Kevin Karnadi Kirmansjah 紀維鑫

109006241

Logic Design

Lab 2: ALU and Carry-Lookahead Adder

I. Design of the 4-Bit Carry Lookahead Adder

```
module CLA_4bit(A, B, Cin, S, Cout);

    parameter n = 4;
    input [n - 1: 0] A, B;
    input Cin;

    output [n - 1: 0] S;
    output Cout;

    wire [3:0] G,P,C;

    assign G = A & B;

    assign P = A ^ B;

    assign C[0] = Cin;
    assign C[1] = G[0] | (P[0] & C[0]);
    assign C[2] = G[1] | (P[1] & C[1]);
    assign C[3] = G[2] | (P[2] & C[2]);

    assign Cout = G[3] | (P[3] & C[3]);
    assign S = A ^ B ^ C;

endmodule
```

This 4-bit Carry Lookahead Adder uses A and B, which are 4-bit binary numbers, and Cin as the inputs, and it will produce S, which is also a 4-bit binary number, and Cout as the output. In order to compute S and Cout, we also need the to find the Generate, Propagate, and Carry values for each bit:

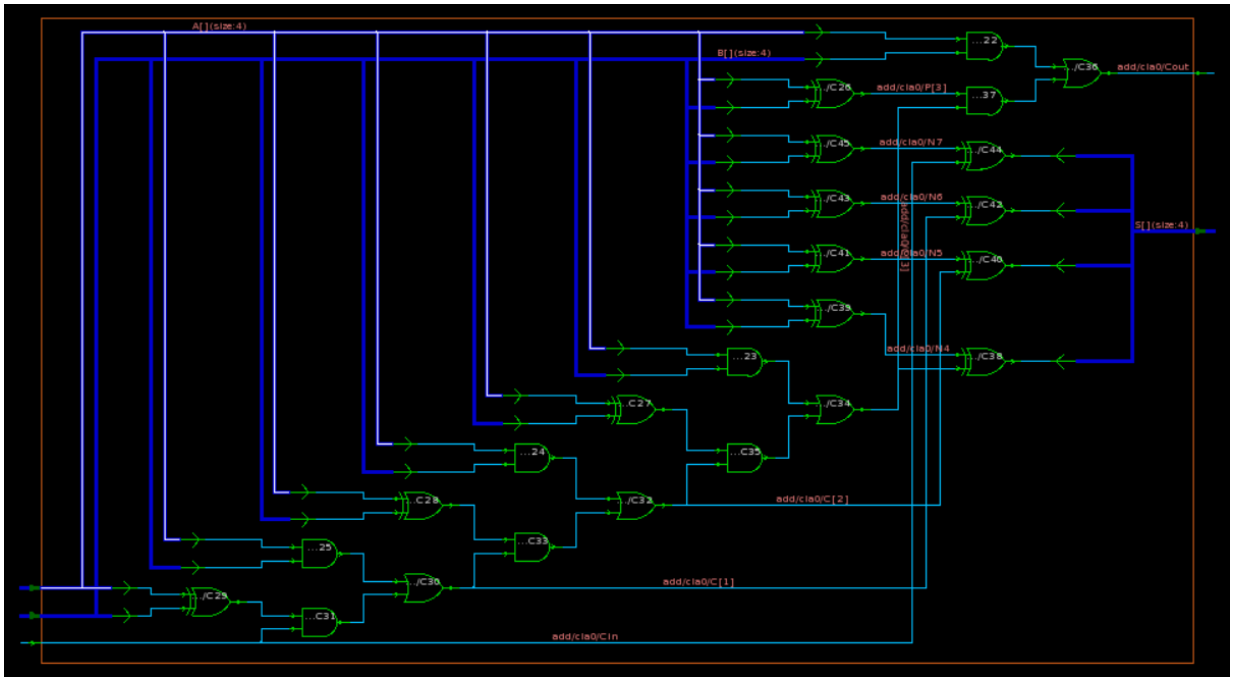
The Generate function is defined as G[i] = A[i] & B[i], or can be written as G = A & B.

The Propagate function is defined as P[i] = A[i] ^ B[i], or can be written as P = A ^ B.

The Carry bit is defined as C[i] = G[i-1] | (P[i-1] & C[i-1]), but for C[0] the value is just Cin.

Finally, the sum of each bit can be defined as S[i] = A[i] ^ B[i] ^ C[i], or can be written as S = A ^ B ^ C, whereas the carry output can be defined as Cout = G[3] | (P[3] & C[3]).

This is what the design of the CLA, obtained by using Design Vision, looks like:



II. Construction of the 16-Bit Adder with 4-Bit CLAs

```
module Adder_16bit(A, B, Cin, S, Cout);

    parameter n = 16;
    parameter m = 4;

    input [n - 1: 0] A, B;
    input Cin;

    output [n - 1: 0] S;
    output Cout;

    wire C4, C8, C12;
    wire [m - 1: 0] S0_3, S4_7, S8_11, S12_15;
    assign S = {S12_15, S8_11, S4_7, S0_3};

    CLA_4bit cla0(A[3:0], B[3:0], Cin, S0_3, C4);
    CLA_4bit cla1(A[7:4], B[7:4], C4, S4_7, C8);
    CLA_4bit cla2(A[11:8], B[11:8], C8, S8_11, C12);
    CLA_4bit cla3(A[15:12], B[15:12], C12, S12_15, Cout);

endmodule
```

This 16-bit Adder uses A, B, which are 16-bit binary numbers, and Cin as the inputs, and it will produce S, also a 16-bit binary number, along with Cout as the outputs.

Here, with the help of the “CLA_4bit” module, the 16-bit Adder is divided into four 4-bit CLAs, and it will produce S0_3, S4_7, S8_11, S12_15, along with C4, C8, C12, and Cout.

For the first group, we use A[0] – A[3], B[0] – B[3], and Cin as the inputs, and it will produce S0_3 and C4 as the outputs.

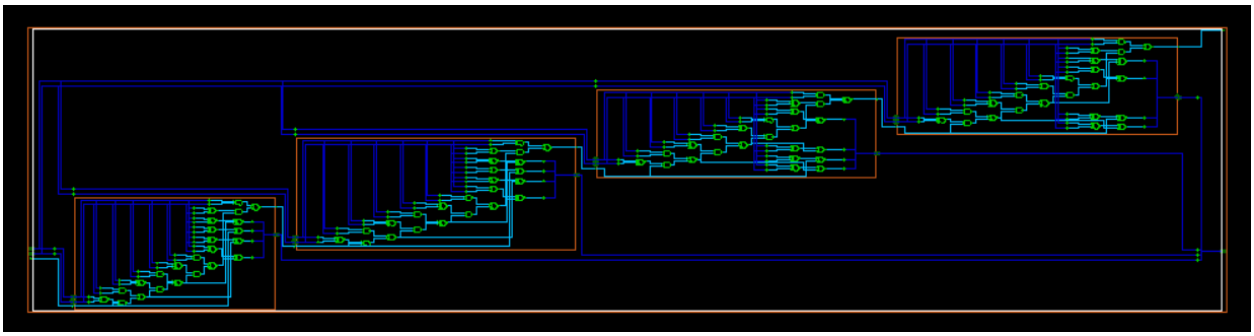
For the second group, we use A[4] – A[7], B[4] – B[7], and C4 as the inputs, and it will produce S4_7 and C8 as the outputs.

For the third group, we use A[8] – A[11], B[8] – B[11], and C8 as the inputs, and it will produce S8_11 and C12 as the outputs.

For the last group, we use A[12] – A[15], B[12] – B[15], and C12 as the inputs, and it will produce S12_15 and Cout as the outputs.

Lastly, we are going to assign S as the concatenation of S12_15, S8_11, S4_7, and S0_3, which will be equivalent to a 16-bit binary number.

This is what the design of the 16-bit Adder, obtained by using Design Vision, looks like:



III. The Design and Implementation of my ALU

This ALU uses A, B, which are 16-bit binary numbers, along with Cin, and Mode as the inputs. Mode is a number that will determine the operation that is going to be used. The ALU will then produce Y, also a 16-bit binary number, along with Cout, and Overflow as the outputs.

The Operations:

1. Logical shift A left by 1-bit

```
4'd0: begin
Y = A << 1'b1;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will move each bit to the left by one. For example, if the input is 11001011, the output will be 10010110.
To achieve this, we use the operator <<, and write Y = A << 1'b1, which means Y is A shifted to the left by 1 bit. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

2. Arithmetic shift A left by 1-bit

```
4'd1: begin
Y = A <<< 1'b1;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation is identical to Logical Shift Left.
To achieve this, we use the operator <<<, and write Y = A <<< 1'b1, which means Y is A shifted to the left by 1 bit. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

3. Logical shift A right by 1-bit

```
4'd2: begin
Y = A >> 1'b1;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will move each bit to the right by one. For example, if the input is 11001011, the output will be 01100101.

To achieve this, we use the operator >>, and write Y = A >> 1'b1, which means Y is A shifted to the right by 1 bit. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

4. Arithmetic shift A right by 1-bit

```
4'd3: begin
Y = A >>> 1'b1;
Y[15] = A[15];
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation is identical to Logical Shift Right, but the empty MSB is filled with the value of the previous MSB. For example, if the input is 11001011, the output will be 11100101.
To achieve this, we use the operator >>>, and write Y = A >>> 1'b1, which means Y is A shifted to the right by 1 bit. After that, we write Y[15] = A[15] to assign the value of the output's MSB as the previous MSB. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

5. Add two numbers with CLA

```
4'd4: begin
Y = Y_Add;
Cout = Cout_Add;
Overflow = (~A[15] & ~B[15] & Y[15]) | (A[15] & B[15] & ~Y[15]);
end
```

In this operation, we use some new wires that was created outside of the always block, called Y_Add and Cout_Add.

```
wire [n - 1:0] Y_Add;
wire [n - 1:0] Y_Sub;
wire Cout_Add;
wire Cout_Sub;
Adder_16bit add(A, B, Cin, Y_Add, Cout_Add);
Adder_16bit sub(A, (~B)+1'b1, Cin, Y_Sub, Cout_Sub);
```

The values of Y_Add and Cout_Add can be generated with the help of the "Adder_16bit" module. To obtain them, we need to call the 16-bit Adder with A,

B, and Cin as the inputs. Then, we will assign the outputs of that module to Y_Add and Cout_Add.

After that, I created a truth table and a K-map from it to find the Boolean expression for the overflow.

A[15]	B[15]	Y[15]	Overflow
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

BY / A	00	01	11	10
0	0	1	0	0
1	0	0	0	1

Overflow = (!A[15] & !B[15] & Y[15]) | (A[15] & B[15] & !Y[15])

6. Subtract B from A

```
4'd5: begin
Y = Y_Sub;
Cout = Cout_Sub;
Overflow = (~A[15] & B[15] & Y[15]) | (A[15] & ~B[15] & ~Y[15]);
end
```

In this operation, we also use some new wires that was created outside of the always block, called Y_Sub and Cout_Sub.

```
wire [n - 1:0] Y_Add;
wire [n - 1:0] Y_Sub;
wire Cout_Add;
wire Cout_Sub;
Adder_16bit add(A, B, Cin, Y_Add, Cout_Add);
Adder_16bit sub(A, (~B)+1'b1, Cin, Y_Sub, Cout_Sub);
```

The values of Y_Sub and Cout_ Sub can be generated with the help of the "Adder_16bit" module. To obtain them, we need to call the 16-bit Adder with A, (~B)+1'b1 (which is the 2's complement for B), and Cin as the inputs. Then, we will assign the outputs of that module to Y_Sub and Cout_Sub.

After that, I created a truth table and a K-map from it to find the Boolean expression for the overflow.

A[15]	B[15]	Y[15]	Overflow
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

BY / A	00	01	11	10
0	0	0	1	0
1	1	0	0	0

Overflow = (!A[15] & B[15] & Y[15]) | (A[15] & !B[15] & !Y[15])

7. And

```
4'd6: begin
Y = A & B;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will perform a Bitwise AND operation between A and B.
To achieve this, we use the operator &, and write Y = A & B, which means Y is the result of performing a Bitwise AND operation between A and B. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

8. Or

```
4'd7: begin
Y = A | B;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will perform a Bitwise OR operation between A and B.
To achieve this, we use the operator |, and write Y = A | B, which means Y is the result of performing a Bitwise OR operation between A and B. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

9. Not A

```
4'd8: begin
Y = ~A;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will perform a Bitwise NOT operation on A.
To achieve this, we use the operator ~, and write Y = ~A, which means Y is the result of performing a Bitwise NOT operation on A. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

10. XOR

```
4'd9: begin
Y = A ^ B;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will perform a Bitwise XOR operation between A and B.
To achieve this, we use the operator ^, and write Y = A ^ B, which means Y is the result of performing a Bitwise XOR operation between A and B. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

11. XNOR

```
4'd10: begin
Y = ~(A ^ B);
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will perform a Bitwise XNOR operation between A and B.
To achieve this, we use the operator ^ and ~, and write Y = ~(A ^ B), which means Y is the result of performing a Bitwise XOR operation between A and B, and then performing a Bitwise NOT operation on the result. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

12. NOR

```
4'd11: begin
Y = ~(A | B);
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will perform a Bitwise NOR operation between A and B.
To achieve this, we use the operator | and ~, and write Y = ~(A | B), which means Y is the result of performing a Bitwise OR operation between A and B, and then performing a Bitwise NOT operation on the result. There won't be any Cout or Overflow in this operation, so we just assign it as 0.

13. Binary to One-Hot

```
4'd12: begin
Y = 16'b1 << A[3:0];
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation will convert a binary number into its' one-hot representation.
To achieve this, we can write Y = 16'b1 << A[3:0], which means that the binary number 1 is going to be shifted into the left by A bits. For example, if A is 0011 (which is equal to 3 in decimal form), the output Y will be 0000000000001000. There won't be any Cout or Overflow in this operation, so we can just assign it as 0.

14. A

```
4'd13: begin
Y = A;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation basically just copies A to the output.
In order to do that, we can write Y = A, which means the value of A is assigned to Y. There won't be any Cout or Overflow in this operation, so we can just assign it as 0.

15. B

```
4'd14: begin
Y = B;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation basically just copies B to the output.
In order to do that, we can write Y = B, which means the value of B is assigned to Y. There won't be any Cout or Overflow in this operation, so we can just assign it as 0.

16. Find first one from left

```
4'd15: begin
if(A[15] == 1)
    Y = 4'd15;
else if(A[14] == 1)
    Y = 4'd14;
else if(A[13] == 1)
    Y = 4'd13;
else if(A[12] == 1)
    Y = 4'd12;
else if(A[11] == 1)
    Y = 4'd11;
else if(A[10] == 1)
    Y = 4'd10;
else if(A[9] == 1)
    Y = 4'd9;
else if(A[8] == 1)
    Y = 4'd8;
else if(A[7] == 1)
    Y = 4'd7;
else if(A[6] == 1)
    Y = 4'd6;
else if(A[5] == 1)
    Y = 4'd5;
else if(A[4] == 1)
    Y = 4'd4;
else if(A[3] == 1)
    Y = 4'd3;
else if(A[2] == 1)
    Y = 4'd2;
else if(A[1] == 1)
    Y = 4'd1;
else if(A[0] == 1)
    Y = 4'd0;
Cout = 1'b0;
Overflow = 1'b0;
end
```

This operation is going to find where the first ‘1’ is located on a binary number.

In order to do that, we can write an if-else statement, to check whether if the number 1 is located on that bit.

For instance,

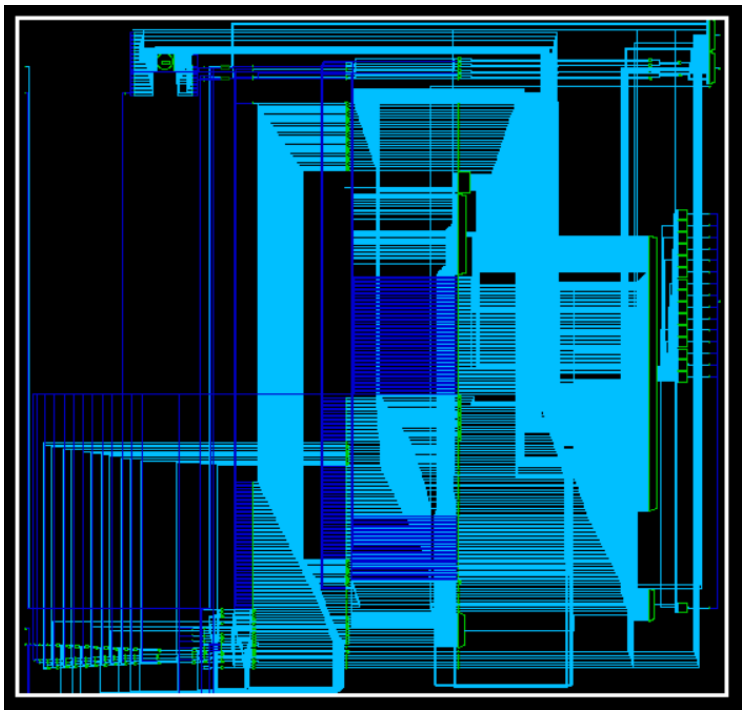
```
if(A[15] == 1)
        Y = 4'd15;
else if( … )
```

means that if the 15th bit of that binary number is the number '1', than the output, or the value of Y, will be 15. Otherwise, check for the next bit (else if …).

For example, if the input is 0011111111111100, we will go through the first if statement first, and check whether if the 15th bit is 1 or not. If not, continue to the next else if statement, until the number '1' is found. In this case, the if statement will stop at the third if statement (else if(A[13] == 1)), because at A[13], the number '1' is found. The statement below the if statement will then be executed, thus making the value of Y into 13.

There also won't be any Cout or Overflow in this operation, so we can just assign it as 0.

This is what the design of the ALU, obtained by using Design Vision, looks like:



IV. Problems I faced, and how I dealt with it

One of the biggest problems that I've faced in this project is when I'm creating the Adder and Subtractor operations. At first, I have no clue about how to assign the Y, Cout, and Overflow values. But after looking at the PPT and doing some research on the internet for some time, I finally found a way to assign the values for Y, Cout, and X.

Another problem that I've faced is about how to create the binary to one-hot representation operation. At first, I didn't know how to do such an operation, but after asking my friends for some help, I finally found a way to solve it, which is simply by moving the number 1 to the left by A bits.

V. Questions for TAs

For the 16^{th} operation (Find first one from left), is there any shorter and more efficient method to do it, besides using multiple if-else statements?