6 CLA & Structural Verilog Modeling

6.1 實驗目的

- 練習撰寫 structural Verilog modeling 並設計 CLA
- 使用 Icarus Verilog Simulator 完成設計驗證

6.2 實驗器材與元件

名稱	說明
 Icarus Verilog	Icarus Verilog 是一套可進行 Verilog 編譯與模擬的免費軟體，其內建 GTKWave 可顯示輸出波型。本次實驗課，會使用 Icarus Verilog 中 iverilog、vvp、gtkwave 來模擬及觀測 CLA 的執行 pattern 的結果和波形。

6.3 實驗內容

同學在先前 Lab 已經學會 verilog 語法以及架構，並會利用 gtkwave 看每個時間點的數值。本週要讓同學用 structural verilog modeling 實作 CLA。

實驗內容如下:

1. 使用 Icarus Verilog 模擬 4-bit CLA (carry-lookahead adder)行為，範例程式碼已提供於 E-course 上，請同學利用 vvp 指令觀察其結果。
2. 利用實驗一的兩個 module 完成 16-bit CLA 的 Verilog design，16-bit CLA 的 testbench 已經附在範例程式資料夾。

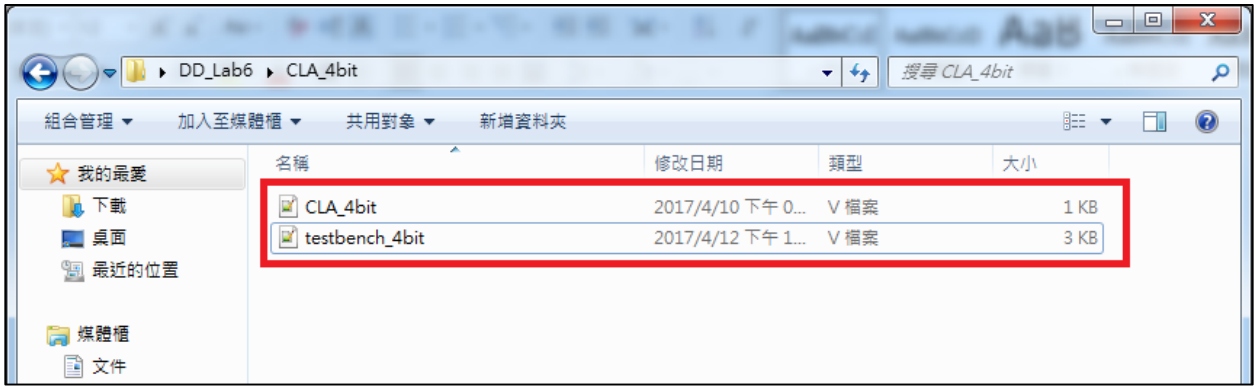
本次實驗主要是藉由 Icarus Verilog 學習 CLA 的 structural Verilog modeling 基本架構，接下來我們將依序介紹本次的實驗。

實驗一

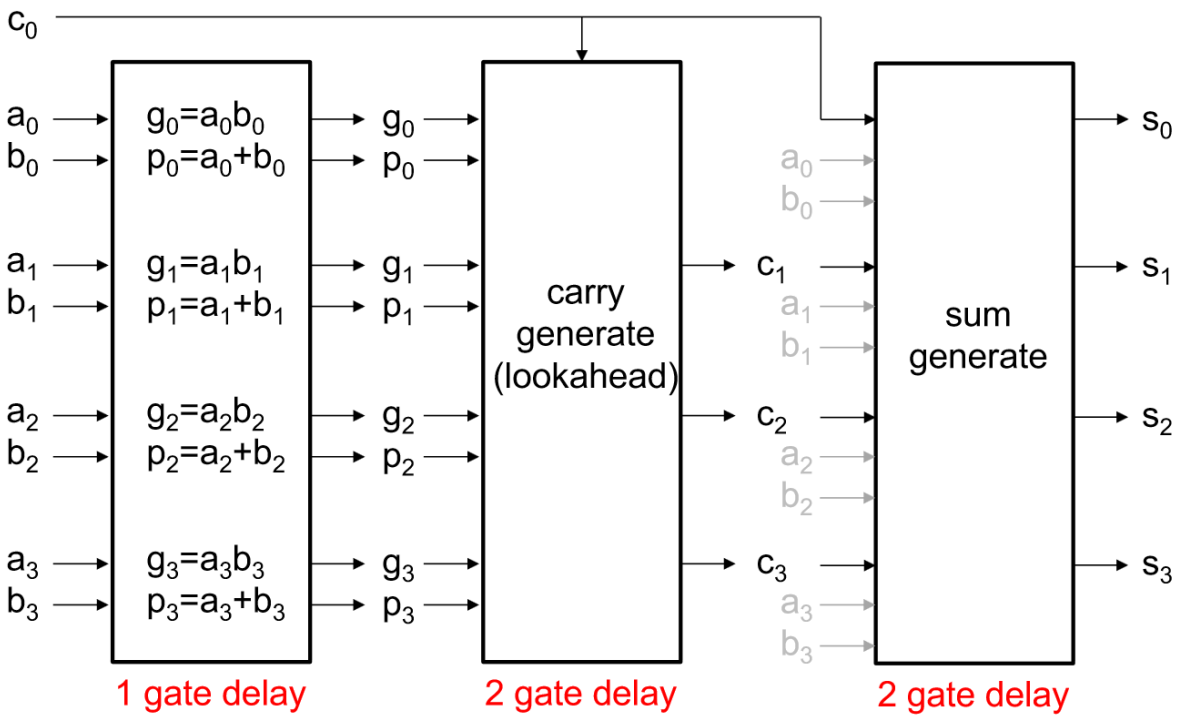
實驗一將透過 4-bit CLA (carry look-ahead adder)的範例，了解 CLA 的 verilog 基本結構並藉由 Icarus Verilog 進行編譯、模擬驗證。

在 Lab5 中，同學已經實作過 RCA，由於 RCA 需要循序執行，所以執行速度慢，因此本次實驗將實作執行速度較快的 CLA。

步驟一：請打開C:\iverilog\bin 資料夾，將本次範例檔案CLA_4bit.v 以及testbench_4bit.v 加入其內(範例檔已放置於E-course上，將DD_Lab6壓縮檔裡的檔案解壓縮後可找到這兩個檔案)。

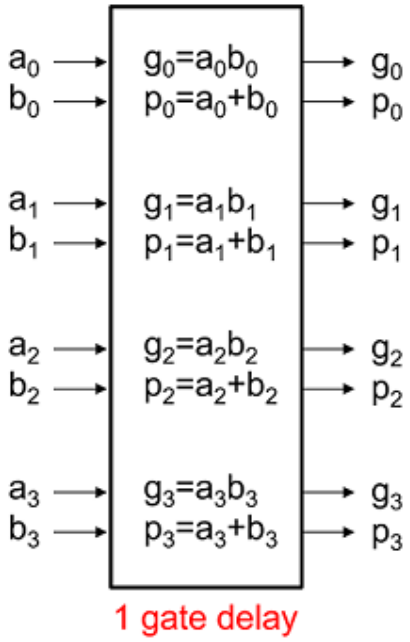


步驟二：利用文字編輯器編輯CLA_4bit.v，並依照下圖Hierarchical Carry Lookahead Adder完成Verilog structural module。在該檔，我們已經完成三個module 的behavior description，同學只需透過下圖block diagram 的設計，了解4bit CLA的架構。



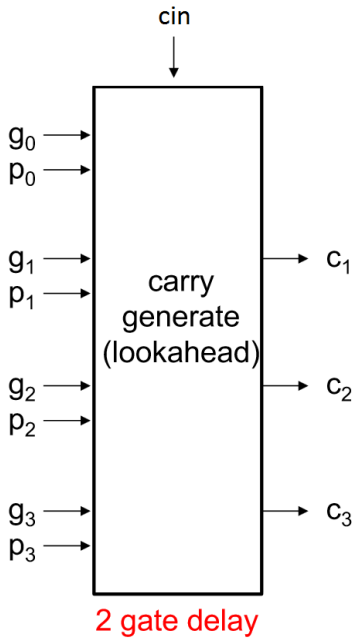
Hierarchical Carry Lookahead Adder

第一個 module 為 Hierarchical Carry Lookahead Adder 圖中左邊區塊的 behavior description。



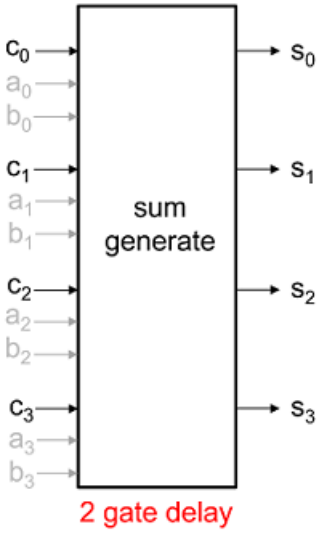

```
//產生所需的小p小g
module gp_generator(a, b, p, g );
      input [3:0] a,b;
      output [3:0] p,g;

      assign g[0] = a[0] & b[0];
      assign p[0] = a[0] | b[0];

      assign g[1] = a[1] & b[1];
      assign p[1] = a[1] | b[1];

      assign g[2] = a[2] & b[2];
      assign p[2] = a[2] | b[2];

      assign g[3] = a[3] & b[3];
      assign p[3] = a[3] | b[3];

endmodule
```

第二個 module 為 Hierarchical Carry Lookahead Adder 圖中中間區塊 behavior description




```
//產生所需的carry in
module carry_generator( p , g , cin , c ,cout);

    input [3:0] p, g;
    input cin;
    output [3:0] c;
    output cout;

    assign c[0] = cin;
    assign c[1] = g[0] | (p[0] & cin);
    assign c[2] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & cin);
    assign c[3] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0]) | (p[2] & p[1] & p[0] & cin);
    assign cout = g[3] | (p[3] & g[2]) | (p[3] & p[2] & g[1]) | (p[3] & p[2] & p[1] & g[0]) | (p[3] & p[2] & p[1] & p[0] & cin);

endmodule
```

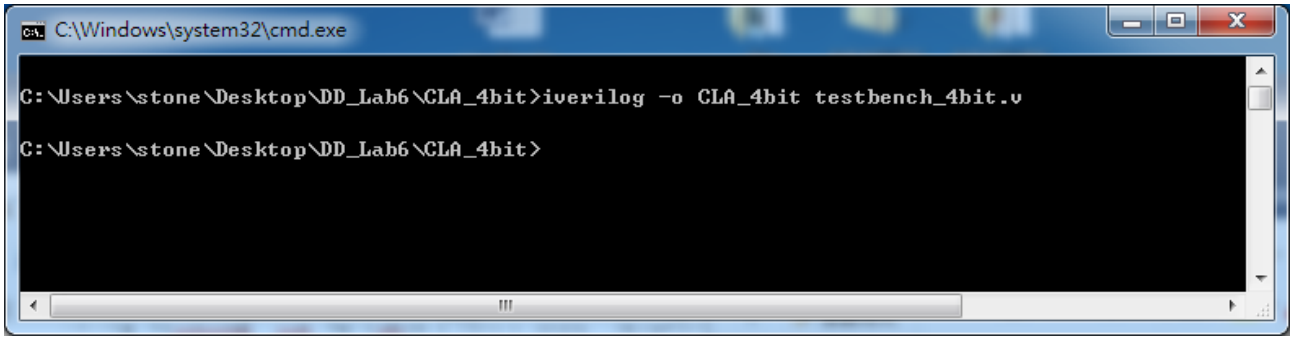
第三個 module 為 Hierarchical Carry Lookahead Adder 圖中右邊區塊 behavior description




```
//求得最後結果
module sum_generator(a, b, c, s );

      input [3:0] a,b,c;
      output [3:0] s;

      assign s[0] = a[0] ^ b[0] ^ c[0];
      assign s[1] = a[1] ^ b[1] ^ c[1];
      assign s[2] = a[2] ^ b[2] ^ c[2];
      assign s[3] = a[3] ^ b[3] ^ c[3];

endmodule
```

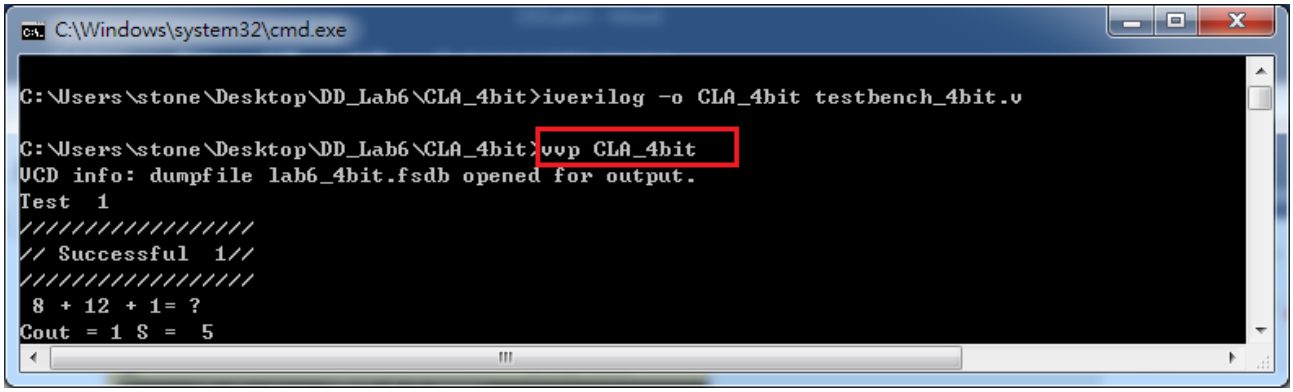
步驟三：藉由iverilog、vvp 指令編譯testbench_4bit.v。我們可以藉由螢幕顯示觀察這20筆 pattern 所運算出的答案是否正確。請先到開始鍵打開命令提示字元 ，並利用Winodow 的 cmd 指令進入C:\iverilog\bin 的資料夾(如下圖)，請注意空白鍵。(由於此電腦有設定環境變數，因此可以不用到C:\iverilog\bin的路徑下編譯)

iverilog –o CLA_4bit testbench_4bit.v

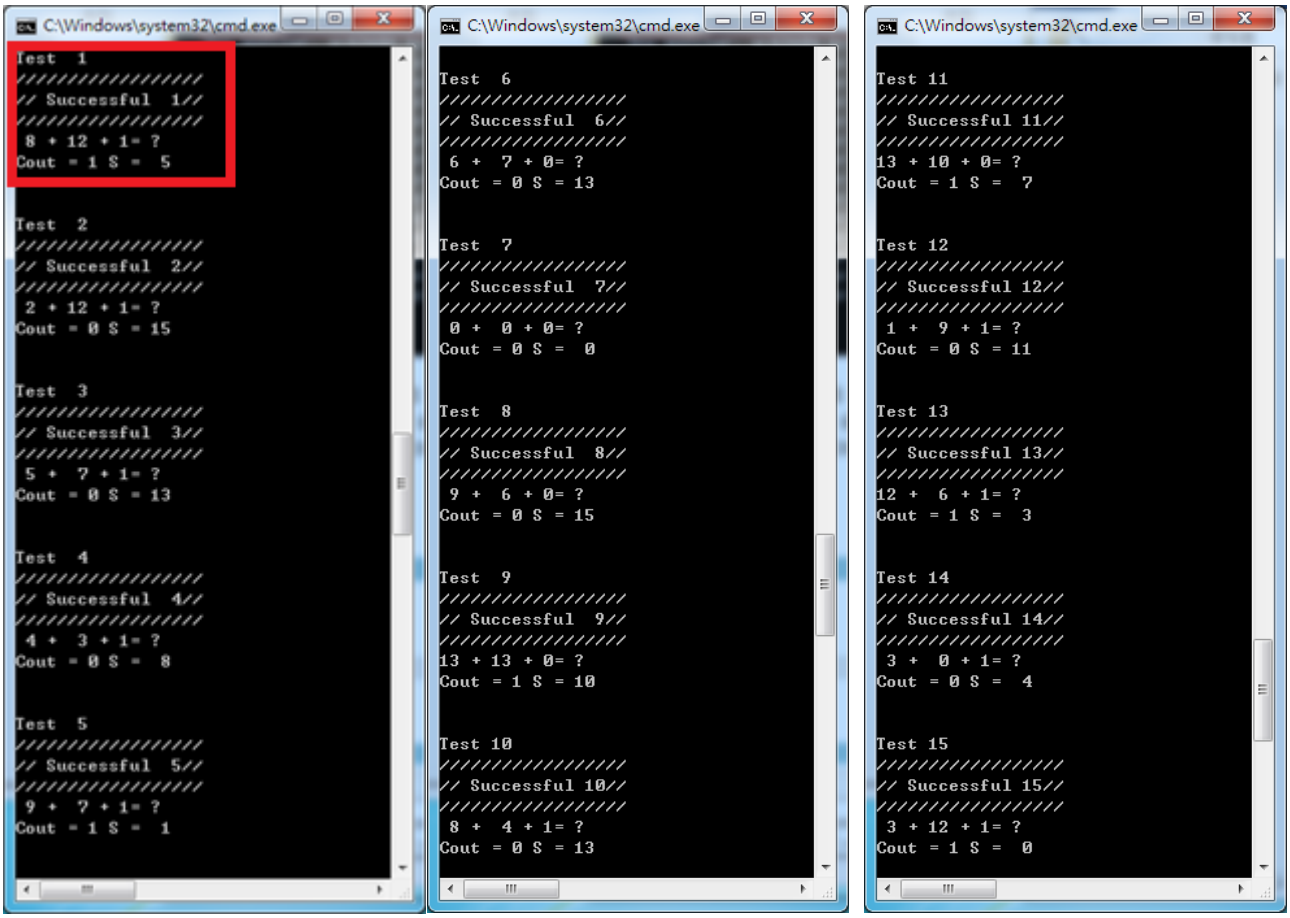


執行 iverilog 完後的檔案，請執行下方指令。

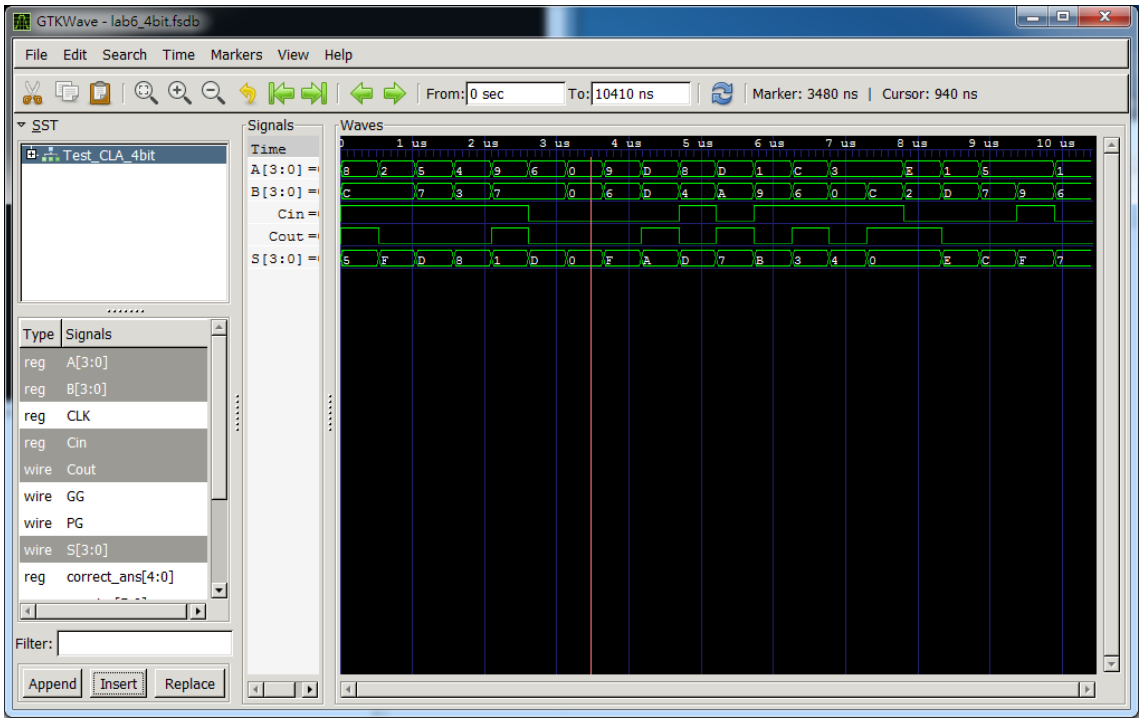
vvp CLA_4bit



同學可以藉由下圖螢幕的輸出，來檢視你在步驟二設計的4-bit CLA功能是否錯誤。
例如下圖紅框內，三個input 分別為a = 8、b = 12、cin = 1，經計算應為8+12+1 = 1 + 5，則 sum 即為5 而carry out 為1，得結果正確。



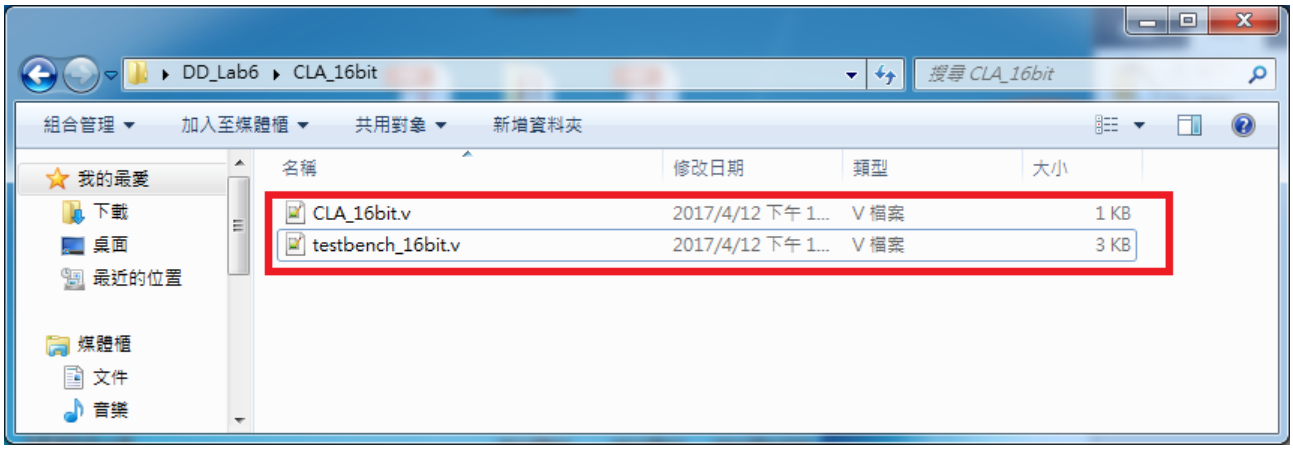
也可以用 lab5 學到的 gtkwave 開啟 lab6_4bit.fsdb，觀察數值的變化



實驗二

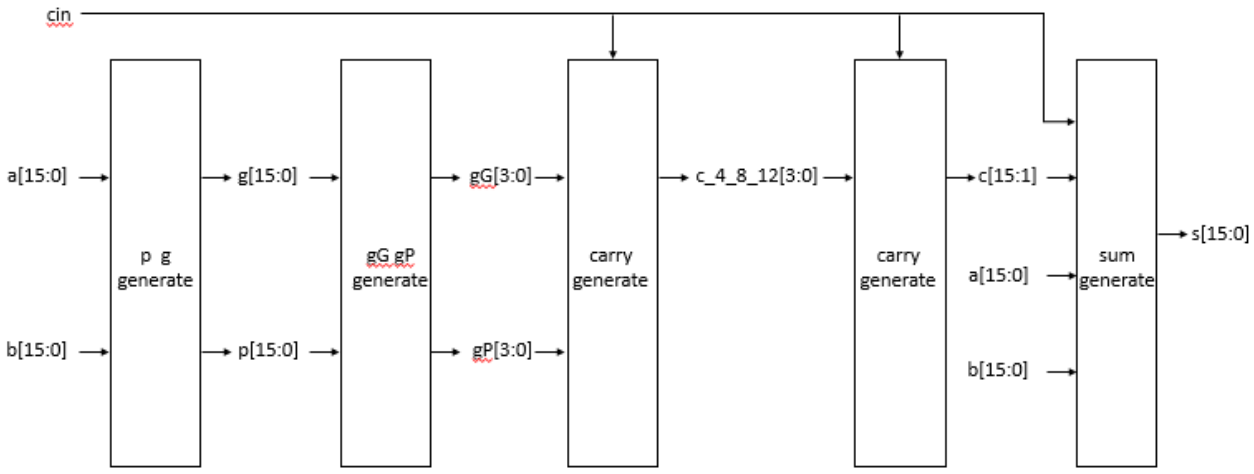
從實驗一中已經了解 4bit CLA 的 verilog 結構。實驗二則是希望透過利用已經實驗一使用的兩個 module 實作出 16bit CLA。

步驟一： 請打開C:\iverilog\bin 資料夾， 將本次範例檔案CLA_16bit.v 以及testbench_16bit.v加入其內(範例檔已放置於E-course上，將LAB6壓縮檔裡的檔案解壓縮後可找到這兩個檔案)。



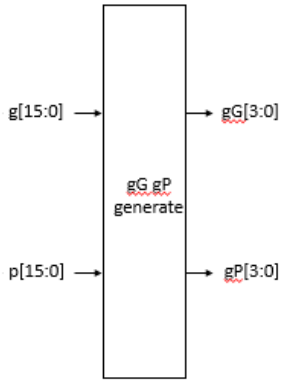
步驟二：利用文字編輯器編輯 CLA_16bit.v，並依照下圖 Hierarchical Carry Lookahead Adder 完成 16bit CLA structural module 描述。

首先說明 16bit CLA，輸入經過一個 gate delay 可求得所有 p、g，經過兩個 gate delay 可求得 group G、group P，再經過兩個 gate delay 可求得 c4、c8、c12，之後經過兩個 gate delay 即可求得所有 carry in，再經過一個 gate delay 即可求得所有答案。



Hierarchical Carry Lookahead Adder

以下說明與實驗一不同的部分:

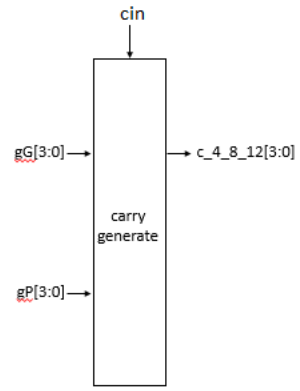


輸入 p g

```
carry_generator carry_gen_1(.p(p[3:0]) , .g(g[3:0]) , .cin(cin) , .c(c[3:0])
carry_generator carry_gen_2(.p(p[7:4]) , .g(g[7:4]) , .cin(c_4_8_12[1]) , .c(c[7:4])
carry_generator carry_gen_3(.p(p[11:8]) , .g(g[11:8]) , .cin(c_4_8_12[2]) , .c(c[11:8])
carry_generator carry_gen_4(.p(p[15:12]) , .g(g[15:12]) , .cin(c_4_8_12[3]) , .c(c[15:12])
```

求得 gG gP

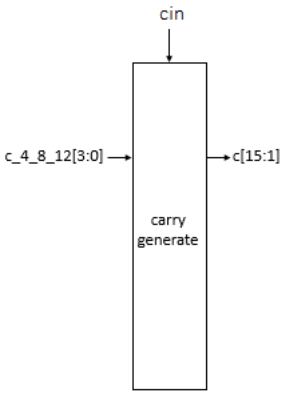
```
,.gG(gG[0]) , .gP(gP[0]));
, .gG(gG[1]) , .gP(gP[1]));
, .gG(gG[2]) , .gP(gP[2]));
, .gG(gG[3]) , .gP(gP[3]));
```



輸入 group G, group P, cin

求得c4、c8、c12

```
carry_generator carry_gen_5(.p(gP) , .g(gG) , .cin(cin) , .c(c_4_8_12) ,.cout(cout));
```



輸入cin

c_4_8_12[3:0]

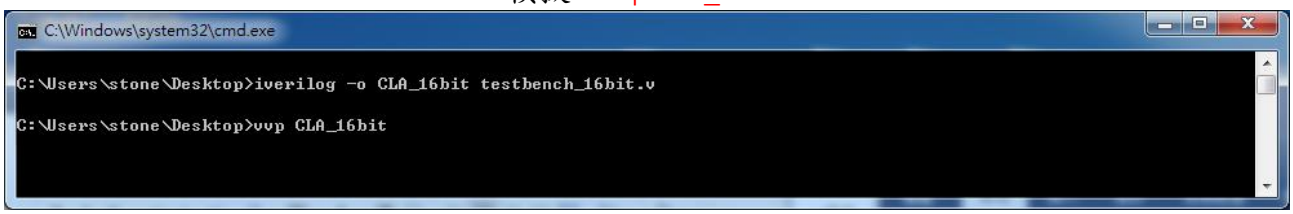
求得所有carry in

```
carry_generator carry_gen_1(.p(p[3:0]) , .g(g[3:0]) ,  .cin(cin) ,  .c(c[3:0])  ,.gG(gG[0]) , .gP(gP[0]));
carry_generator carry_gen_2(.p(p[7:4]) , .g(g[7:4]) ,  .cin(c_4_8_12[1]) ,  .c(c[7:4])  , .gG(gG[1]) , .gP(gP[1]));
carry_generator carry_gen_3(.p(p[11:8]) , .g(g[11:8])  .cin(c_4_8_12[2]) ,  .c(c[11:8])  , .gG(gG[2]) , .gP(gP[2]));
carry_generator carry_gen_4(.p(p[15:12]) , .g(g[15:12])  .cin(c_4_8_12[3]) ,  .c(c[15:12])  ,  .gG(gG[3]) , .gP(gP[3]));
```

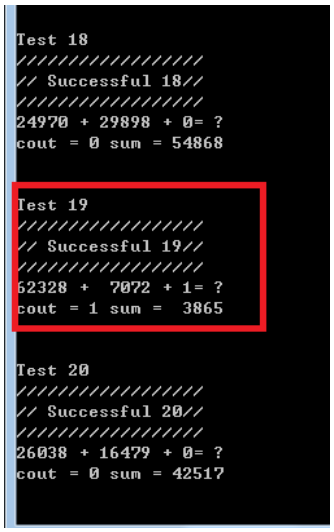
步驟三：藉由iverilog、vvp 指令編譯testbench_16bit.v。利用以下這兩道指令完成編譯和模擬。

編譯：iverilog –o CLA_16bit testbench_16bit.v

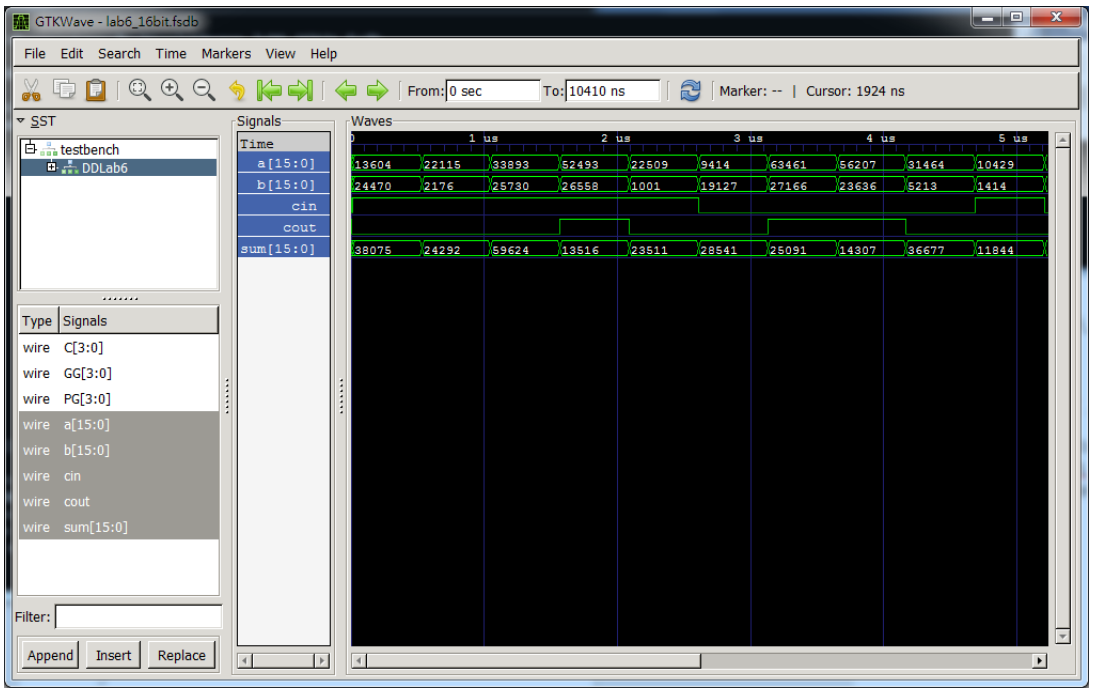
模擬：vvp CLA_16bit



以下圖Test 19 的pattern 為例，a = 62328; b = 7072; cin = 1，經計算後為62328+7072+1 = 1+3865，則sum = 3865 且carry out 為1，得正確結果。



也可以用 gtkwave 開啟 lab6_16bit.fsdb，觀察數值的變化



6.4 練習題：

請同學完成64bit CLA的Verilog design，需使用structural Verilog modeling的方式。練習題需修改的檔案為CLA_64bit，DD_Lab6壓縮檔解壓縮後可找到檔案。CLA_64bit的testbench也在資料夾中。

實驗紀錄簿的實驗結果包含兩個部分：

1.64 bit CLA structural modeling的Verilog code

2.testbench模擬結果。

若無使用structural verilog modeling則視為未完成。

繳交期限:下次上實驗課前上傳至 E-course。

僅需上傳修改過的 CLA_64bit 。

64bit CLA testbench 模擬結果截圖

```
Test  1
////////////////
// Successful //
////////////////
16226723179588100684 + 13176437954090906410 + 1 = ?
cout = 1 sum = 1095641059969455479

Test  2
////////////////
// Successful //
////////////////
 2293428950028369934 + 15686080261204998696 + 0 = ?
cout = 0 sum = 17979509211233368630

Test  3
////////////////
// Successful //
////////////////
10797539492441569866 + 14593661000307487712 + 0 = ?
cout = 1 sum =  6944456419039505962

Test  4
////////////////
// Successful //
////////////////
12483945778981510487 + 16204287138789670912 + 1 = ?
cout = 1 sum = 10241488844061629784

Test  5
////////////////
// Successful //
////////////////
  990478765399375438 +  4145347938394661252 + 1 = ?
cout = 0 sum =  5135826703794036691

Test  6
////////////////
// Successful //
////////////////
16948150326956966292 + 12655037205730758448 + 0 = ?
cout = 1 sum = 11156443458978173124

Test  7
////////////////
// Successful //
////////////////
17072043589202823664 + 12304323107246957792 + 1 = ?
cout = 1 sum = 10929622622740229841

Test  8
////////////////
// Successful //
////////////////
15663087280634926969 +   481047672181217724 + 1 = ?
cout = 0 sum = 16144134952816144694

Test  9
////////////////
// Successful //
////////////////
17544980758103898776 +  2182509820828561952 + 1 = ?
cout = 1 sum =  1280746505222909113

Test 10
////////////////
// Successful //
////////////////
 7393712817001089968 +  7296158426190529808 + 0 = ?
cout = 0 sum = 14689871243191619776

Test 11
////////////////
// Successful //
////////////////
 8293755018743836724 + 15540887475770368513 + 1 = ?
cout = 1 sum =  538789842080465362 2

Test 12
////////////////
// Successful //
////////////////
 4044699813938346008 +  6235916339177919680 + 0 = ?
cout = 0 sum = 10280616153116265688

Test 13
////////////////
// Successful //
////////////////
16279544406665574716 + 13067541765769482188 + 1 = ?
cout = 1 sum = 10900342098725505289

Test 14
////////////////
// Successful //
////////////////
12595099493608435348 +   71754838655199773 0 + 1 = ?
cout = 0 sum = 13312647880160433079

Test 15
////////////////
// Successful //
////////////////
 6145239175589579904 + 15598882358670973680 + 1 = ?
cout = 1 sum =  3297377460551001969

Test 16
////////////////
// Successful //
////////////////
13213154456384126404 + 18397025370343134496 + 0 = ?
cout = 1 sum = 13163435753017709284

Test 17
////////////////
// Successful //
////////////////
17026141831244382832 +  4244563662141061883 + 0 = ?
cout = 1 sum =  2823961419675893099

Test 18
////////////////
// Successful //
////////////////
11135475890514363848 + 16947746856453958192 + 0 = ?
cout = 1 sum =  9636478673258770424

Test 19
////////////////
// Successful //
////////////////
 1310366849899877904 +  1017675568432270432 + 0 = ?
cout = 0 sum =  2328042418332148336

Test 20
////////////////
// Successful //
////////////////
12042640388605648008 +  6882426339790651328 + 0 = ?
cout = 1 sum =   478322654686747720
```