Lab 5
Verilog – Combinational Design

Department of Electrical Engineering
National Cheng Kung University

Outline

1. Verilog 補充
2. Behavioral Design – 使用 `assign`
3. 實作題(一) 4-bit Ripple Carry Adder
4. Behavioral Design – 使用 procedural block
5. 實作題(二) 4-bit 乘法器
6. 實作題(三) 8-bit Carry Select Adder
7. 課間檢查與結報內容
8. 參考資料

Verilog 補充

宣告 Input / Output

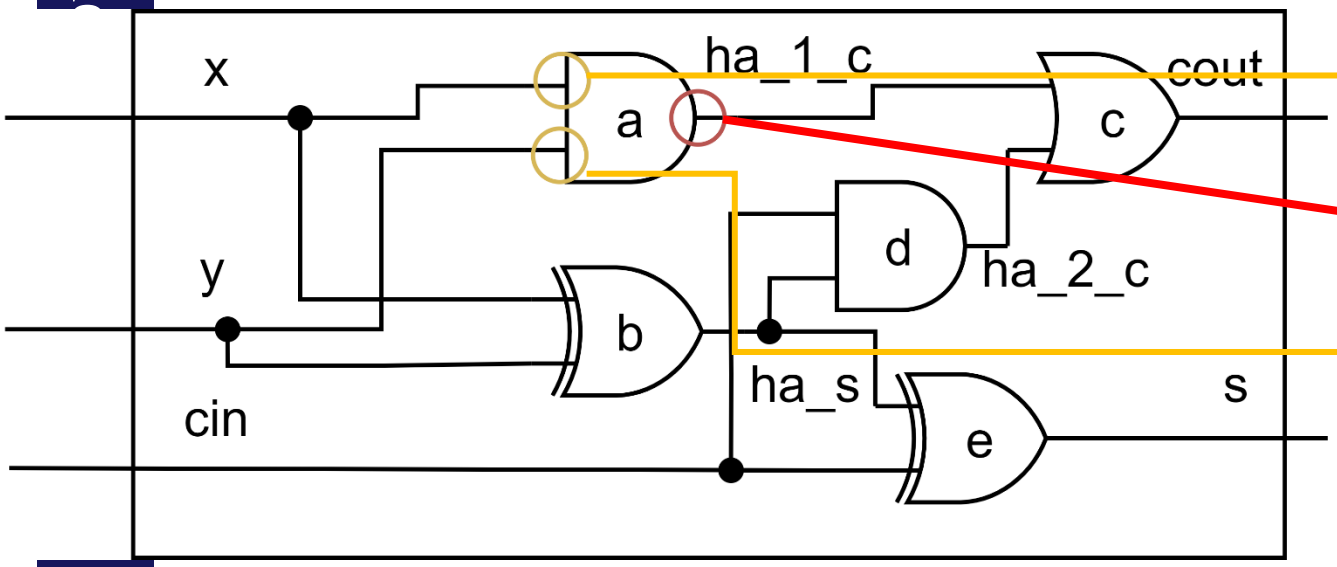
- 宣告 input/output port 有兩種方法，這兩種方法不能混用
  - 在上方標出 input/output 的類型與名字
  - 在上方標出 port 的名字，下方再寫上 input/output 的類型

```
module mux(input sel, input in0, input in1, output out);
    assign out = (sel) ? in0 : in1;
endmodule
```

```
module mux(sel, in0, in1, out);
    input sel;
    input in0;
    input in1;
    output out;
    assign out = (sel) ? in0 : in1;
endmodule
```

Module 連接

- 大家上次在進行實驗的時候都是這樣將 gate 互相連接的
- 這種連接方式叫做 ordered list ，也就是照著這個 module（原生的 gate 也算一種 module）所對應的 port 的**順序**去接。
- 原生的 gate 只支援用 ordered list 進行連接。




```
1    wire x, y, cin;
2    wire ha_1_c, ha_2_c, ha_s;
3    wire cout, s;
4
5    and a(ha_1_c, x, y);
6    xor b(ha_s, x, y);
7    or  c(cout, ha_1_c, ha_2_c);
8    and d(ha_2_c, cin, ha_s);
9    xor e(s, ha_s, c);
```

Module 連接

- 不過如果 module 比較複雜， input/output 繁多
- 可以使用 **connected by name** 的方式進行連接。
- 這樣的好處是即便順序不小心顛倒仍舊可以連接，對於 port 繁多的 module 連線比較方便。

```
1  module operator(input A, input B, input C, input D, input tx_s0, input tx_s1, input rx_s0, input rx_s1, output E, output F, output G, output H);
2      wire channel;
3      mux mux_0(.in0(A), .in1(B), .in2(C), .in3(D), .s0(tx_s0), .s1(tx_s1), .out(channel));
4      demux demux0(.in(channel), .out0(E), .out1(F), .out2(G), .out3(H), .s0(rx_s0), .s1(rx_s1));
5  endmodule
```

```
mux mux_0(.in0(A), .in1(B), .in2(C), .in3(D), .s0(tx_s0), .s1(tx_s1), .out(channel));
```

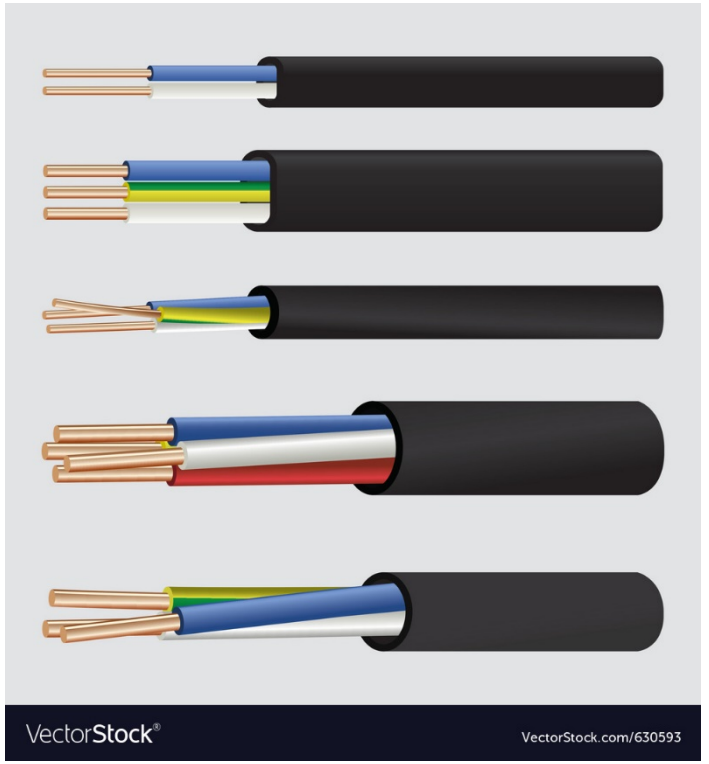
```
module mux(input in1, input in2, output reg out, input s0, input s1, input in0, input in3);
    always @* begin
        case ({s1, s0})
        2'd0: out = in0;
        2'd1:
        2'd2:
        2'd3: out = in3;
        endcase
    end
endmodule
```

```
module mux(input in1, input in2, output reg out, input s0, input s1, input in0, input in3);
```

多條連線 - Vector

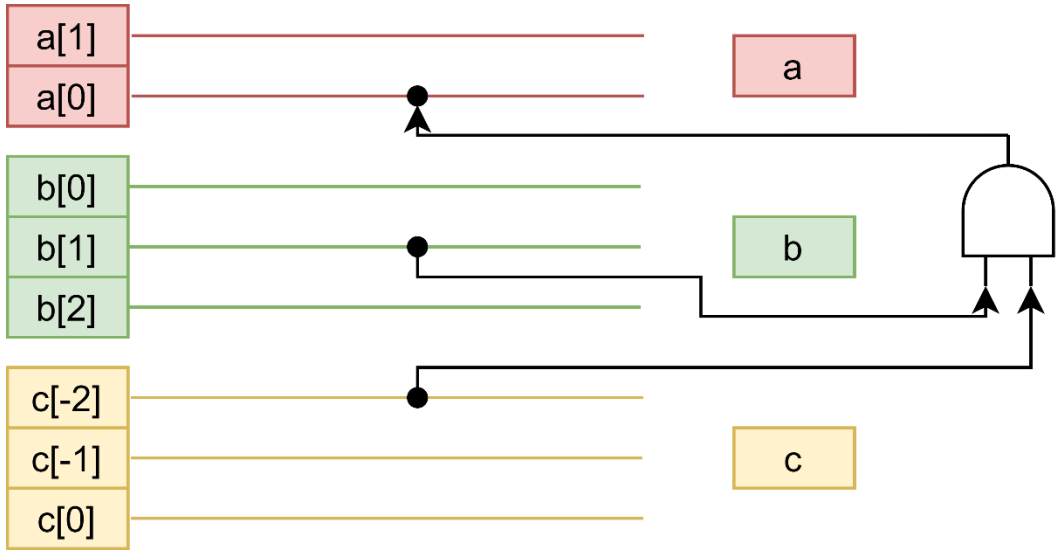
- Scalar - 單一條 wire/單一個 reg.
- Vector - 一組 wire 或是一組 reg.


```
wire [1:0] a;
wire [0:2] b;
wire [-2:0] c;
wire [3:0] d;
wire [2:0] e;
```



多條連線 - Vector

- 可以只連其中一條嗎？ 可以，使用類似 index 的方式即可。

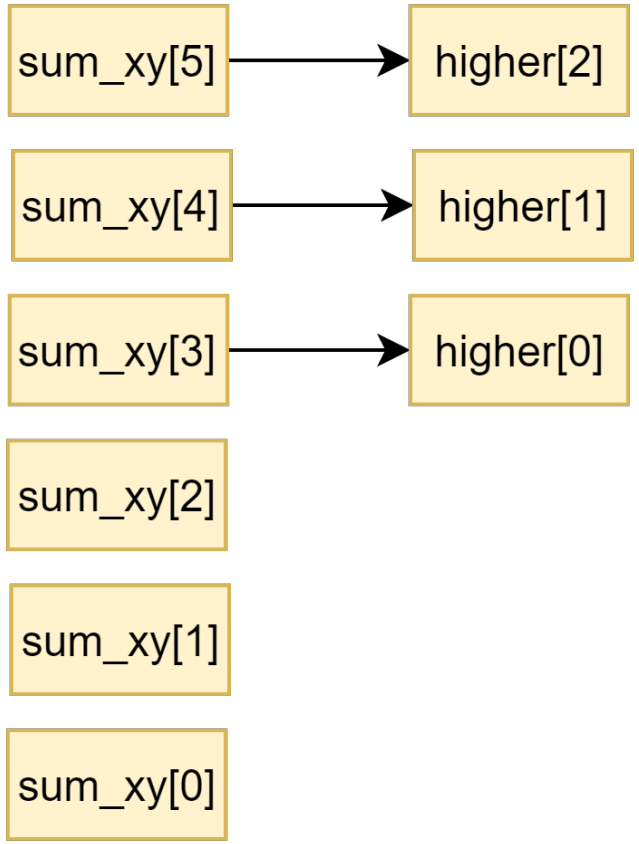



```
wire [1:0] a;
wire [0:2] b;
wire [-2:0] c;

and and0(a[0], b[1], c[-2]);
```

多條連線 - Vector

- 可以一次使用到多條線嗎？ 可以，用跟宣告時候很像的方式。




```
reg[5:0] sum_xy;
wire [2:0] higher;
assign higher = sum_xy[5:3];
```

多條連線 - Vector

- 最高位（MSB）或最低位（LSB）的數字有甚麼限制嗎？
- 建議是以 N-1:0 這樣的方式去命名那 N 條線。


```
wire [3:0] d;
```

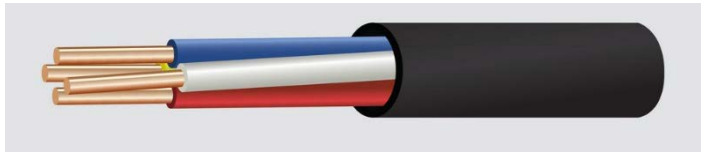
多條連線 - Array

- Array – 可以視為很多組 vector 跟 scalar

```
wire        arr_scalar [0:7];
wire [3:0]  arr_vector [0:7];
```

x8

arr_scalar



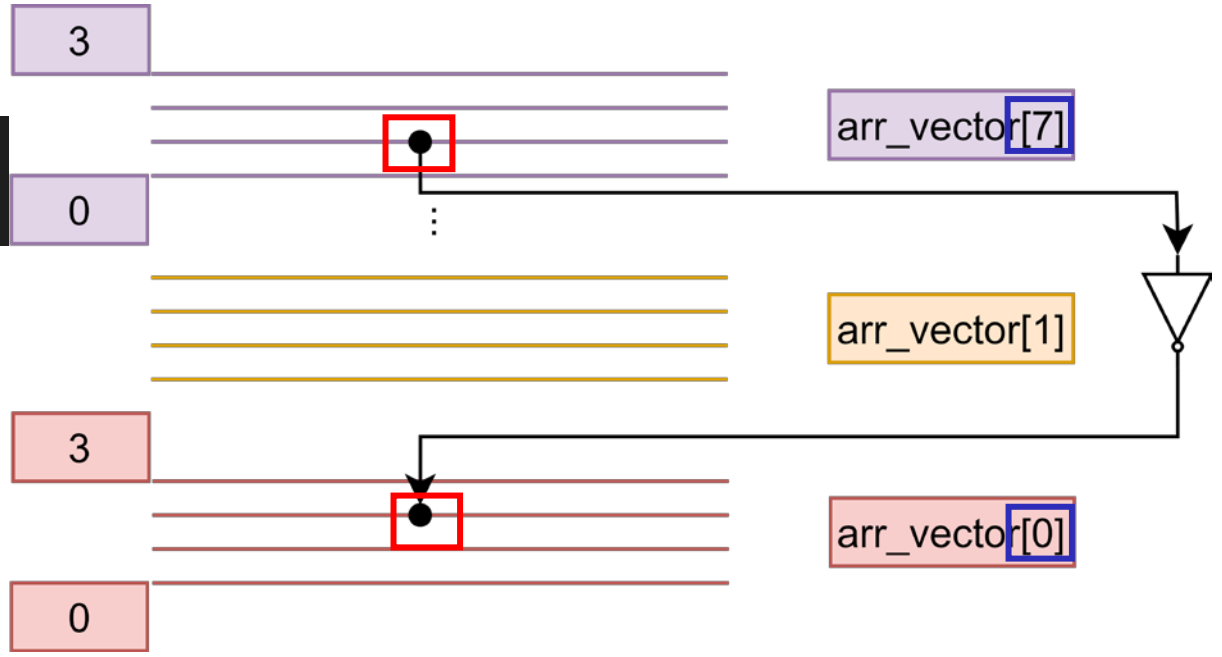
x8

arr_vector

多條連線 - Array

- 那要怎麼在一個都是 vector 的 array 裡面的其中一條線？
- 第一個 index 會是指出「哪一個 vector」，而第二個 index 則是指出「vector 中的哪一條線」。

```
wire [3:0]  arr_vector [0:7];
not not0(arr_vector[0][2], arr_vector[7][1]);
```

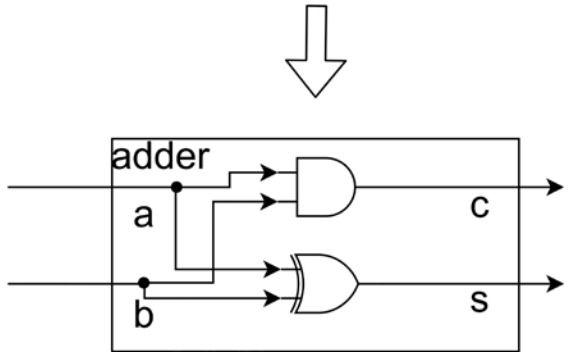


Behavioral Design – 使用 assign

今天不當接線生了

- Behavior level 可以提供更多的設計方式，像是下面這張圖片可以直接以運算元 & 跟 ^ ，就可以取代原本需要 and 與 xor 閘的電路。

```
module adder (input a, input b, output c, output s);
    assign c = a & b;
    assign s = a ^ b;
endmodule
```



- 但大家還是要記得 **Verilog 是在模擬電路，而不是寫程式。**

Behavioral Level 的設計

- 有兩種 Behavioral level 的設計
  - Continuous assignment

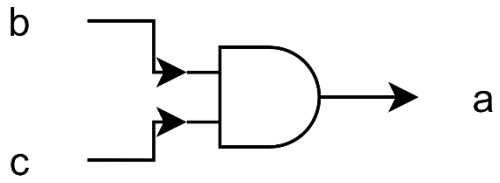
```
assign a = b + c;
```

  - Procedural block

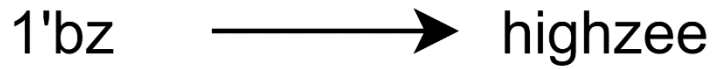
```
always @(*) begin
    a = b + c;
end
```

Continuous Assignment

- continuous assignment 包含四個部分, assign, lhs(左值), = 與 rhs(右值)
- 可以想像成是運算完的右值直接被接到左值的那條線上
- 我們可以把右值設計成需要的表達式，表達式裡面可以包含運算元(wires, constant, regs) 跟運算子



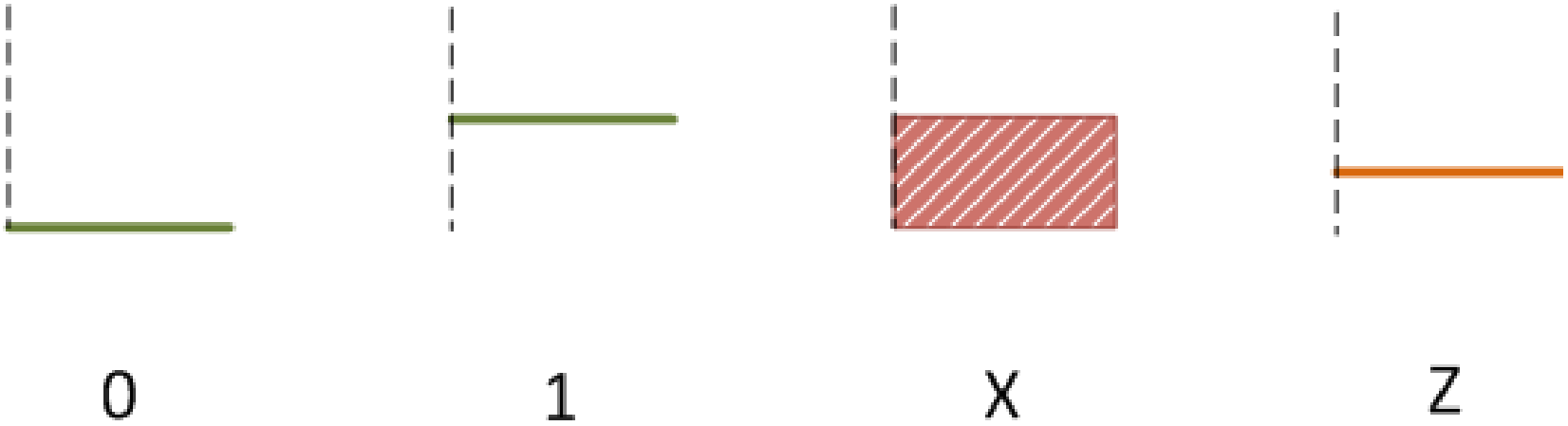

```
wire a, b, c;
assign a = b & c;
```




```
wire highzee;
assign highzee = 1'bz;
```

Behavioral Level 的運算元

- 首先先來說明在 Verilog 裡資料的可能的狀態：
  - 二元運算所需的 0 與 1
  - 不過為了模擬現實中的電路，所以有未知狀態(unknown, x) 與高阻抗(high impedance, z)



Verilog Data Types (chipverify.com)

Behavioral Level的運算元

- 常數：
  - 因為資料在 verilog 裡，可以是任意的 bit 數，不像在 C/C++ 裡面只有 8 的倍數，所以如果有需要，可以在常數的開頭去宣告長度，如果沒有宣告就是 32-bit 。
  - 常數資料也可以是不同進位制，像是 16 進制、10 進制、8 進制或是 2 進制；預設是 10 進制。
  - 為了避免整串常數太長，可以在不是第一個位數的地方寫下底線(_)分割整串常數。

Behavioral Level的運算元

- 常數範例：
  - 659　　是個 32-bit 的 10 進制數字
  - ‘h837FF　　是個 32-bit 的 16 進制數字
  - ‘o7460　　是個 32-bit 的 8 進制數字
  - 6‘d32　　是個 6-bit 的 10 進制數字
  - 4‘b1001　　是個 4-bit 的二進制數字
  - 3‘b01x　　是個 3-bit 的二進制數字，LSB 是未知(x)
  - 16‘b1111_0101_1010_0000 是個 16-bit 的二進制數字，用底線方便肉眼閱讀

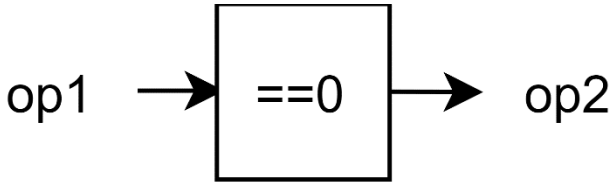
Behavioral Level的運算子

- 現在要來了解在 behavioral level 可以做哪些運算

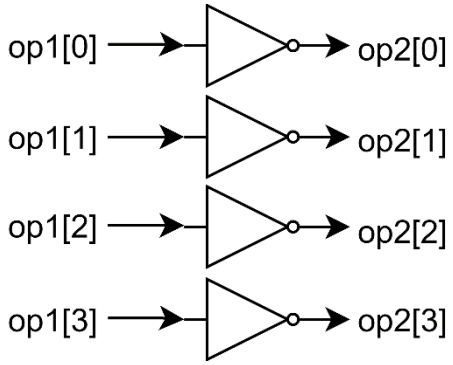
Operator	Description	Usage	Operator	Description	Usage
+, -, *, /, %	Arithmetic operator	op1 + op2	&, |, ^, ^~(~^)	Bitwise operation	op1 & op2
!	Logical negation	!op	~	Bitwise negation	~op
&&, ||, ==, !=	Logical operator	op1 || op2	&, ~&, |, ~|, ^, ~^(^~)	Reduction operator	&op
<<, <<<, >>, >>>	Shift operator	op1 << op2	?:	Conditional	Condition? op1 : op2
{}	Concatenation	{op1, op2,…}	{{}}	Replication	{op1{op2}}

Behavioral Level的運算子

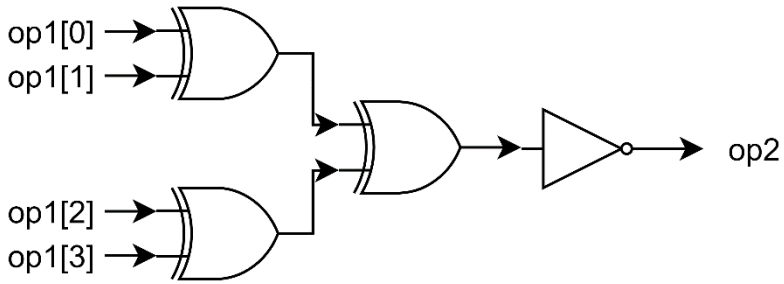
- 首先先來說明一下 negation 與 reduction 的運算
  - Logical negation 會比較運算元，如果運算元等於 0 就會輸出 1 ，如果運算元非 0 就會輸出 0 。
  - Bitwise negation 則會把輸入的所有 bit 都反向。
  - Reduction 會讓運算元的所有 bit 都一起進行同種運算。




```
wire [3:0]  op1;
wire        op2;
assign op2 = !op1;
```




```
wire [3:0]  op1, op2;
assign op2 = ~op1;
```

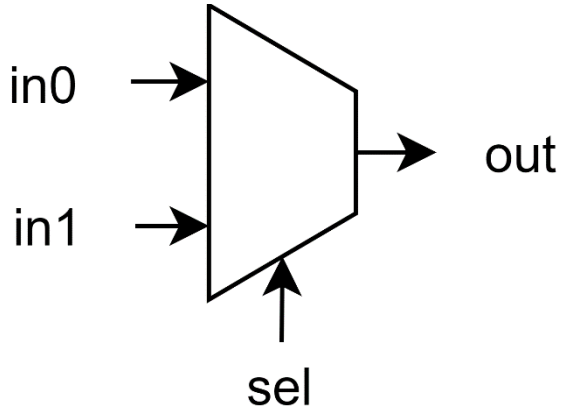



```
wire [3:0]  op1;
wire        op2;
assign op2 = ~^op1;
```

Behavioral Level的運算子

- 三元運算
  - 三元運算是由三個部分組成: 條件, 條件達成的值, 條件沒有達成的值
  - 其中構造為 **條件?條件達成的值:條件沒有達成的值**
  - 可以把它想像成簡單的 Multiplexer

```
wire          sel;
wire[3:0]     in0, in1, out;
assign out = (sel) ? in0 : in1;
```

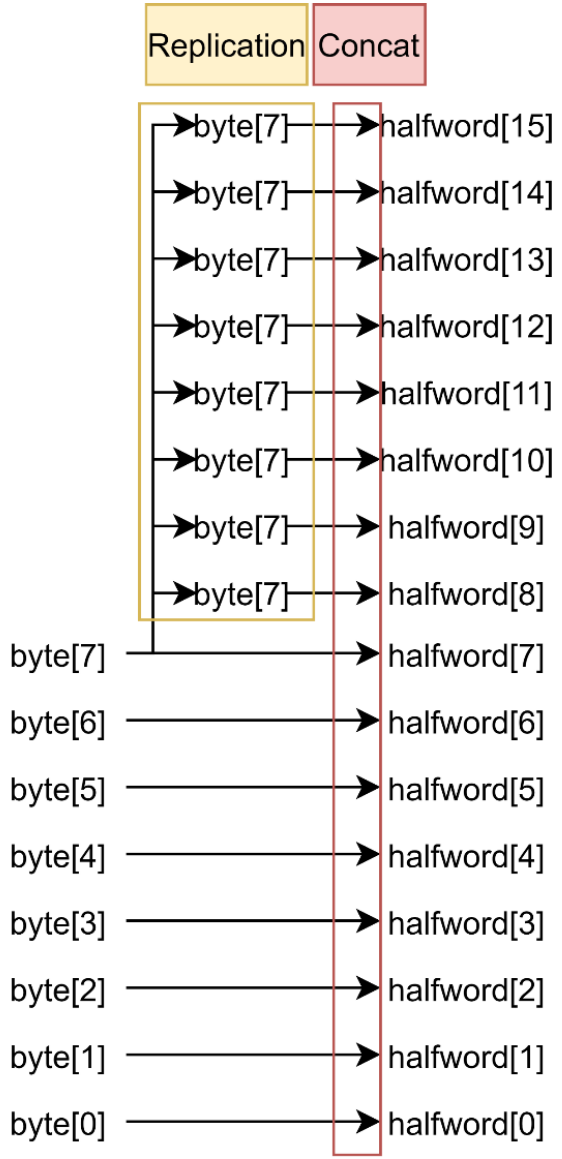


Behavioral Level的運算子

- Concatenation 與 replication
  - 這兩種運算子在處理 vector 時很方便
  - Concatenation 會把兩個運算元連接在一起
  - Replication 會複製裡面那個運算元 k 次，但記得 replication 複製的次數一定要是常數，不能是 wire 、 reg 或其他變數

```
wire [15:0] byte, half_word;
assign half_word = {{4'd8{byte[7]}}, byte[7:0]};
```

一定要是常數



使用 Behavioral Model 進行設計

- 在接線生這個範例裡，我們使用了幾個東西去改寫原本的 module
  - Concatenation
  - Vector
  - Ternary operator

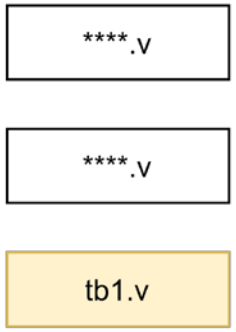
```
1   module operator(input A, input B, input C, input D, input tx_s0, input tx_s1,
            input rx_s0, input rx_s1, output E, output F, output G, output H);
2       wire channel;
3       mux mux_0(.in({D, C, B, A}), .sel({tx_s1, tx_s0}), .out(channel));
4       demux demux0(.in(channel), .out0(E), .out1(F), .out2(G), .out3(H), .s0
                (rx_s0), .s1(rx_s1));
5   endmodule
6
7   module mux(input[3:0] in, input[1:0] sel, output out);
8       assign out = (sel < 2'd2) ? ((sel == 2'd0) ? in[0] : in[1]) :
9                                   ((sel == 2'd2) ? in[2] : in[3]) ;
10  endmodule
```

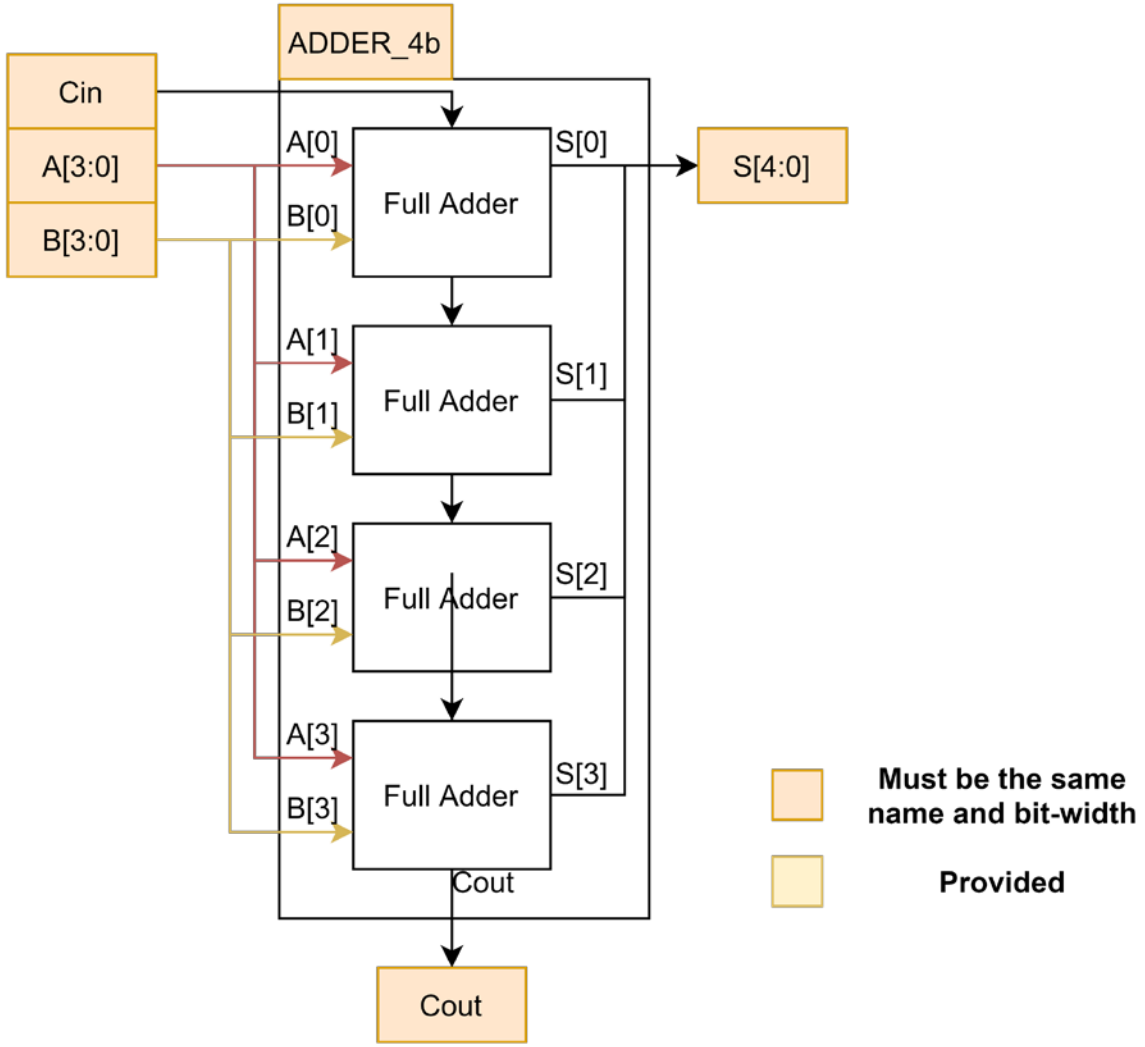
實作題(一) 4-bit Ripple Carry Adder

- 在實作題(一)裡，你們需要完成
  - 1-bit 的 full adder ，需要使用 continuous assignment 的方式去撰寫。
  - 把 4 個 full adder 串接起來便得到一個使用 ripple carry 運算的加法器。
  - 在 FA.v 裡面，使用 behavioral 的方式去寫自己的 full adder 模組。
  - 在 ADDER_4b 那個檔案裡，需要自己去宣告 ADDER_4b 那個模組，並且 instantiate 需要的四個 1-bit 加法器。

實作題(一) 4-bit Ripple Carry Adder

File Structure





Behavioral Design – 使用 Procedural Block

更 Behavioral 的寫法

- 下圖多工器寫法更像是 C/C++ 的形式了

```
module mux(input[3:0] in, input[1:0] sel, output reg out);
    always @ (in or sel) begin
        case (sel)
        2'd0: out = in[0];
        2'd1: out = in[1];
        2'd2: out = in[2];
        2'd3: out = in[3];
        endcase
    end
endmodule
```

- 不過大家還是要記得你們是在**設計一個硬體電路，而非軟體**

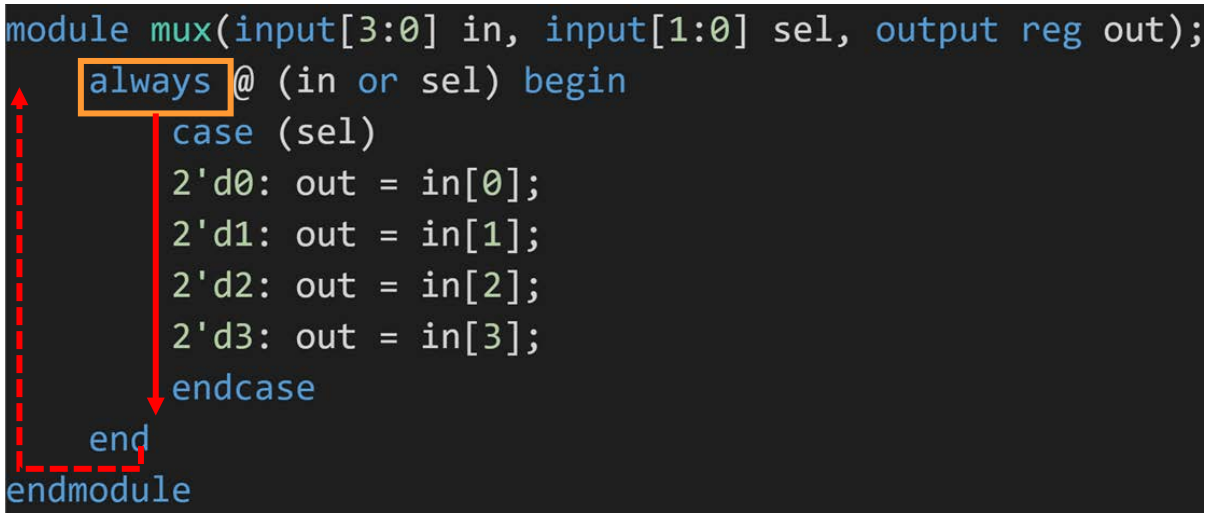
Can We be More Behavioral?

- 為了使用 procedural block 進行設計，有幾件事情需要注意:
  - reg
  - always
  - Sensitive list

```
module mux(input[3:0] in, input[1:0] sel, output reg out);
    always @ (in or sel) begin
        case (sel)
        2'd0: out = in[0];
        2'd1: out = in[1];
        2'd2: out = in[2];
        2'd3: out = in[3];
        endcase
    end
endmodule
```

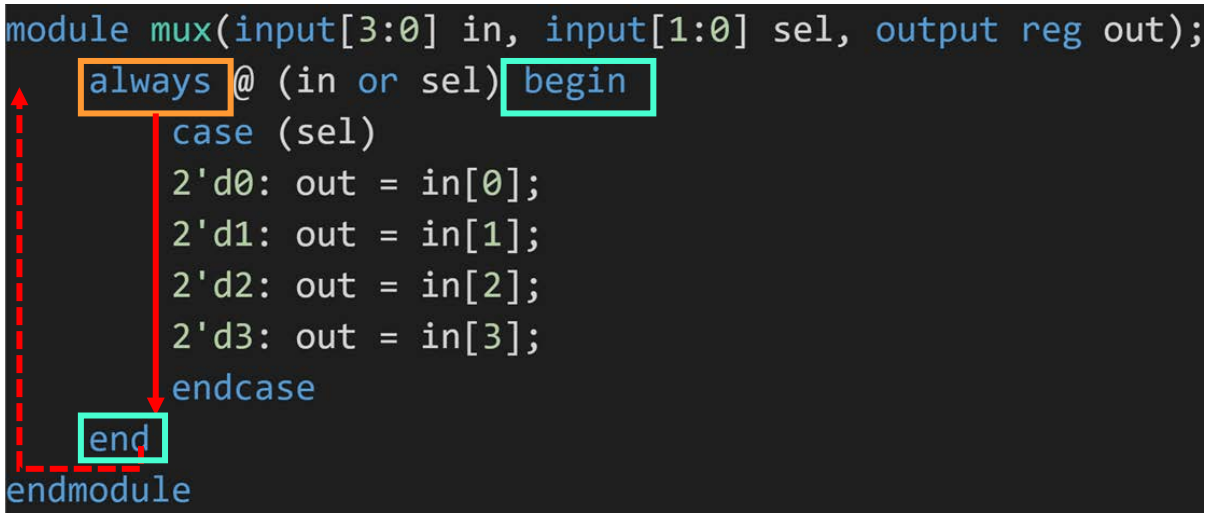
Always Block

- Continuous assignment 相當於把右值（RHS）的結果接線接到左值(LHS)上
- 但 always block 本身並不是一直相連的線路，而是類似一個持續不斷執行的迴圈，當這個迴圈被觸發了，就會執行一次下方的內容，之後再等著第二次被觸發；透過這樣不斷地觸發、執行、觸發、執行，就可以模擬直接接線的設計



Always Block

- 那接下來就要定義這個 always block 會持續不斷執行的範圍，可以透過 begin 與 end 將一個程式片段包起來，類似 C/C++ 的左右括弧，那 always 在被觸發時就會跑整個程式片段；如果沒有使用 begin/end ，那 always 就只會執行到下一個分號的地方
- 不過因為硬體電路都會需要比較多行進行描述，所以使用 always 搭配 begin/end 是一個比較常見的作法



Sensitivity List

- 我們有了一個被觸發就會執行的區段，可以去模擬電路的行為，但我們還要決定甚麼時候這個區段要被觸發
- 這時候就要靠我們的 sensitivity list 出場，他會決定這個 always block 在哪些訊號發生改變時會執行
- 因為需要在右值（RHS）或是 if/case 等等參照到的 wire/reg 有變化時就執行一次 always ，所以 sensitivity list 裡可以放上述的 wire/reg 名稱

```
module mux(input[3:0] in, input[1:0] sel, output reg out);
    always @ (in or sel) begin
        case (sel)
        2'd0: out = in[0];
        2'd1: out = in[1];
        2'd2: out = in[2];
        2'd3: out = in[3];
        endcase
    end
endmodule
```

Sensitivity List

- 不過因為在寫 combinational 電路時，我們不會希望 sensitivity list 裡漏掉任何 input ，所以 Verilog 裡面提供了一個比較方便的語法 always(*)/always* ，這種寫法會讓 simulator 自己去辨識有哪些變數需要被放入 sensitivity list 裡，因此比較建議這種做法

```
module mux(input[3:0] in, input[1:0] sel, output reg out);
    always @ (*) begin
        case (sel)
        2'd0: out = in[0];
        2'd1: out = in[1];
        2'd2: out = in[2];
        2'd3: out = in[3];
        endcase
    end
endmodule
```

```
module mux(input[3:0] in, input[1:0] sel, output reg out);
    always @ * begin
        case (sel)
        2'd0: out = in[0];
        2'd1: out = in[1];
        2'd2: out = in[2];
        2'd3: out = in[3];
        endcase
    end
endmodule
```

Reg Type

- 在 sequential block 裡面如果是左值（LHS）就要被宣告成 reg ，無論是 output 或是內部接線都會需要
- 在有用到 sequential block 的地方宣告 reg 並不代表真正意義上的 register ，他不會產生一個 flip-flop ；這個 reg 在這裡的意思是要讓左值去記住上一次跑完 always block 之後的值

```
module mux_demux(in, sel_mux, sel_demux, out0, out1, out2, out3);
    input[3:0] in;
    input[1:0] sel_mux;
    output reg out0;
    output reg out1;
    output reg out2;
    output reg out3;
    input[1:0] sel_demux;

    reg channel;
    always @ * begin
        case (sel)
        2'd0: channel = in[0];
        2'd1: channel = in[1];
        2'd2: channel = in[2];
        2'd3: channel = in[3];
        endcase
    end
    always @* begin
        case ({s1, s0})
        2'd0: begin
            out0 = channel;
            out1 = 0;
            out2 = 0;
            out3 = 0;
        end
        2'd1: begin
            out0 = 0;
            out1 = channel;
            out2 = 0;
            out3 = 0;
        end
        2'd2: begin
            out0 = 0;
            out1 = 0;
            out2 = channel;
            out3 = 0;
        end
        2'd3: begin
            out0 = 0;
            out1 = 0;
            out2 = 0;
            out3 = channel;
        end
        endcase
    end
endmodule
```

Blocking Assignment

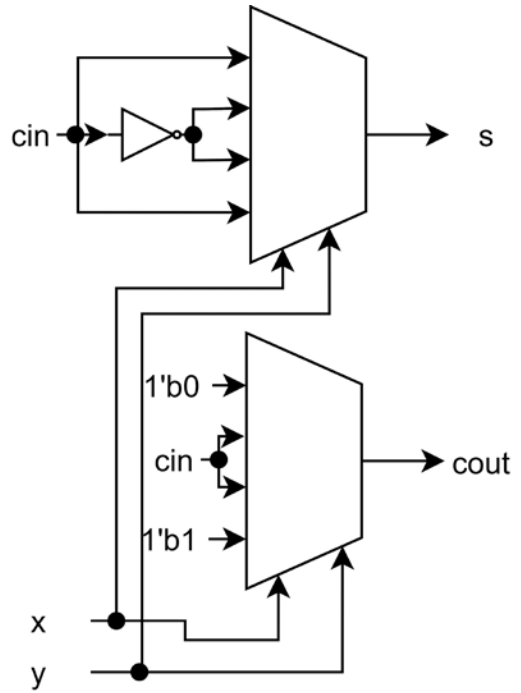
- 因為我們現在要模擬的都是邏輯運算以及接線，沒有 register 那種儲存單元，所以目前都是用 **blocking assignment (=)** ；之後在教 sequential 電路時才會講到 non-blocking assignment
- 使用 blocking assignment 的話，以下圖全加器為例，他會先執行完的第一行，再執行第二行，有點像我們 gate 前後串接的樣子

```
module Adder(input [3:0] x, input [3:0] y, input cin, output reg [3:0] s, output reg cout);
reg[5:0] sum_xy;
always @* begin
    sum_xy = x + y; 1
    {cout, s} = sum_xy + cin; 2
end
endmodule
```

使用 If-else/Case

- 使用 always block 的好處，除了不用一直寫 assign 外，還有一個好處是 mux/demux 可以使用 if-else 或是 case 來實現
- 那下面會使用一個 mux adder 來說明 if-else 跟 case 的用法

```
1  module FA(cout, s, x, y, cin);
2  output reg cout, s;
3  input x, y, cin;
4  always @ * begin
5      case({x, y})
6      2'd0: begin
7          cout = 1'b0;
8      end
9      2'd1: begin
10         cout = cin;
11     end
12     2'd2: begin
13         cout = cin;
14     end
15     2'd3: begin
16         cout = 1'b1;
17     end
18     default: begin
19         cout = 1'b0;
20     end
21     endcase
22
23     if (x ^ y) begin
24         s = !cin;
25     end
26     else if (!x && !y) begin
27         s = cin;
28     end
29     else begin
30         s = cin;
31     end
32 end
33 endmodule
```



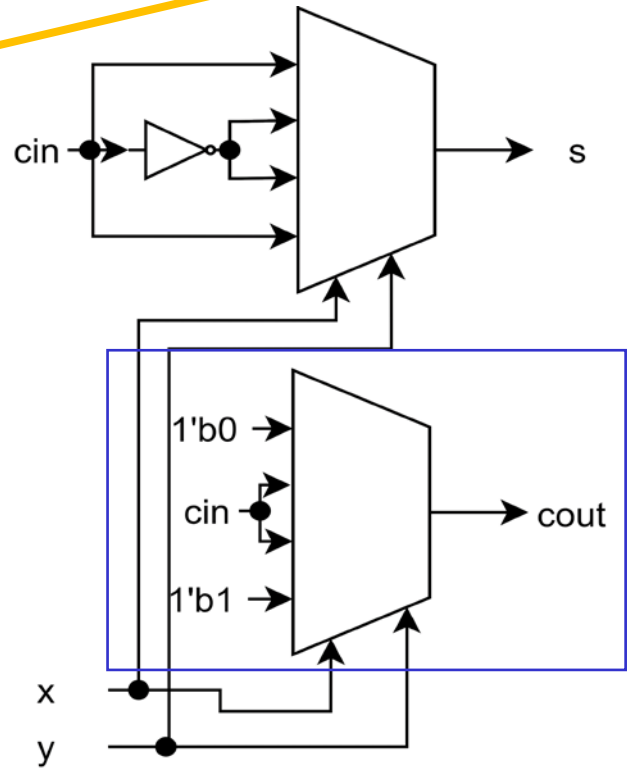
使用 Case

- 要使用 `case` ，就要用 `case(variable)` and `endcase` to enclose all cases, 底下需要列出各種情況，以及各種情況下需要做的行為

```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @ * begin
    case({x, y})
    2'd0: begin
        cout = 1'b0;
    end
    2'd1: begin
        cout = cin;
    end
    2'd2: begin
        cout = cin;
    end
    2'd3: begin
        cout = 1'b1;
    end
    default: begin
        cout = 1'b0;
    end
    endcase

    if (x ^ y) begin
        s = !cin;
    end
    else if (!x && !y) begin
        s = cin;
    end
    else begin
        s = cin;
    end
end
endmodule
```

```
case({x, y})
2'd0: begin
    cout = 1'b0;
end
2'd1: begin
    cout = cin;
end
2'd2: begin
    cout = cin;
end
2'd3: begin
    cout = 1'b1;
end
default: begin
    cout = 1'b0;
end
endcase
```

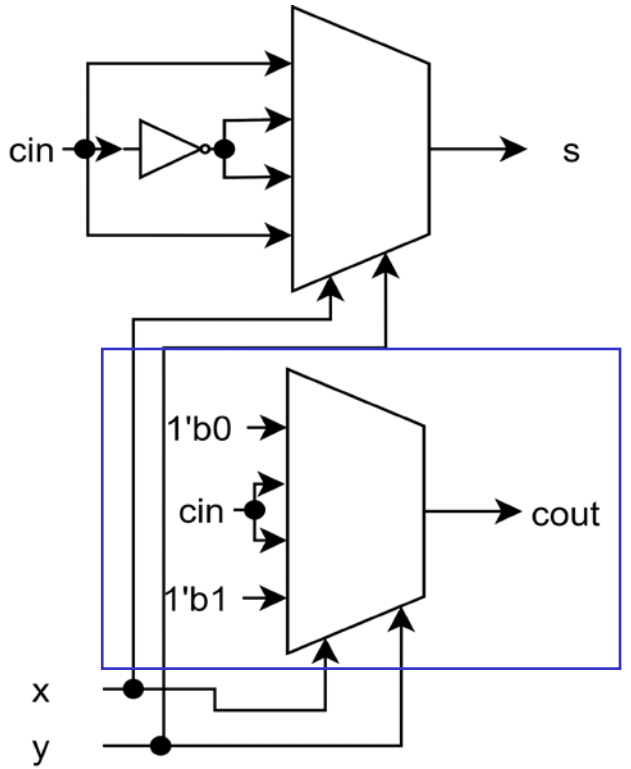


使用 Case – Default Case

- Default case 是為了避免所有情形都沒有 match 到而產生 latch 所寫的情況
- 乍看之下這個 mux 都有相應的 case，但真實世界的 x 、 y 有可能會有 unknown(x) 或是 high impedance (z) 的狀況，所以在 case 還是都要記得補上 default

```
1  module FA(cout, s, x, y, cin);
2  output reg cout, s;
3  input x, y, cin;
4  always @ * begin
5      case({x, y})
6      2'd0: begin
7          cout = 1'b0;
8      end
9      2'd1: begin
10         cout = cin;
11     end
12     2'd2: begin
13         cout = cin;
14     end
15     2'd3: begin
16         cout = 1'b1;
17     end
18     default: begin
19         cout = 1'b0;
20     end
21     endcase
22
23     if (x ^ y) begin
24         s = !cin;
25     end
26     else if (!x && !y) begin
27         s = cin;
28     end
29     else begin
30         s = cin;
31     end
32 end
33 endmodule
```

```
case({x, y})
2'd0: begin
    cout = 1'b0;
end
2'd1: begin
    cout = cin;
end
2'd2: begin
    cout = cin;
end
2'd3: begin
    cout = 1'b1;
end
default: begin
    cout = 1'b0;
end
endcase
```

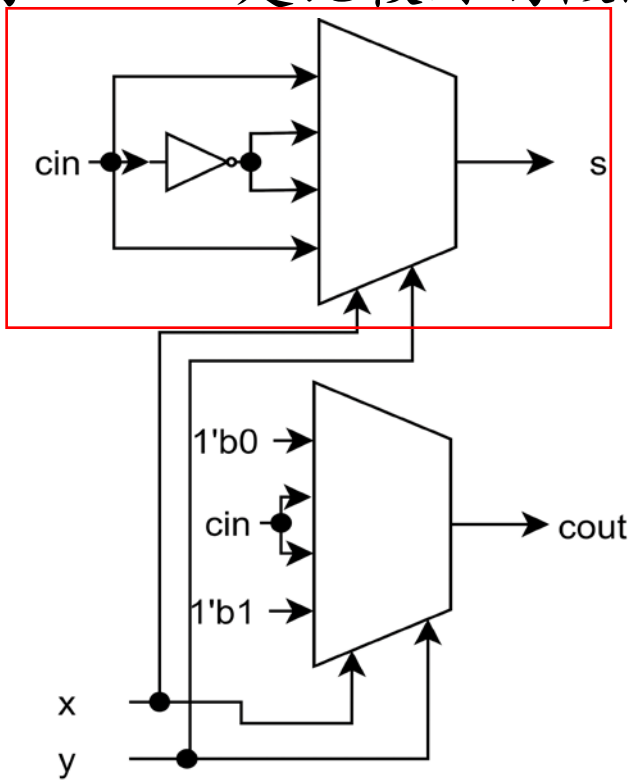


使用 If-else

- 使用 if/else if/else 的情形與 C 蠻雷同
- 同樣的，為了避免 if/else if 的情形都不符合而產生 latch ，所以所有的 if/else if 都加上相應的 else 是比較好的做法

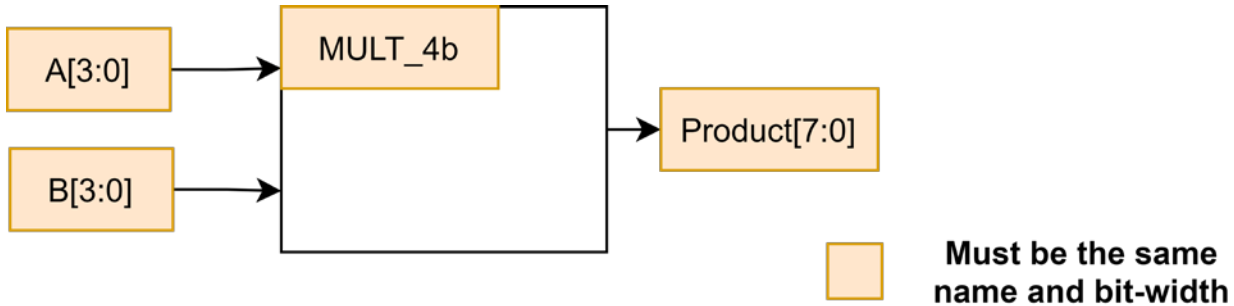
```
1   module FA(cout, s, x, y, cin);
2   output reg cout, s;
3   input x. y, cin;
4           * begin
5       case({x, y})
6       2'd0: begin
7           cout = 1'b0;
8       end
9       2'd1: begin
10          cout = cin;
11      end
12      2'd2: begin
13          cout = cin;
14      end
15      2'd3: begin
16          cout = 1'b1;
17      end
18      default: begin
19          cout = 1'b0;
20      end
21      endcase
22
23      if (x ^ y) begin
24          s = !cin;
25      end
26      else if (!x && !y) begin
27          s = cin;
28      end
29      else begin
30          s = cin;
31      end
32  end
33  endmodule
```

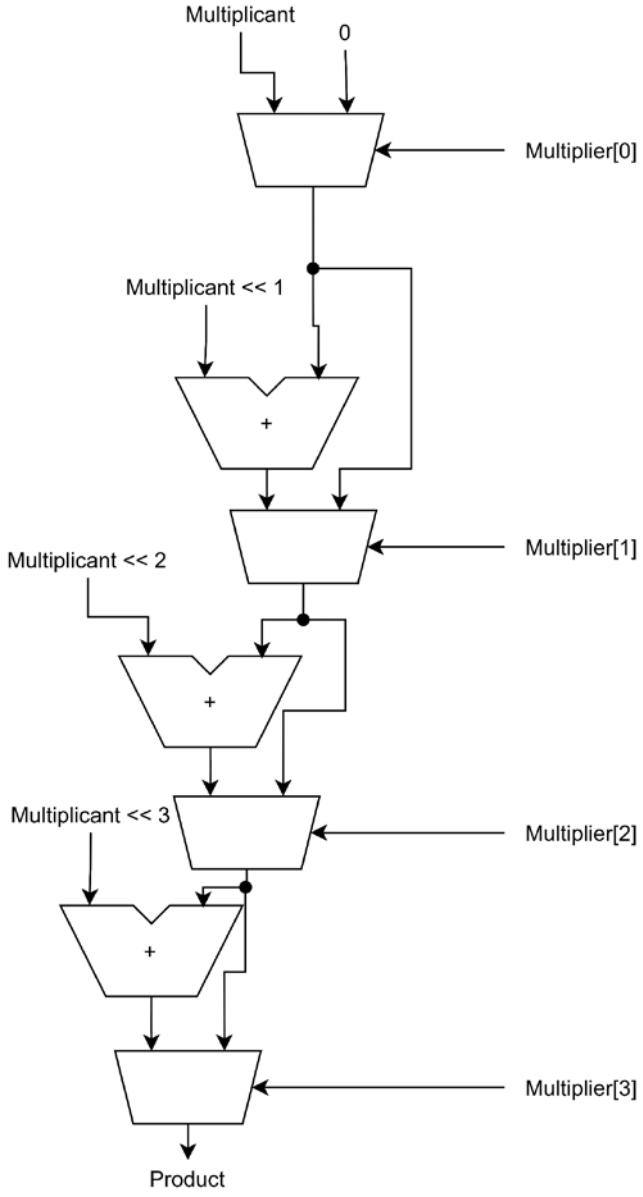
```
if (x ^ y) begin
    s = !cin;
end
else if (!x && !y) begin
    s = cin;
end
else begin
    s = cin;
end
```



實作題(二) 4-bit Multiplier

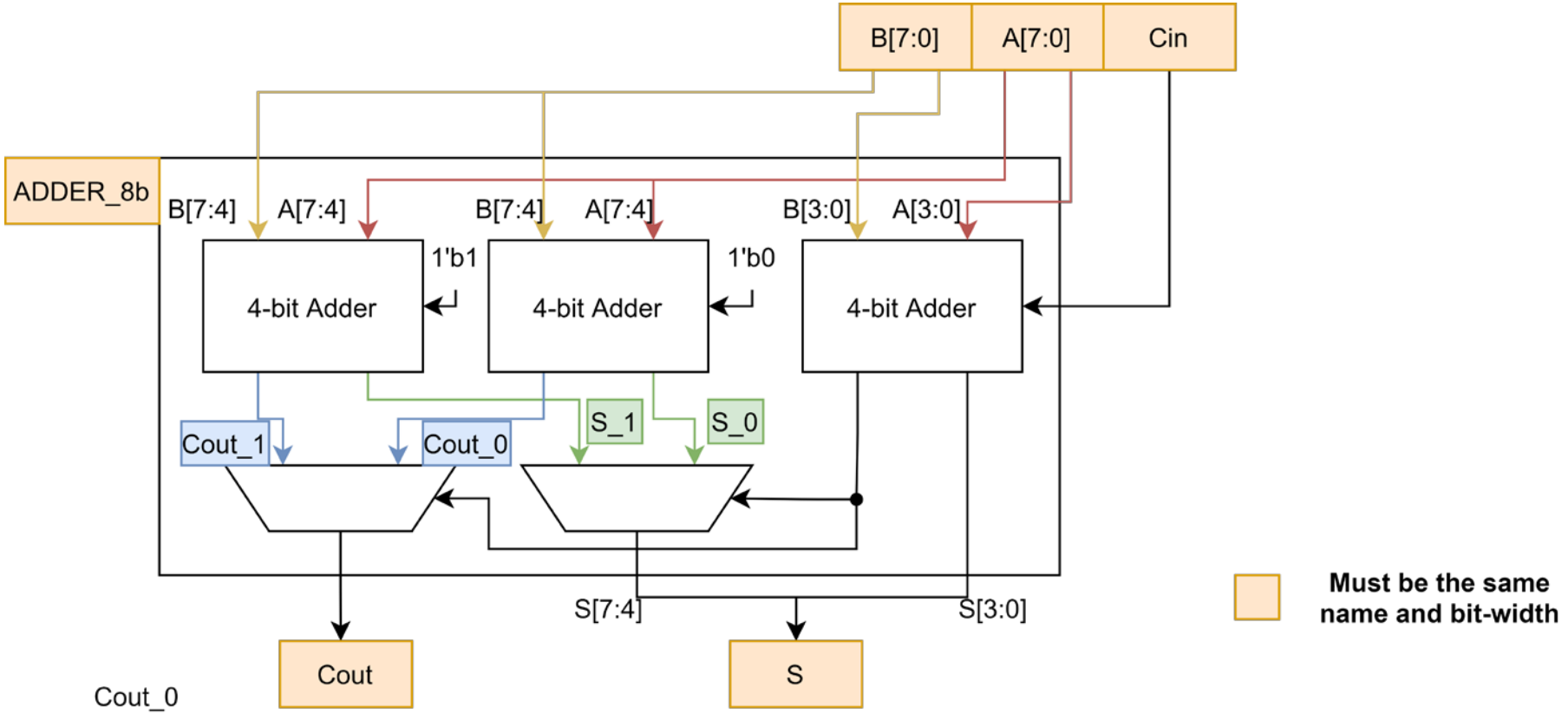
- 使用 tb2.v 並且建立自己的檔案以完成右圖設計





實作題(三) 8-bit Carry Select Adder

- 使用實作題(一)的 4-bit adder 以及 tb3.v ，並自己建立自己的 Verilog 檔案以完成下圖設計



課間檢查與結報內容

課間檢查與結報內容

- 課間檢查
  - 實作(一)~(三)的結果與程式碼
- 結報內容
  - 三個實作分別隨機抽 4 組數針對波形進行說明
  - 比較 8-bit Ripple Carry Adder 與 8-bit Carry Select Adder
    - 比較兩者最長路徑(經過的 gate 數最多的那組 input-output)上的 gate 總數
    - 比較兩者所有 gate 的數量
  - 實驗心得

參考資料

- IEEE Standard 1364-2005