Lab 1: Verilog HDL

Experiments

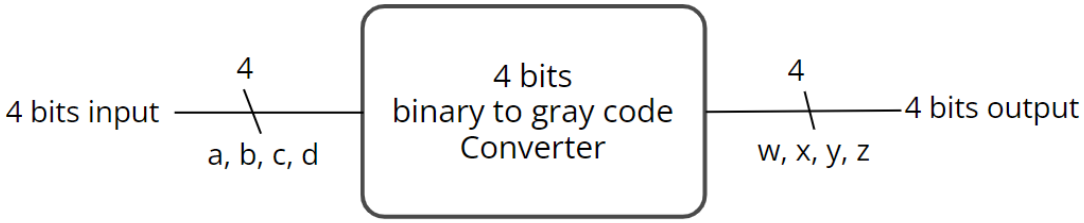
1. Design and verify a binary-to-Gray-code converter for a Gray code sequence with 10 code words (input: abcd, output: wxyz, a and w are the MSB).

Design Specification

IO 輸出入設定:

輸入: a, b, c, d, 每個大小皆為 1bit (a 為 MSB

輸出: w, x, y, z, 每個大小皆為 1bit (w 為 MSB

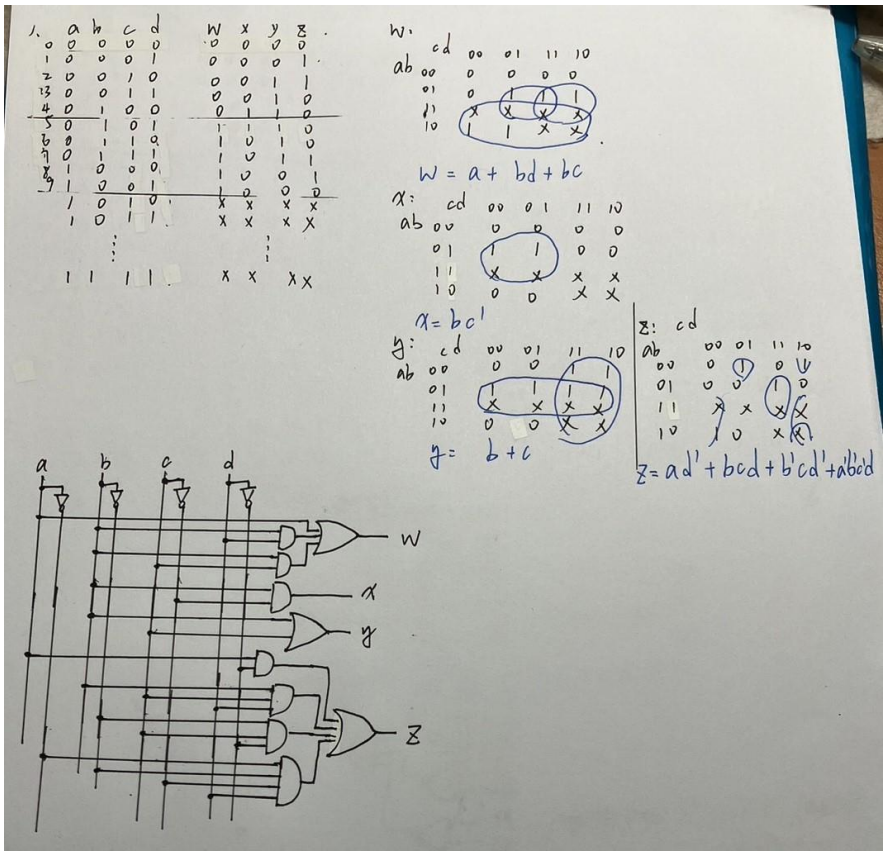


Design Implementation

a. Logic diagram and truth table:

見右圖→

abcd 10~15 為 don't care condition



b. Logic Func.

用卡諾圖進行化簡

w = a + bd + bc

x = bc'

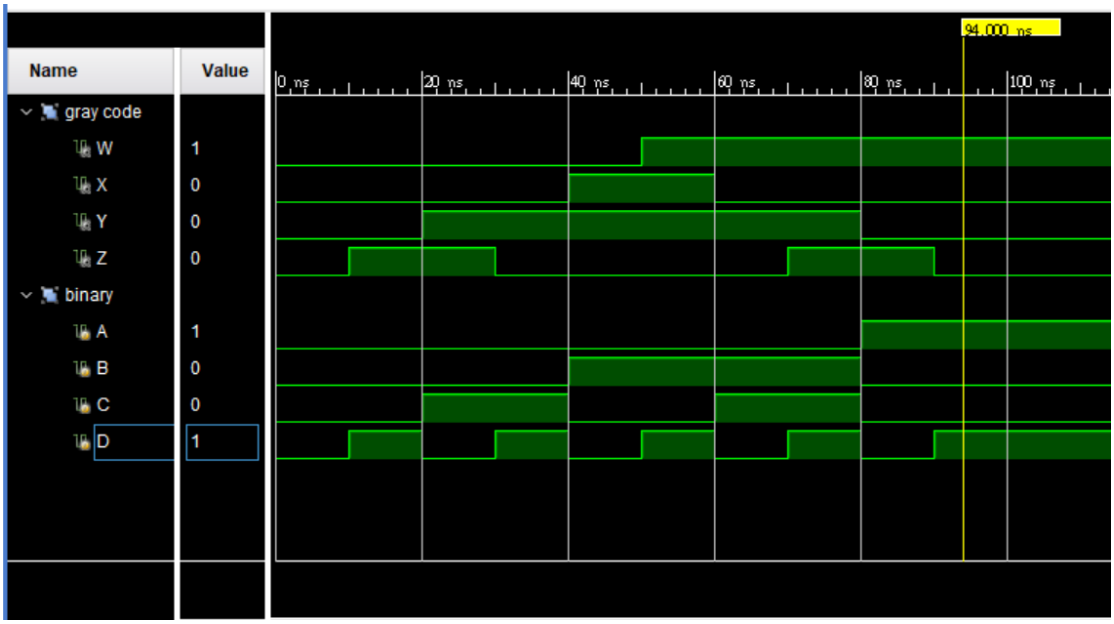
y = b + c

z = ad' + bcd + b'cd' + a'b'c'd

c. Code:

(請見附件程式碼檔案)

Discussion



圖中黃線所在的位置是最後一筆輸入和輸出。
在輸入 abcd 為 0~9 時都符合題目要求的 gray code，其餘的輸入(10~15)由於題只要求 10 個 gray code 便沒有額外執行。在實作過程中全部都是用 continue assignment 處理，設計電路也是直接用上學期學到的組合電路的思維去做。由於 Truth table 和化簡過程皆沒有出錯，因此輸出的 wavetable 也符合我的預期。

想法:
在寫程式的過程中，我在想這些邏輯運算子能不能支援三元運算，還是只能支援二元，且彼此之間的優先度會不會影響。為了保險起見，我用括號來處理 logic func.。這部分可能還是得去查一下 Verilog 的運算子優先權順序。(優先度見右圖→

另外一個想法是在寫完後仍覺得程式有個缺點，就是我覺得程式寫太死板了，缺乏彈性。若今天改為要求 12 個 gray code，整個程式碼便要打掉重寫。但我還未想出解法，仍嘗試找出其他想法來解決這樣的問題。

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	logical bit-wise reduction reduction reduction reduction reduction reduction
+ -	unary (sign) plus unary (sign) minus	arithmetic arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
* / %	multiply divide modulus	arithmetic arithmetic arithmetic
+ -	binary plus binary minus	arithmetic arithmetic
<< >>	shift left shift right	shift shift
> >= < <=	greater than greater than or equal to less than less than or equal to	relational relational relational relational
== !=	logical equality logical inequality	equality equality
=== !==	case equality case inequality	equality equality
&	bit-wise AND	bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	bit-wise bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Conclusion

在經過這實驗後，我學到了

- Assign 的使用
- 基本邏輯運算子的使用
- Module 輸出入的設定
- Testbench 的寫法

Assign 是基本的組合邏輯會用到的寫法，而邏輯運算子的結構也蠻簡單的。我覺得大致上和 C 相同，只多了一個~negative 的運算，和 C 的 ！ 寫法不同。
Module 是我覺得較難的地方，和 C 一樣都要先宣告變數型態。但這變數型態和 C 就有很大的不同了，尤其是 reg 的應用還需要多點時間摸索。
Testbench 遇到的問題大致上跟前面提到的一樣，另外麻煩的是寫 Testbench 要寫每個 condition，這部分就由點枯燥繁瑣。
雖遇到許多問題和疑惑，但整體上還是學到了很多，且也很能感受到 C 和 Verilog 的不同。

References

https://class.ece.uw.edu/cadta/verilog/operators.html
如同前面提到的運算子優先權，經查詢後才發現原來 OR 的優先層級較 AND 低，所以在處理時要注意。還有 NOT 的運算，雖然是一元且優先層級很高，但我認為加個括號還是比較好，且能增加程式可讀性。

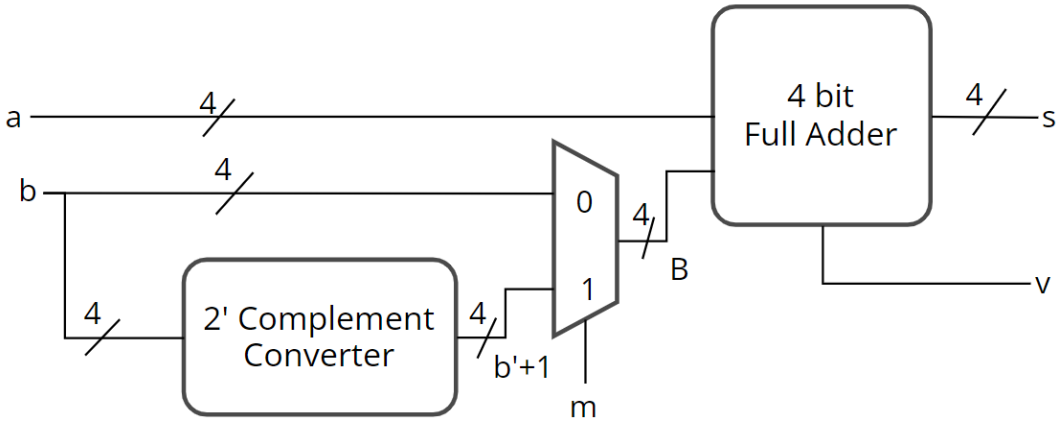
2. Design a signed 4-bit binary adder/subtractor with input a (a3a2a1a0), b (b3b2b1b0), m as theoperator control (0 for addition and 1 for subtraction); output s (s3s2s1s0), v as overflow indicator.

Design Specification

IO 輸出入設定

輸入: [3:0]a, [3:0]b, m 前面兩個大小皆為 4 bits(最左邊 a[3], b[3]為 MSB) , m: 1 bit

輸出: [3:0]s, v 大小分別為 s: 3 bits(最左邊 s[3]為 MSB), v: 1 bit

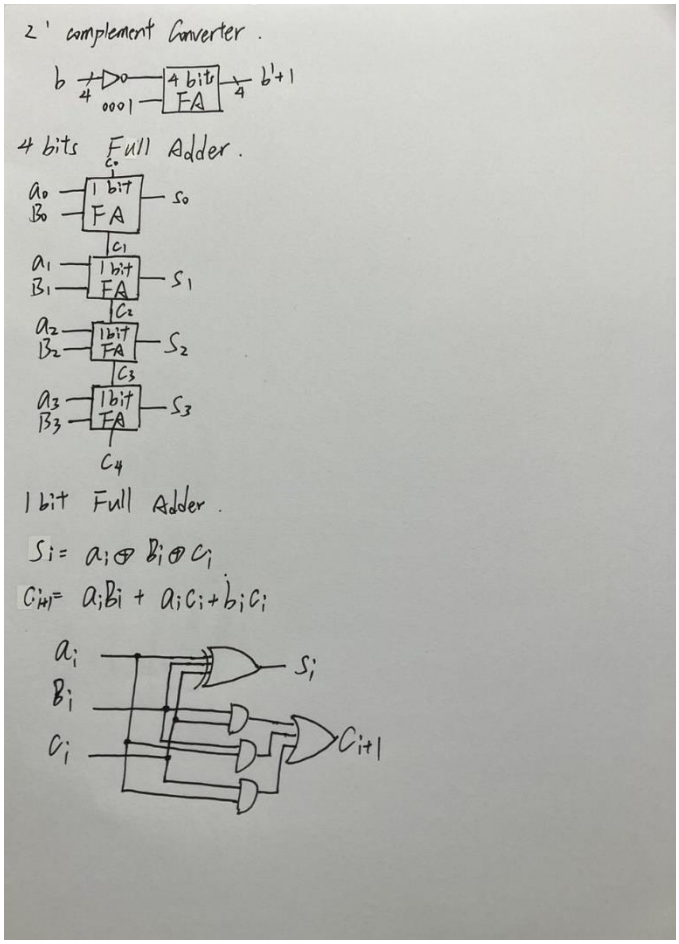


Design Implementation

a. Logic diagram and detaily design

見右圖→

程式內我另外宣告兩個變數分別為 B 和 c，前者紀錄經過多功器(if else)的 b，後者記錄加法中每一位的進位。(在實作 Converter 時是直接進行+算術運算子)



b. Logic func.

$s_i = a_i \oplus b_i \oplus c_i$

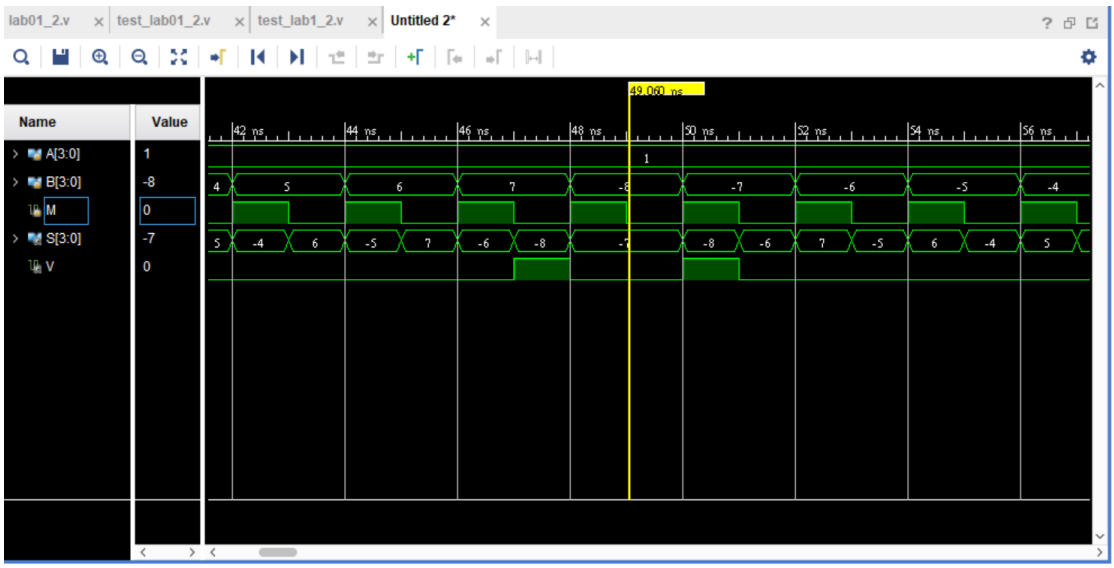
$c_{i+1} = a_i b_i + b_i c_i + a_i c_i$

$v = c_4 \oplus c_3$

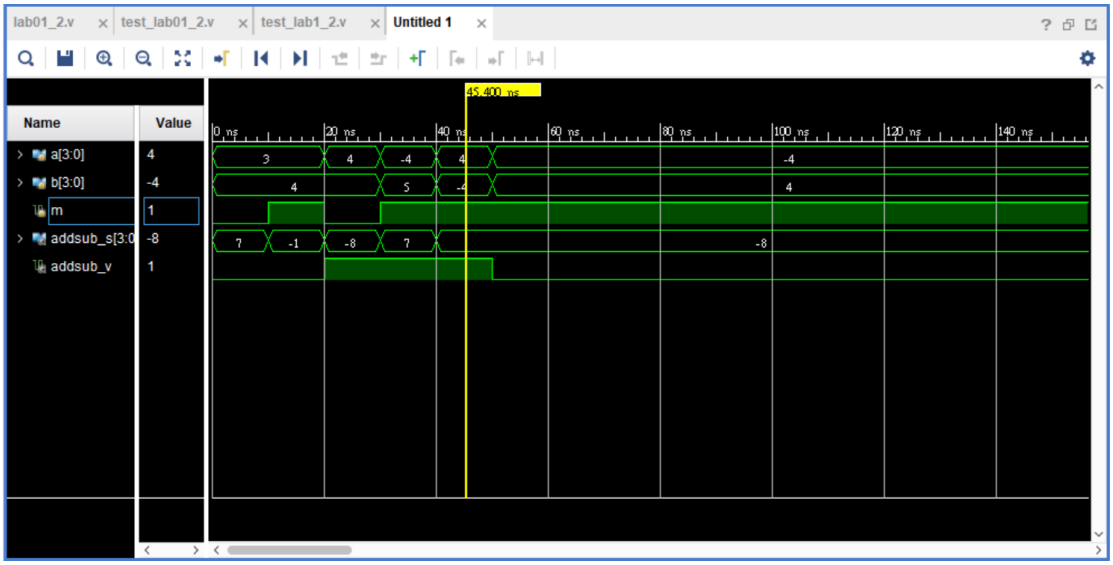
c. Code

(請見附件程式碼檔案)

Discussion



▲這張是我另外做的 testbench，我窮舉了所有可能並檢查。



▲這張為老師提供的測試檔。

大致上皆符合預期的成果，若出現數學上的錯誤也會顯示 Overflow 溢位，運行後的結果也符合題目的要求。實作的過程因為有用到 for loop 迴圈，所以使用 always block。在做 Converter 時可以直接用運算子+1 來進行。這裡另外要注意的是 input 變數只能是 wire，所以我另外設一個變數 B 為暫存器 reg，才能在 always block 裡計算 b 的 2’ complement。

至於 4 bits FA 的部分，由於要判斷是否 Overflow，所以我得手動寫一個加法器，而不是直接用+運算子進行。但我後來有想到另一個寫法便是利用 if else 判斷，若 正+正=負 或 負+負=正 那就可以知道有 Overflow，這樣或許就可以省去建立 c 的空間。

另外一個困難的點就是在寫窮舉所有可能的 testbench。一開始想要用 for loop

做，但就得必須寫成巢狀。且出來的結果會看到多兩行輸出 i, j，且 i 和 j 都會變成陣列的形式。經查訊後發現 for loop 在 Verilog 裡是一次展開並一次執行，除非用 task 包裝後在 initial block 呼叫便可達到像 C 語言的效果。
所以我後來改用 repeat 的形式，並用 if else 判斷當 b 是 1111 時，a 加 1 進位，而 b 重新設成 0。值得注意的是，延遲時間必須要設小一點，否則會跑不完。

Conclusion

在經過這實驗後，我學到了

- always block 的使用
- For loop, if else 的使用
- repeat 的應用
- integer

經過上網查詢後，我發現 if else, for loop 這些流程控制都必須寫在 always block 裡，而 always 裡的變數都要設定成 reg 暫存器。但 a, b 這種變數陣列可以統一操作便令我感到新奇，和 C 有很大的不同。還有在用 for loop 時，要另外建立一個 integer。這個 integer 就跟 C 的 int 很像，可以設定有號無號、bit 長度。
最後，在經過上網查詢後才發現連 for loop、if else 都要加 begin, end，這樣讓我可以輕鬆地看懂某些指令運行的範圍，也大大增加了可讀性。

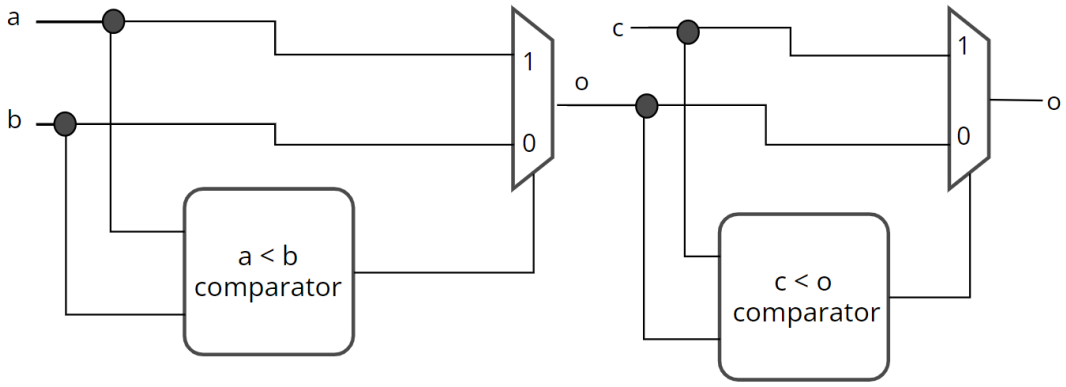
References

https://ithelp.ithome.com.tw/articles/10192465
在討論裡提到的 for loop 和 C 有很大的不同，但用 task 包裝後就方便許多，verilog 也不會在每次執行迴圈時重新宣告 i。
https://www.chipverify.com/systemverilog/systemverilog-data-types-integer-byte
這篇提到了更多關於 int 的操作，有分 short , long int，且變數可以用 bit_wise() 函式來查詢變數所佔的記憶體空間。

3 (Bonus) For three 3-bit signed numbers a (a2a1a0), b (b2b1b0), and c (c2c1c0), build a logic circuit to output o(o2o1o0) as the smallest number and use a given testbench for verification.

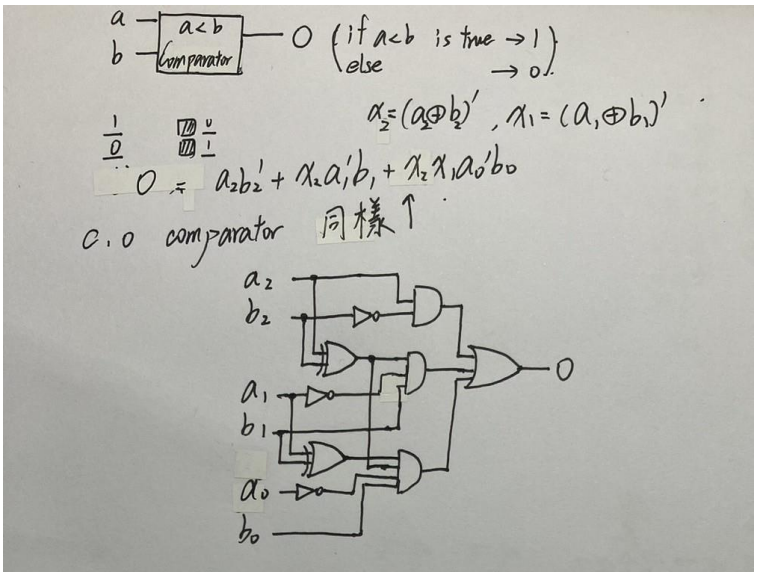
Design Specification

IO 輸出入設定
輸入: [2:0]a, [2:0]b, [2:0]c 三個大小皆為 3 bits(皆為最左邊為 MSB)
輸出: [2:0]o 大小為 3 bits(最左邊為 MSB)



Design Implementation

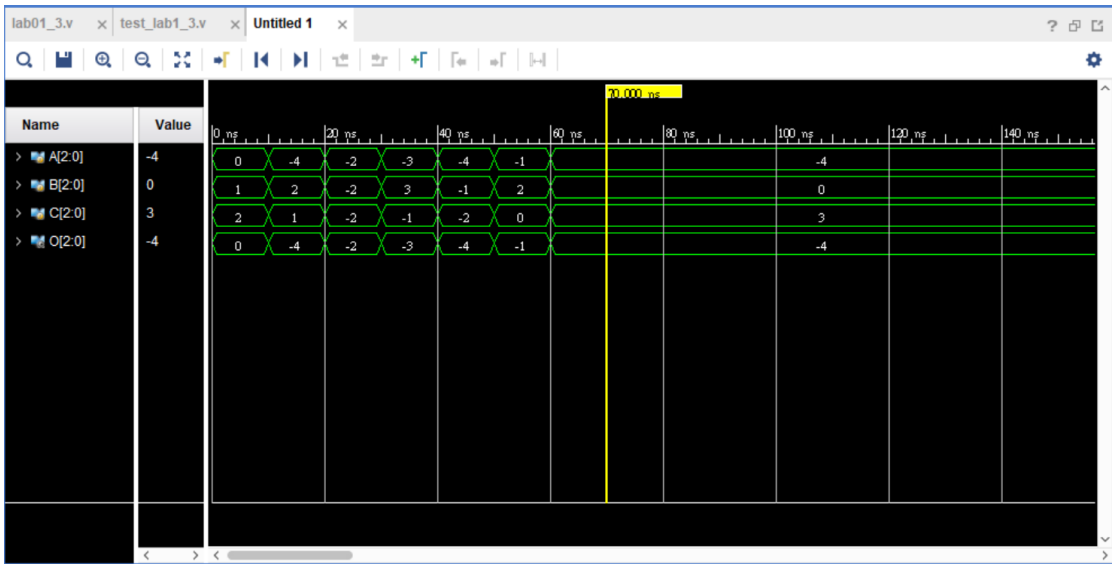
a. Logic diagram
如右圖，但實際操作時直接以 < 運算子判斷。



b. Logic func.
判斷小於
$O = a_2b_2' + x_2a_1'b_1 + x_2x_1a_0'b_0$
$x_2 = a_2 \oplus b_2 \quad x_1 = a_1 \oplus b_1$

c. Code(請見附件程式檔)

Discussion



▲老師提供的 testbench

程式跑完符合預期的成果。能正確地在正數負數之間判斷大小關係。實作的過程我是直接用 if else 加上 < 的運算子做判斷。一開始我寫成-4*a[2]+a[1:0]的形式，但經過 display 檢查後發現數字出來是一個 INT_MAX。所以我懷疑是因為乘以-4 的關係，但 a 預設為 unsigned int，所以碰到負號變成溢位。後來直接在宣告 input 時就也同時宣告 a 為 signed，這樣一來便能輕鬆地進行大小判斷。

Conclusion

經過實驗後，我學到了

- 注意宣告變數時的正負號運算

經過上網查詢後，我發現若一開始宣告 unsigned，但在賦值給了一個負號，例如: -4'd12，對於有號數來說不會影響，但對無號數就有。-4'd12 是直接賦值 10100 給變數，就如同老師上課講的，這樣的賦值是直接從 bit 設定。

但正負號的變數在進行同樣地加減運算時可能就會影響(參見 References)。

這次的 bonus 感覺算簡單，透過 signed 的宣告就可以避免掉寫一大段 Logic func 和宣告一堆變數。希望可以學到更多好讓我們可以做實驗更輕鬆一點。

References

https://stackoverflow.com/questions/12399991/how-does-verilog-behave-with-negative-numbers

這裡提到了關於正負號數在進行除法時的差異。留言也提到更多相關的問題，例如: /2 和 >> 1 在實際運算上的不同。