Lab2 ALU and Carry-Lookahead Adder

4-bit Carry-Lookahead Adder 設計流程

Carry-Lookahead Adder(CLA) 是一種快速將兩個 n-bit binary 相加的加法器，之所以能夠快速計算是因為不像 Ripple Carry Adder 需要等待前面計算出來的 Carry bit 才能計算現在的 Sum bit，實現的方法是將 Carry bit 預先計算出來。什麼時候 Carry bit 為 1 呢?假如 A, B 分別表示要相加的兩個數字，C, S 分別表示 Carry 以及 Sum，那麼令

$$G_i = A_i B_i$$
$$P_i = A_i \oplus B_i$$

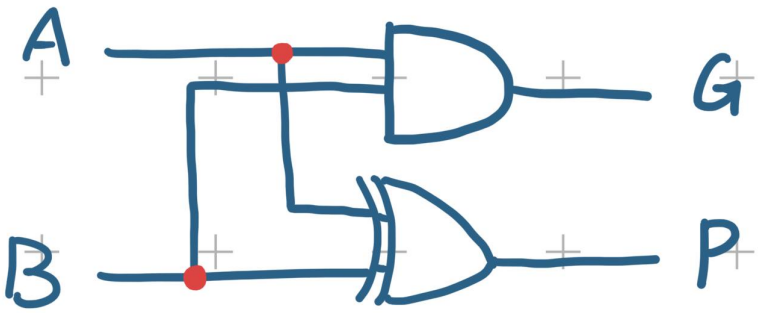
則 C_i 可以表示成：

$$C_i = \begin{cases} 1 & \texttt{if } G_{i-1} + (P_{i-1} C_{i-1}) \texttt{ is true} \\ 0 & \texttt{else} \end{cases}$$

現在要設計 4-bit 的 CLA，所以需要計算 C_1~C_4，在這次的 Lab 當中 $C_0 = Cin$，而輸出的 $Cout$ 則為 C_4 。因此可以得到：

$$C_0 = Cin$$
$$C_1 = G_0 + P_0 C_0 = G_0 + P_0 Cin$$
$$C_2 = G_1 + P_1 C_1 = G_1 + (P_1 G_0 + P_0 P_1 Cin)$$
$$C_3 = G_2 + P_2 C_2 = G_2 + (P_2 G_1 + P_1 P_2 G_0 + P_0 P_1 P_2 Cin)$$
$$C_4 = G_3 + P_3 C_3 = G_3 + (P_3 G_2 + P_2 P_3 G_1 + P_1 P_2 P_3 G_0 + P_0 P_1 P_2 P_3 Cin)$$

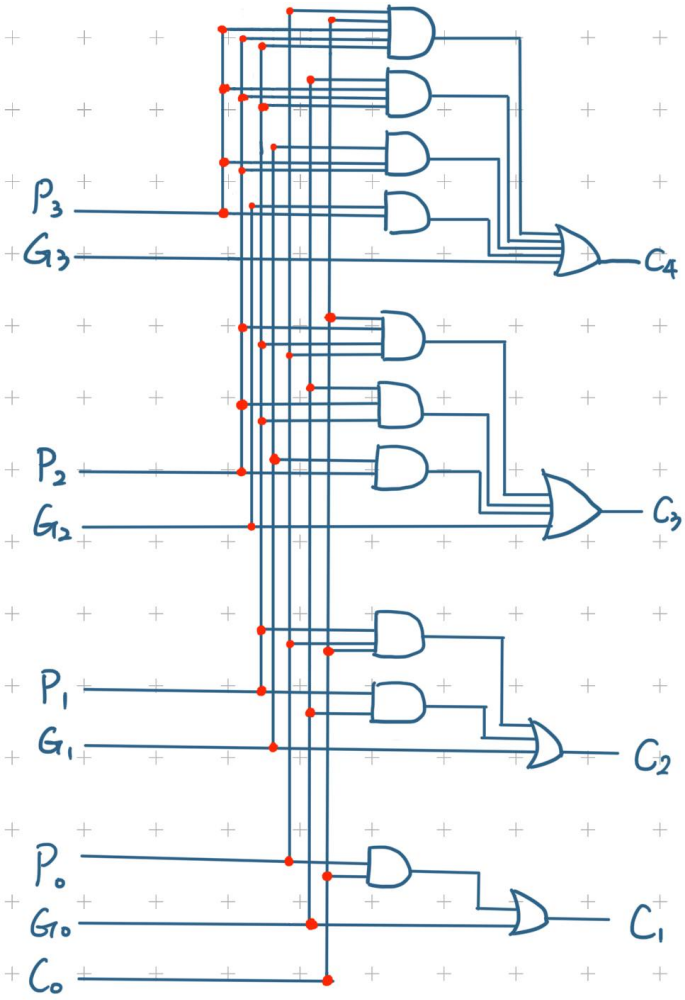
有了這幾條式子，就可以同時將 C_1~C_4 計算出來，有了 Carry bit 就可以和 A, B 計算出 Sum bit。這裡我將每一項功能都拉出來成一個 module 撰寫。首先是 G, P，實際上過去做過的 Half Adder 構造就跟這個一樣，當時的 Carry bit 就是這裡的 G_i Sum bit 就是這裡的 P_i，因此首先先構造 Half Adder。



Code 的部分設定輸入為 A, B，輸出為 P, G，接下來直接 assign P, G 的結果即可。

```
module HalfAdder(A, B, P, G);
        input A, B;
        output P, G;

        assign P = A ^ B;
        assign G = A & B;
endmodule
```

接下來將產生 Carry bit 的部分也構造成一個 module 稱為 Carry-Lookahead Generator(CLG)，依照上述等式建構出電路圖。



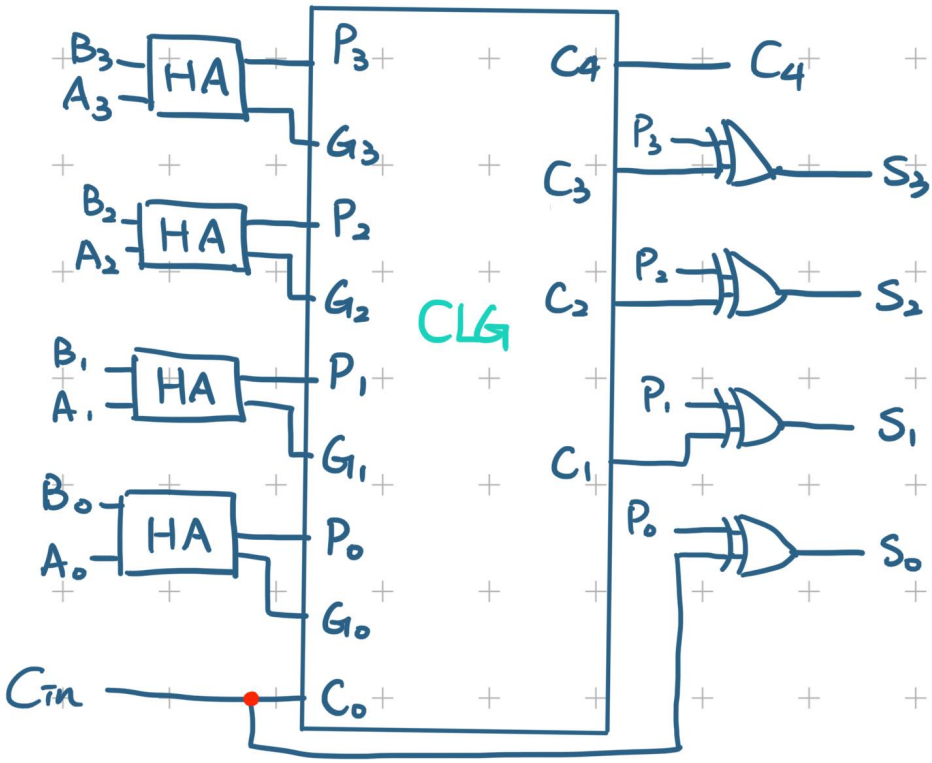
Code 的部分將 Cin, P, G 設為輸入，$Cout$ 設為輸出，接著一一 assign 每個 $Cout$ 的值。這裡的 $Cout[3:0]$ 等同於上述等式當中的 C_1~C_4。

```
module CLG(Cin, P, G, Cout);
    input Cin;
    input [3:0] G, P;
    output [3:0] Cout;

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

有了 CLG 以及 Half Adder，將兩個組合後加上接下來要處理 Sum bit。Sum bit 實際上等同於在做 A_i, B_i, C_i 的加法運算，由於 Half Adder 已經計算出 P_i，接下來只需要看 P_i, C_i 的加法運算，因此可得

$$C_i = \begin{cases} 1 & \texttt{if } P_i \oplus C_i \texttt{ is true} \\ 0 & \texttt{else} \end{cases}$$

以圖像呈現如下：



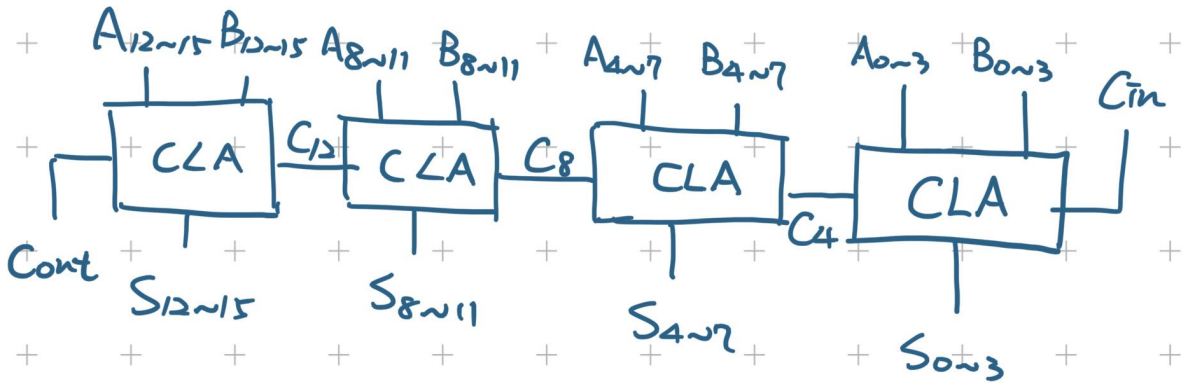
Code 的部分將 A, B, Cin 設為輸入，$S, Cout$ 設為輸出。接下來透過 Half Adder 以及 CLG 將 Carry bit 以及 P, G 計算出來，最後將 Sum bit 賦值。

> 附註：在前面我們使用的 C_1~C_4 在實作上寫成 C_0~C_3，而前面使用的 C_0 實作上則寫成 Cin。

```
module CLA_4bit(A, B, Cin, S, Cout);
        input [3:0] A, B;
        input Cin;

        output [3:0] S;
        output Cout;

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

        assign S[0] = Cin ^ P[0];
        assign S[1] = C[0] ^ P[1];
        assign S[2] = C[1] ^ P[2];
        assign S[3] = C[2] ^ P[3];
        assign Cout = C[3];

        HalfAdder HA0(A[0], B[0], P[0], G[0]);
        HalfAdder HA1(A[1], B[1], P[1], G[1]);
        HalfAdder HA2(A[2], B[2], P[2], G[2]);
        HalfAdder HA3(A[3], B[3], P[3], G[3]);

        CLG CLG0(Cin, P, G, C);
endmodule
```

16-bit Adder 設計流程

有了 4-bit CLA 之後，接下來要就可以組成 16-bit Adder。想法上很簡單，16-bit 的加法可以拆解成 4 個 4-bit 的加法，不過每個 CLA 計算出來的 $Cout$ 會是下一個 CLA 的 Cin，最後一個 $Cout$ 會是最後的 $Cout$。



Code 的部分首先設定 A, B, Cin 為輸入， $S, Cout$ 為輸出。接下來透過 4 個 CLA 計算 $S, Cout$，而每個 CLA 之間透過 wire C 連接在一起，如此一來就可以將 16-bit Adder 實作出來。

```
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 [2:0] C;

        CLA_4bit CLA0(A[3:0], B[3:0], Cin, S[3:0], C[0]);
        CLA_4bit CLA1(A[7:4], B[7:4], C[0], S[7:4], C[1]);
        CLA_4bit CLA2(A[11:8], B[11:8], C[1], S[11:8], C[2]);
        CLA_4bit CLA3(A[15:12], B[15:12], C[2], S[15:12], Cout);
endmodule
```

ALU 設計流程

接下來要設計 ALU，在這次 Lab 當中 ALU 有 16 種功能，接下來一一介紹每個功能以及設計流程。

Mode 0:Logical Left Shift and Mode2:Logical Right Shift

Logical shift 是單純地將所有的 bit 向指定的方向移動一位，並且空 bit 補 0，超出範圍的 bit 則忽略。舉例來說，將 $1010_{(2)}$ 做 Left Shift 會變成 $0100_{(2)}$，做 Right Shift 會變成 $0101_{(2)}$。在 verilog 當中可以直接使用 << 表示 Left Shift，>> 表示 Right Shift。因此在 ALU 當中直接將 Y 賦值即可。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            // Logical Shift A left by 1-bit
            4'd0: begin
               Y = A << 1;
            end
            // Logical Shift A right by 1-bit
            4'd2: begin
                Y = A >> 1;
            end
            // ignore other cases
        endcase
    end
endmodule
```

Mode1:Arithmetic Left Shift and Mode3:Arithmetic Right Shift

Arithmetic Left Shift 與 Logical Left Shift 相同，而 Arithmetic Right Shift 則不太相同。Arithmetic Right Shift 會將所有的 bit 向後移動，但是最高位的 bit 會和原本相同。舉例來說，將 $1010_{(2)}$ 做 Arithmetic Right Shift 會變成 $1101_{(2)}$。在 verilog 當中可以直接使用 <<< 表示 Left Shift，>>> 表示 Right Shift，不過因為我們的數字是有號的 (Signed)，所以需要將 A, B 的宣告改成 Signed，否則計算出來的結果會有誤。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    input signed [n - 1: 0] A, B;
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            //Arithmetic shift A left by 1-bit.
            4'd1: begin
                Y = A <<< 1;
            end
            //Arithmetic shift A right by 1-bit.
            4'd3: begin
                Y = A >>> 1;
            end
            // ignore other cases
        endcase
    end
endmodule
```

Mode4:Sum of A and B with Cin

前面我們已經實作好 16-bit Adder，接下來只需要將功能接上，再來處理 Overflow 即可。由於在 always block 當中不能直接使用其他 module，這裡多宣告兩個 wire $AtmpY, AtmpCout$ 來連接到 Adder，接下來只需要在 ALU 當中將這兩條線接到 Adder，在 always block 當中分別將 $AtmpY, AtmpCout$ 的結果交給 $Y, Cout$ 即可。

Overflow 則需要考慮到 A, B 可能為負數的情況。什麼時候會 Overflow 呢?唯有在 A, B 同號時才有機會，因為如果 A, B 不同號但是 Overflow，這表示其中一者本來就已經 Overflow 了，因此接下來只需要考慮 A, B 同號的情況。在同號的情況下如果相加後的結果與 A, B 不同號就表示出現 Overflow。也就是說， Overflow 的條件為：

$$Overflow = \begin{cases} 1 & \texttt{if } A_{n-1}B_{n-1}S'_{n-1} + A'_{n-1}B'_{n-1}S_{n-1} \texttt{ is true} \\ 0 & \texttt{else} \end{cases}$$

在 Code 的部分透過 $AtmpY, AtmpCout$ 連接到 Adder_16bit，接著將兩者的值分別賦予給 $Y, Cout$，而 Overflow 則如同上述將邏輯直接寫下。記得要將 Cin 也連接進去。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    wire [n-1:0] AtmpY;
    wire AtmpCout;
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            //Add two numbers with cla.
            4'd4: begin
                Y = AtmpY;
                Cout = AtmpCout;
                Overflow = (A[n-1]&B[n-1]&(~AtmpY[n-1])) | ((~A[n-1])&(~B[n-1])&AtmpY[n-1]);
            end
            // ignore other cases
        endcase
    end
    Adder_16bit A0(A, B, Cin, AtmpY, AtmpCout);
endmodule
```

Mode5:Subtract B from A

設計的概念與 Mode4 相同，不過這次不需要跟 Cin 做相加減，因此 Cin 的部分直接傳入 0。作法仍然是使用 16-bit Adder，將 $A - B$ 看作是 $A + (-B)$，這裡的 $-B$ 可以直接寫成 $-B$ 或是寫成 $(\sim B) + 1$ 也可以。此外，在計算 Overflow 的部分會有些不同，可以看作是 Mode4，但是 B 的 sign bit 反轉過後的情況。可以注意 Code 當中將 $B[n-1]$ 改為 $\sim B[n-1]$，$\sim B[n-1]$ 改為 $B[n-1]$。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    wire [n-1:0] BtmpY;
    wire BtmpCout;
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            //Subtract B from A.
            4'd5: begin
                Y = BtmpY;
                Cout = BtmpCout;
                Overflow = (A[n-1]&(~B[n-1])&(~AtmpY[n-1])) | ((~A[n-1])&(B[n-1])&AtmpY[n-1]);
            end
            // ignore other cases
        endcase
    end
    Adder_16bit A0(A, B, Cin, AtmpY, AtmpCout);
```

```
endmodule
```

Mode6~11: Some bitwise operator

在 Mode6~11 分別要實作 $A \wedge B$, $A \vee B$, $\sim A$, $A \oplus B$, $\sim (A \oplus B)$, $\sim (A \vee B)$。這些 operation 在 verilog 當中都已經有實作了，因此單純地將每個結果給 Y 即可。

> 有趣的是，在 verilog 當中 nxor 可以直接使用 `~^` 表示。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            //and
            4'd6: begin
                Y = A&B;
            end
            //or
            4'd7: begin
                Y = A|B;
            end
            //not A
            4'd8: begin
                Y = ~A;
            end
            //xor
            4'd9: begin
                Y = A^B;
            end
            //xnor
            4'd10: begin
                Y = ~(A^B);
            end
            //nor
            4'd11: begin
                Y = ~(A|B);
            end
            // ignore other cases
        endcase
    end
endmodule
```

Mode12:Decoder

Decoder 也就是要將 Binary 轉換成 One-hot encoding，在這次的 Lab 當中我們要取 A 的最後 4 個 bit，轉換成 10 進位後的數字表示在結果 Y 的該 bit 為 1，其餘為 0。舉例來說，A 的最後 4 個 bit 為 $0100_{(2)}$，由於轉換成 10 進位後結果為 $4_{(10)}$，因此輸出結果為 $0000\ 0000\ 0001\ 0000_{(2)}$。

那麼該怎麼實作呢？直接將 A 的最後 4 個 bit 取出來，假設這個稱為 $A[3:0]$，接下來將 1 做 Left Shift $A[3:0]$ 次就會是我們要的結果。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            //binary to one-hot
            4'd12: begin
                Y = 1<<A[3:0];
            end
            // ignore other cases
        endcase
    end
endmodule
```

Mode13:Comparater

Comparater 要在 $A \leq B$ 的時候回傳 0，否則回傳 1。數字的大小判斷在 verilog 當中可以直接使用 <= 判斷，選擇要回傳哪一個數值可以單純用 if-else 處理。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            //Comparator
            4'd13: begin
                if(A >= B)
                    begin
                        Y = 16'd0;
                    end
                else
                    begin
                        Y = 16'd1;
                    end
            end
            // ignore other cases
        endcase
    end
endmodule
```

Mode14:Y=B

這個 Mode 要直接將 Y 的值設為 B。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            //B
            4'd14: begin
                Y = B;
            end
            // ignore other cases
        endcase
    end
endmodule
```

Mode15:Find MSB of A

這個 Mode 要找到 A 的 MostSignificantBit，回傳的結果前 12 個 bit 皆為 0，後四個 bit 表示 MSB 在第幾位，以 2 進位表示。舉例來說，假如 $A = 0000\ 0000\ 1100\ 1000_{(2)}$，由於 MSB 在第 7 個 bit，$7_{(10)} = 0111_{(2)}$，因此回傳結果為 $0000\ 0000\ 0000\ 0111_{(2)}$。我沒有想到比較好的寫法，因此在這裡直接用 if-else 枚舉出 16 種情況，從第 15 到第 0 個 bit 檢查看看是不是 1。

```
module ALU(A, B, Cin, Mode, Y, Cout, Overflow);
    // ignore some codes here, just jump to always block
    always@(*)begin
        case(Mode)
            //find first one from left
            4'd15: begin
                if((A&(1<<15)) != 0) Y = 15'd15;
                else if((A&(1<<14)) != 0) Y = 15'd14;
                else if((A&(1<<13)) != 0) Y = 15'd13;
                else if((A&(1<<12)) != 0) Y = 15'd12;
                else if((A&(1<<11)) != 0) Y = 15'd11;
                else if((A&(1<<10)) != 0) Y = 15'd10;
                else if((A&(1<<9)) != 0)  Y = 15'd9;
                else if((A&(1<<8)) != 0)  Y = 15'd8;
                else if((A&(1<<7)) != 0)  Y = 15'd7;
                else if((A&(1<<6)) != 0)  Y = 15'd6;
                else if((A&(1<<5)) != 0)  Y = 15'd5;
                else if((A&(1<<4)) != 0)  Y = 15'd4;
                else if((A&(1<<3)) != 0)  Y = 15'd3;
                else if((A&(1<<2)) != 0)  Y = 15'd2;
                else if((A&(1<<1)) != 0)  Y = 15'd1;
                else Y = 15'd0;
            end
            // ignore other cases
        endcase
    end
endmodule
```

遇到的問題以及解決方法

問題一：CLG 參數過多

第一次進行設計時在 CLG 設計的參數完全是拆開來如下：

```
module CLG(C0, G0, P0, G1, P1, G2, P2, G3, P3, C1, C2, C3, C4);
    input C0, G0, P0, G1, P1, G2, P2, G3, P3;
    output C1, C2, C3, C4;

    assign C1 = G0 | (P0&C0);
    assign C2 = G1 | (P1&G0 | P0&P1&C0);
    assign C3 = G2 | (P2&G1 | P1&P2&G0 | P0&P1&P2&C0);
    assign C4 = G3 | (P3&G2 | P2&P3&G1 | P1&P2&P3&G0 | P0&P1&P2&P3&C0);
endmodule
```

這個版本雖然內容比現在的版本來的少，但是在輸入參數需要花很多的心力確認傳進去的參數是否正確無誤，設計 CLA 呼叫 CLG 會變得很麻煩，後來想到可以將 $P, G, C_1 \sim C_4$ 都用向量表示法，如此一來就可以減少許多的輸入參數。

問題二：Adder_16bit CLA 串接問題

雖然知道將 4 個 4-bit CLA 串接起來可以變成 16-bit Adder，但是因為每個 CLA 都需要等待前一個 CLA 傳遞過來的 Carry bit(Cin) 才能夠開始運行，要怎麼串接起來是個問題，必須要有個 $Cout$ 跟 CLA 接起來。後來想到可以用 wire 串接，因為有 4 個 CLA，因此建立 3 條 wire 串接起來，接著就能順利運作了！

問題三：Adder_16bit 使用 CLA 問題

同樣是在 16-bit Adder 當中，因為要將 A, B 都拆成 4 份傳入 CLA 當中，當時不太知道該怎麼傳一部分的 Bit 進去。想到講義當中有提到大括號(`{}`) 可以將許多的 Bit 包起來，所以一開始嘗試傳進去的參數大概如下：

```
CLA_4bit CLA0({A[3], A[2], A[1], A[0]}, {B[3], B[2], B[1], B[0]}, Cin, {S[3], S[2], S[1], S[0]},
C[0]);
```

但是這樣顯然很麻煩，想著應該存在更好的方法，照著 python 以及過去寫 verilog 的思維，嘗試使用 $A[3:0]$ 來取代 $\{A[3], A[2], A[1], A[0]\}$，發現是可行的，接下來把全部都換成這個樣子，看起來的確好很多。

問題四：ALU Arithmetic Shift

在 ALU 當中要設計 Arithmetic Shift，一開始忘記了 Arithmetic Shift 和 Logical Shift 的差別，後來發現到 Arithmetic Shift 的 Left Shift 和 Logical Shift 相同，但是 Right Shift 的最高 bit 會維持相同，並沒有想法應該怎麼設計比較好，但是看到 HackMD 當中寫到 `>>>` 以及 `<<<`，讓我懷疑實際上 verilog 當中已經有實作過 Arithmetic Shift，在網路上查到一篇文章，當中提到實際上 verilog 當中確實有 `>>>` 以及 `<<<` 的存在，不過需要注意到 ***signed*** 以及 ***unsigned*** 的問題。透過 `make sim` 可以確認在沒有加上 `signed` 的情況下確實會有問題，反之加上後就完全沒問題了。

問題五：ALU 使用 Adder & Substracter

雖然已經設計好了 Adder ，也知道 Substracter 只是把 B 改成 $-B$ 傳入 Adder 當中就可以完成，不過具體要怎麼把 Adder 放進 ALU 當中是個大問題。當初嘗試這樣做：

```
always@(*)begin
    case(Mode)
        // ignore some cases
        4'd4: begin
            Adder_16bit A0(A, B, Cin, Y, Cout);
        end
        // ignore some cases
    endcase
end
```

不過仔細想想會發現問題很多。首先是 Overflow 要怎麼計算，如果直接經過 Adder，那 Y 以及 $Cout$ 都會直接傳進 Output 當中，那就沒辦法計算 Overflow 了。再來這個結構本身就很奇怪，在某個 case 底下才會把 Adder 跟 ALU 連接在一起，然後輸出 $Y, Cout$，這就像是兩塊電路板之間的接線會根據不同 case 自己決定要不要接在一起，怎麼想都很怪。

所以不如一開始就接著，不過傳出來的值不一定都會用上，也就是說一開始就透過 wire 連接，只是在特定的 case 底下才決定要不要取得 wire 的值，拿去計算或是輸出。

問題六：ALU Substracter 計算 Overflow

使用 Adder 計算 $A + (-B)$ 會記得把 B 加上一個負號，不過在計算 Overflow 的時候忘記了應該要加上，導致好幾次測試的時候測資都不會通過。原本的 Overflow 等同於 $A_{n-1}B_{n-1}S'_{n-1} + A'_{n-1}B'_{n-1}S_{n-1}$，不過現在 B 的正負號會顛倒過來，因此會變成 $A_{n-1}B'_{n-1}S'_{n-1} + A'_{n-1}B_{n-1}S_{n-1}$。

問題七：reg 初始值

原本在處理 MSB 想要透過一個 while 迴圈執行，設定一個 reg i 從 15 開始，每次遞減 1，去看看 A `&` $(1 << i)$ 是否非零。不過在賦予 i 初始值為 15 的時候出了問題，後來在stackoverflow 的一篇文章找到解答。如果要賦予 reg 初始值有兩種方法

1.
```
reg [7:0] data_reg = 8'b10100110;
```

2.
```
reg [7:0] data_reg;
initial data_reg = 8'b10100110;
```

問題八：Procedural-continuous assignments are not supported by synthesis.

跑過 `make sim` 沒問題後，接下來進行 `make syn` 出現了上述的錯誤訊息，在stackoverflow 的一篇文章當中找到解答。原來是因為我在 ALU 的 always block 當中給輸出賦值時習慣使用 `assign`，這樣的寫法稱做 **procedural assignment**，這種寫法是沒辦法 **Synthesis** 的。將 `assign` 移除後就解決了這個問題。

想要問 TA 的問題

問題一：簡短以及優化

這次設計起來覺得很麻煩的部分有兩個，分別是 Adder 以及 MSB。Adder 的部分成功使用 CLA 實作出來感覺很開心，不過還想知道有沒有更簡單快速的做法可以提供。MSB 的部分這次我只是單純的使用 if-else 來判斷，不知道能不能用迴圈實作，以及其他實作方式。就我所知 LSB 可以透過 x & $-x$ 取得，如果 LSB 有這種方法，那 MSB 是不是也會有呢？

問題二：synthesis 以及 simulation 差異

經過這次的 Lab 可以知道在 synthesis 的設計方式下 always block 當中不能使用 assign，相信還有更多 synthesis 與 simulation 的差異，不過實際上是甚麼造成有這樣的差異存在呢？

問題三：要怎麼畫出漂亮的電路圖

在這次的 report 當中我有嘗試畫了幾個電路圖，不過在畫電路圖常常會把線畫得很亂，不知道有沒有畫電路圖的小撇步，或是相關的軟體可以推薦。我在網路上嘗試找過幾個畫電路圖的網站，不過我們的電路圖當中經常會出現一個 Gate 有兩個以上的輸入的情況，這些網站通常沒有辦法設定更多接腳(或是只是我沒有找到)，所以想請教這個問題。

問題四：有沒有好的 debug 方法

學習 verilog 到現在對我來說最不方便的地方就在於 debug，過去寫 C/C++ 甚至是 python 要測試的時候通常我會在要測試的地方塞個輸出，看看結果是不是如預期，但是在 verilog 當中 debug 的方法是寫 testbranch，但是同時還需要先確保 testbranch 的 code 是沒有 bug 的，這大大增加 debug 複雜度和 Coding 時間，不知道 TA 有沒有其他 debug 的好方法可以提供。