Ai XOR Bi (Carry Propagate)

Gi = Ai AND Bi (Carry Generate)

Both propagate and generate only depend on the input bits. Thus the expression for the output sum and carry-out is:

Si = Pi XOR Cin-i

Ci+1= Gi OR PiCi

So, the i-th carry signal will be generated if:

a) Both A(i-1) and B(i-1) are 1
b) Either A(i-1) or B(i-1) is 1 and the C(i-1) is 1

In a 4 bit adder the carry would be,

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

All the respective C2, C3, C4 does not depend on the previous carry in, therefore it does not need to wait for its previous carry to propagate. As soon as C0 is computed, C4 can be generated.

The result of propagate and generate introduces whether each position has a carry in. Each full adder first calculates its own propagate and generate, and the previous bits group carry in is handed to the carry look ahead module to calculate the carry in of the position. The overall structure refers to the circuit, which is divided into 1 bit full adder and 2 4-bits carry look ahead adder, which will be combined with a 2-bits carry look ahead adder to form an 8-bits carry look ahead adder.

On the 1 bit Full adder, we can derive the NAND gate form of the sum, propagate and generate, then continue to implement it. We can then utilize the to create the 4 bit carry look ahead adder. CLA adder merges the information of each full adder quickly and independently calculates the carry of each bit, which significantly speeds up the entire adder.

6. (Gate-level) 4 bit multiplier

Multiplying two 4-bit binary numbers involves the multiplication and the addition of digits with or without carry. After multiplication of each bit to the multiplicand, partial products are generated. Then these products are added to produce the total sum which represents the binary multiplication value. The multiplication is implemented by a combinational circuit such that the multiplication is performed with AND gates where the addition is carried out by using full adders.

The first partial product is obtained by the AND gate which gets the least significant bit of the multiplication result. Since the second partial product is shifted to the left position, the first partial second term and second partial product is added by the adder and produce the sum output along with the carry out. The carry out is then added at the next half adder as the input. It will produce the multiplication result of the binary numbers by using the circuit configuration. Multiplication of two 4-bit numbers results in a 8-bit binary number.

Each partial product consists of four product terms and these are shifted to the left relative to the previous partial product. These partial products are then added by using a 4-bit parallel adder. Then the result is added to the next partial product with carry out and it goes on until the final partial product and produces 8 bit sum, which is the multiplication value of the two binary numbers.

So, in binary multiplication actually use AND gate, and use adder to add and carry the value of each bit to get the answer. Multiplying a and b by AND gate, and because both a and b are 4 bits, so after multiplying 4 bits, use RCA to multiply the result. Add up and count one by one to get the answer.

The value 0 is added to every adder that does not require carry in.

7. (Gate-level) An exhausted testbench design

The exhausted testbench was created for a 4-bit adder circuit consisting of ripple carry adder. There are two additional pins, error and done. First, initialize every input into 0. Then start to test every possible case with a repeat block of value (2 ** 9). Every 1 nano second, we test whether or not it is correct, and every 4 nano second, we update the input. After all the combinations are done, set the delay into 5 nano second and set done into 1. Whenever there is error during testing, the error would turn into 1, else it would turn back into 0.

8. (FPGA) Decode and execute

Program the FPGA by providing the module file and constraint as an XDC file. Configuring the I/O port according to the display light's port. As what we want to make is a single hexadecimal at the rightmost 7-segment display,

therefore we make the AN[3:0] into 1110, so only the rightmost lights up because AN is used to enable one of the four digits. When it is set into LOW, it illuminates a digit. . To figure out how the LED will form hexadecimal numbers. There are Eight signals for the segments and dot. Which are segments A~G. We set its value to low according to each shape of the hexadecimal character that we want.

Then after configuring the XDC file according to the I/O of the switches and segment display, connect the FPGA to the computer and program it.

III. List of Contributions

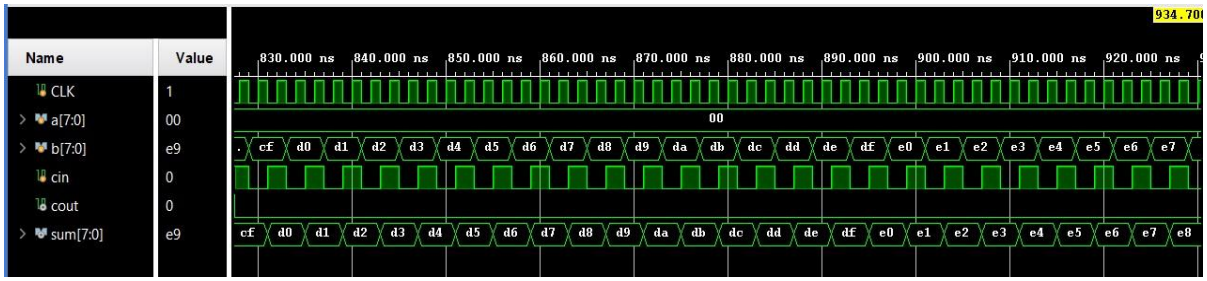
徐美妮 Mary Madeline Nicole 109006205's contributions, such as :

- Design and write module Ripple_Carry_Adder,
- Design and write Ripple_Carry_Adder testbench,
- Design and write Decode_And_Execute_4bit testbench,
- Design and write module Carry_Look_Ahead_Adder_8bit,
- Design and write Carry_Look_Ahead_Adder_8bit testbench,
- Design and write module Multiplier_4bit,
- Design and write Multiplier_4bit testbench,
- Draw gate level circuit,
- Program the FPGA board,
- Design and write XDC file,
- Write the report.

林之耀 Kevin Richardson Halim 109006277's contributions, such as :

- Design and write testbench Exhausted_Testing,
- Design and write module Decode_And_Execute_4bit,
- Design and write module Carry_Look_Ahead_Adder_8bit,
- Design and write Carry_Look_Ahead_Adder_8bit testbench,
- Design and write module Multiplier_4bit,
- Design and write Multiplier_4bit testbench,
- Design and write Ripple_Carry_Adder testbench,
- Simulate and synthesis modules in Vivado,
- Program the FPGA board,
- Design and write XDC file,
- Write the report.

IV. Design Test



4.1 (Gate-level) 8-bit ripple carry adder (RCA)

Clock signal was simulated to give input at the negative edge, and to test at the positive edge. We repeat this 2^9 times because there are 9 bit in total for the input, which is 4 bits of a (input 1), 4 bits of b (input 2) and 1 bit of Carry in. In the repeat block, {a,b,cin} is updated every negative edge by 1 binary bit, so that the combination would go as follows:

a = 0, b = 0, cin = 0

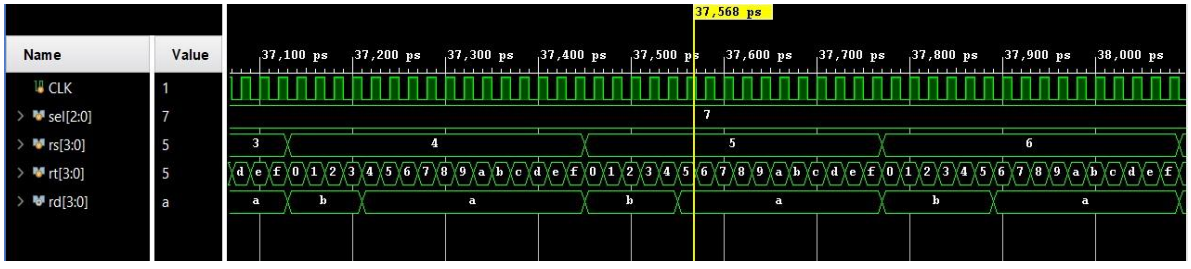
a = 0, b = 0, cin = 1

a = 0, b = 1, cin = 0

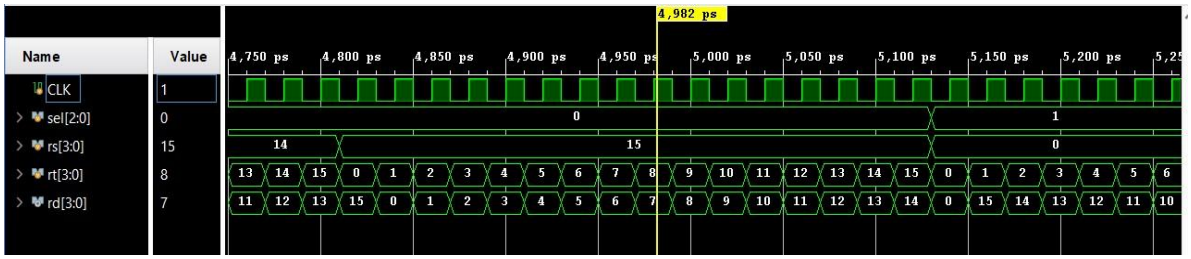
a = 0, b = 1, cin = 1

And it will go on until every combination from 2 to the power of 9 is completed. Utility task is used for testing and debugging in the CAD server to make sure that all of the output is right, the waveform is also observed to double check the answer.

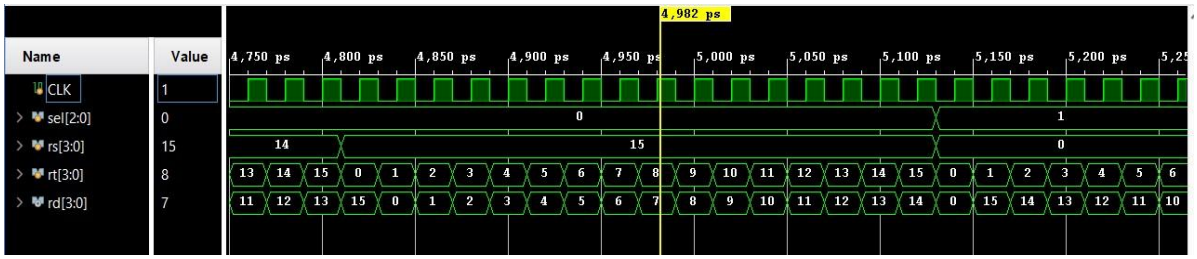
The value of the output received was right, thus the simulation results proved that the program was successfully created.



(a) Comparator



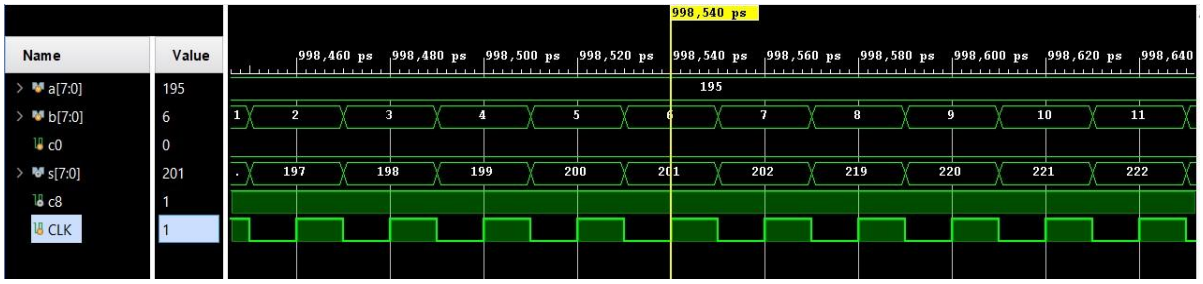
(b) Addition



(c) Bitwise operation

4.2 (Gate-Level) Decode and execute

In the testbench, a clock signal was created to simulate and manage timing. It will be tested at every positive edge, and the input will change at every negative edge. We test all possible combination by using repeat block (2 ** 3) to update the select and (2 ** 8) to update the inputs {rs,rt} = {rs, rt} + 8'b1 is used to update, so rs and rt will be updated by 1 bit on each input to check all combinations. Task test was created to debug and display the error of any operation. Case is used to check the outputs according to the operation select (sel). Moreover, the code for decode and execute successfully work when implemented to the FPGA board, therefore the program is successful.

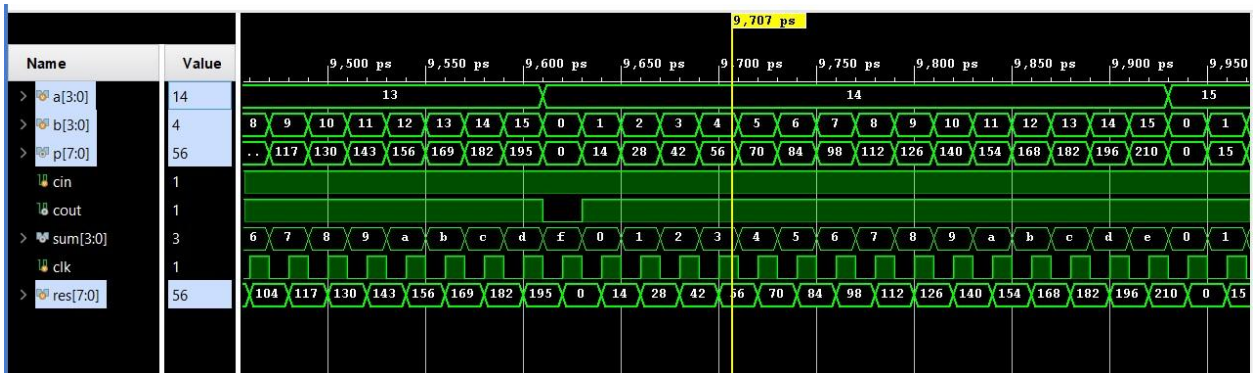


4.3 (Gate-level) 8-bit carry-lookahead adder (CLA)

In the testbench, a clock signal was used to simulate and manage timing. At every positive edge, the input will be tested and it will be updated at every negative edge. There

are 2^17 total combinations that can be tested to cover all possible inputs. At the test task, we check whether or not the output cout and sum is right. To ensure that the overflow is not overlooked, we did the following:

```
task Test;
begin
  if (s !== (a + b + c0) & 32'b0000000011111111) begin
    $display("[ERROR] sum (s)");
    $write("a: %d\t", a);
    $write("b: %d\t", b);
    $write("c0: %d\n", c0);
    $write("s: %d\t", s);
    $write("should be: %b\n", {(a + b + c0) & 16'b0000000011111111});
    $display;
  end
  if (c8 !== &((a + b + c0) & 32'b000000100000000)) begin
    $display("[ERROR] cout (c8)");
    $write("a: %d\t", a);
    $write("b: %d\t", b);
    $write("c0: %d\n", c0);
    $write("c8: %d\t", c8);
    $write("should be: %b\n",&((a + b + c0) & 16'b000000100000000));
    $display;
  end
end
endtask
```



4.4 (Gate-level) 4-bit multiplier

In the input, a total of 2^4 * 2^4 = 2^ 8 = 256 combinations all can be tested within time to ensure the correctness of the code. Task test was created to check and debug the error of any calculation. After running it through CAD server and vivado there is no wrong answer, therefore the program is successful.

V. What we have learned from Lab 2

- ➔ Utilizing the NAND gate to create all other basic gates
- ➔ Creating modules in gate level only
- ➔ Understanding the application of timescale ...ns/...ps
- ➔ Designing testbench that can also be used for debugging
- ➔ Implementing Decode Encode in FPGA