109021115 吳嘉濬

The design of my 4-bit carry lookahead adder

為了節省時間以提高計算效率，我們設計了 4-bit carry lookahead adder(以下稱 CLA)。設計一開始，我先複習了老師講義中所有關於 CLA 的內容，將這些知識充分理解並融會貫通之後，便開始實作。

首先，為了縮短求各個 C 值的算式以方便我們閱覽，我先定義了 Generate function G_i=A[i]&B[i], 還有 Propagate function P_i=A[i]^B[i]，(我使用 wire 來定義這些 signal，並用 assign 賦予其值)，有了這兩項定義，我們可以得到一個恆等式：C_{i+1}=G_i|(P_i&C_i)，如此，我們只需要C_0和各項G_i, P_i (即各項 A[i],B[i])，就能在 3 gate delay 下完成計算，達到省時的效果。

How I construct 16-bit adder with 4-bit CLAs

把 16-bit 的 input A 和 B 各自分成四個區塊，分別是 A[15:12], A[11:8], A[7:4], A[3:0]以及 B[15:12], B[11:8], B[7:4], B[3:0]，讓兩個 input 相對應的位置做配對，如 A[3:0]和 B[3:0]，並利用剛剛建好的 4-bit CLA 做運算。另外，因為 CLA 要考慮c_{in}，所以必須從 least significant bit 開始計算，對應到 4-bit CLA，即從 A[3:0], B[3:0]開始，並以$c_{in} = c_0 = 1'b0$計算，計算後得到的c_{out}是下一個 CLA 運算的c_{in} (下一個 CLA 的 input A, B 是 A[7:4], B[7:4])，利用同樣的方式執行 4 次，先後能得到 S[3:0], S[7:4], S[11:8], S[15:12]以及最後的c_{16}，前 4 項即可組成 16-bit 的 S，c_{16}即整個 16-bit adder 的c_{out}。

Verilog code 的部分，我用 wire 定義c_4, c_8, c_{12}作為前一個 CLA 的c_{out}和下一個 CLA 的c_{in}的"橋樑"，另外，我直接使用 A[3:0], B[3:0], …作為 CLA 的 input，-而被我 comment 掉的 code 中也有提供其他方法，如使用{A[3],A[2],A[1],A[0]}以代表 A[3:0]…等等總共還有三種類似表示方法，一樣都能得到相同結果。

The design and implementation of my ALU

先總結主要的設計方法：所有 always 裡面所用到的 signal 全部都要在 always 外面先用 reg 定義好並給好 bit 大小；所有會用到的 module 都要在 always 外面執行。

Mode 0：

hackmd 上 function 說明中的"<-"，我把他理解為 assign 的概念，如此，便能夠很直覺地寫出"Y=A<<1'b1;"。

Mode 1：

同 Mode 0，把"<-"理解為 assign 的概念，能夠直覺地寫出"Y=A<<<1'b1;"。

Mode 2：

同 Mode 0，把"<-"理解為 assign 的概念，能夠直覺地寫出"Y=A>>1'b1;"。

Mode 3：

同 Mode 0，把”<-”理解為 assign 的概念，能夠直覺地寫出”Y=A>>>1’b1;”，特別需要注意的是，這一行 code 不足以正確做完 arithmetic shift right，因為 most significant bit 會保留 shift 前原本的值，而非以 1’b0 補充，所以需要多加一行 code：”Y[15]=Y[14];”。

Mode 4：

為了避免 bug，我把 16-bit adder 移到 always 外執行，這裡只是把 y1(加法運算完的 16-bit output S)的值賦予 Y；Cout1(加法運算完的 16-bit output Cout)的值賦予 Cout。

關於 overflow 的判斷式：overflow=A[n-1]&B[n-1]&(~Y[n-1])|(~A[n-1])&(~B[n-1])&Y[n-1];，最直觀的理解方式就是當兩個負數(A[n-1]==1 && B[n-1]==1)相加，得到的卻是正數(Y[n-1])==0)；兩個正數(A[n-1]==0 && B[n-1]==0)相加，得到的卻是負數(Y[n-1])==1)時，便意味著 overflow 的發生(overflow==1)。

Mode 5：

執行 A-B 相當於執行 A+(2’s complement of B)，所以我先在 always 外把 ~B+16’b1 賦予 b，使 b 成為 B 的 2’s complement，並比照 Mode 4 的方式進行加法與判斷 overflow，值得注意的是依照題目要求，c_{in}必須是 1’b0，但是 testcase 中的c_{in}沒有被侷限於 1’b0，所以我在使用 16-bit adder 時，讓擺c_{in}參數的位置直接設為 1’b0。

Mode 6：

依照 hackmd 上 function 的說明，能夠直覺地寫出”Y=A&B;”。

Mode 7：

同 Mode 6，能夠直覺地寫出”Y=A|B;”。

Mode 8：

同 Mode 6，能夠直覺地寫出”Y=~A;”。

Mode 9：

同 Mode 6，能夠直覺地寫出”Y=A^B;”。

Mode 10：

同 Mode 6，能夠直覺地寫出”Y=~(A^B);”。

Mode 11：

同 Mode 6，能夠直覺地寫出”Y=~(A|B);”。

Mode 12：

我在 always 外定義 a 為 reg 類別的 signal，並在 always 內讓 a=A[3:0];，a 的值代表著把 1’b1 shift left 的次數，並執行於 16-bit 大小的 Y 身上以成功實作 output 為 16-bit 的 binary to one-hot。

Mode 13：

我的作法是如果 A 和 B 的 most significant bit 相同，則直接比大小，把 most significant bit 即 sign bit 當作是一般的 bit 來比大小，因為如果 A 和 B 的 most significant bit 都是 1’b0 則 A, B 都是正數，大小取決於其他 15 bits 所代表的

數的大小；而如果都是 1’b1 則 A, B 都是負數，把 sign bit 當作一般 bit 時，較大的數(設 A)在取 2’s complement(-A)時會變得較小且是正數；較小的數(設 B)在取 2’s complement(-B)時會變得較大且是正數，所以得到-A<-B，即 A>B，剛好就是把 sign bit 當作是一般 bit 時比大小的結果，如同一般狀態下電腦在比較大小時的結果。

此外，如果 A 和 B 的 most significant bit 不同，則 most significant bit 為 1’b0 的數恆大於另一個數。

Mode 14：

同 Mode 6，能夠直覺地寫出”Y=B;”。

Mode 15：

因為 verilog 的 code 沒有”i++”這種語法，而”i=i+1”的語法也只能在 testbench 使用，所以我想到的方法是用 if else 執行，從 most significant bit 開始往右一個一個檢查，如果碰到某 bit 為 1’b1，則把該 bit 對應的位址賦予 Y。

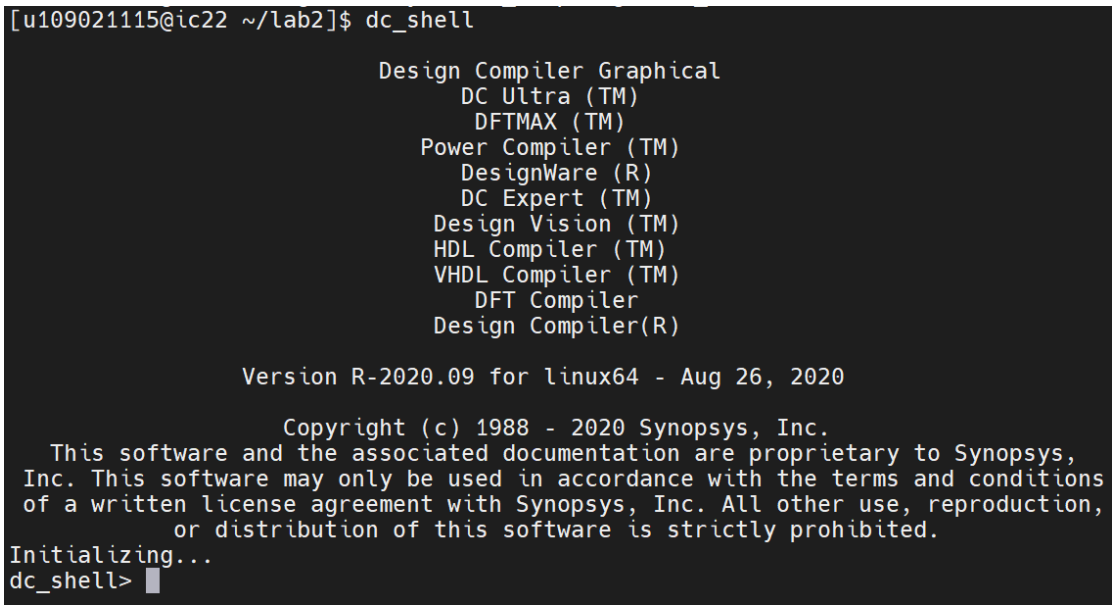
The problems I faced and how I deal with it

實作 lab2 過程中發生了許多的 bug，不論是 module 的設計或是 design vision 的執行，都有出現各式各樣大大小小的 bug。

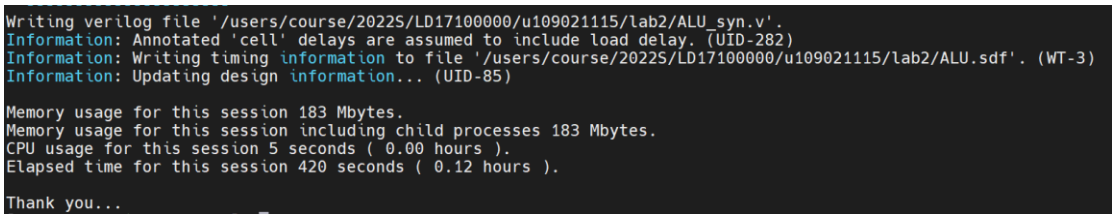
先是 module 的設計，一開始我在寫 ALU 的 Mode 4 和 Mode 5 時，直接把把 Adder_16bit 的使用丟到 always 裡面，在 make sim 時一直出現 error，起初還以為是我的 input, output 給的語法錯誤，所以試了兩三種寫法(這就是為什麼前面介紹我的 16-bit adder 時，提到我給了很多寫法)，結果全部都行不通，當我心灰意冷時，發現我所有內容都有照老師講義裡的 verilog code 的做法做，唯一不同的地方在於：從來沒有在 always block 裡面出現 module，所以我才開始察覺是不是 always block 裡面根本不能放 module，經過一翻的修改，把要使用的 module 放到 always block 前面後，才終於解決了問題。

關於 ALU 的設計，其實在其他 Mode 還有遇到不少問題，像是 Mode 3，一開始我其實不知道電腦不會幫忙保留 most significant bit 的值，所以之後補上了” Y[15]=Y[14];”這一行；Mode 5，沒有注意到 function 的定義是 A-B，沒有考慮c_{in}，所以我直接把對應的 Adder_16bit 的參數c_{in}直接設為 1’b0；Mode 13，起初忘記考慮 sign bit，發現問題後一開始也想不到有效率的解決方法，所以我直接考慮 A, B 兩值可能出現的各種情況，並觀察到上述提及的特性，並寫出相對應的 code；Mode 15，不知道要如何在 always block 裡面操作 iterator，即使知悉不能使用”i=i+1”但還是試了一次，碰壁過後最終決定用 if else 把所有 case 都寫出來。

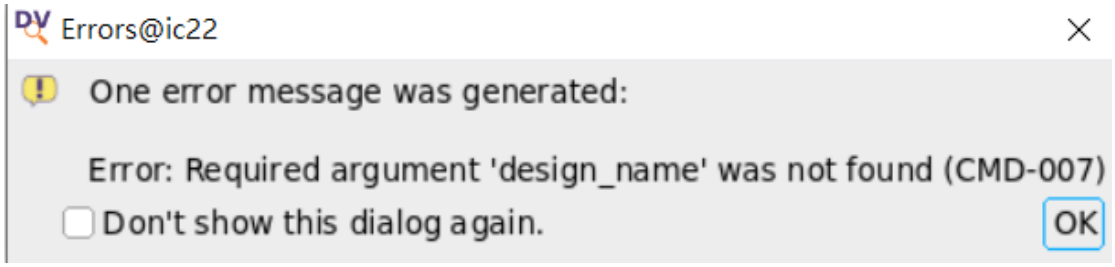
Synthesis 的部分，我在輸入 dc_shell 之後，我發現工作站執行到”initializing…”這一行過後就停住了，少了最後一行，如下圖：



如果我繼續輸入 source ALU.tcl，看似有成功跑完，但是卻遲遲看不到助教說應該要出現的”1”，如圖：



最後，我硬是執行了 design vision，結果在 elaborate 時，出現了 design_name was not found 的問題，如圖：



種種問題頓時出現，加上一整天的 debug 都沒有任何改善，詢問朋友也沒有解決任何問題，甚至在打 report 報告的當下我在討論區問的問題也還沒得到回復，我一度以為這次作業要開天窗了。

直到我發現 design vision 只讀取到我的 CLA_4bit module，所以我猜想，是不是他只會讀取 ALU.v 檔案中第一個 module(正常情況下或許不應該是這樣)，所以我把 CLA_4bit 和 Adder_16bit 這兩個 module 移到 ALU module 後面，讓 ALU 變成 ALU.v 檔案中第一個 module，改完之後再重新跑了一次流程，沒想到這樣改竟然讓我成功執行 synthesis 了！三種 module 的 schematic 以及 synthesized schematic 都能夠呈現出來，另外，這次產生的 ALU_syn.v 檔案我 make syn 後測資全部都有通過！我想，即使 synthesis 的過程跌跌撞撞，只要願

意持續努力並嘗試修改各種可能出現 bug 的地方，如此一來，所有問題都將能迎刃而解！

The questions I want to ask TAs

助教們會以上傳的檔案內容來評分，所以即使 lab 在實行中有一些小瑕疵(並回報在 report 報告中)，只要瑕疵不會出現在上傳的檔案內就不會被扣分嗎？