CO_lab 4

程式說明

1. CPU.v:

- clk, rst, SW 為 input 訊號
- CA~CG, AN 為 output 訊號
    - 由 SW 來控制欲顯示的大小
    - CA~CG 來控制顯示 0~9 哪個數字
    - AN 來控制要亮哪一個七段顯示器
- 由於我 lab3 是把 input 放在 `MEM.DM[0]` 的位置，所以我把 SW 的訊號傳到 MEMORY.v
- 因為要用七段顯示器顯示答案，故多寫一個 module top，把最後算完的結果(`MEM.DM[1]`,`MEM.DM[2]`)丟過去
- 為了方便表示，故把傳到 top 的 CA 訊號改名成 A，依此類推

2. INSTRUCTION_FETCH.v:

- 為了能夠在收到 rst 訊號後就重置，故把原本在 testbench 中用 initial 去初始化的 0~125 行指令，改用 always 去實作
- 接收到**非 rst 訊號**時，則判斷 `PC[10:2]`是否有小於 125 行，是就讀取該行，以避免讀取到錯誤的位置

3. INSTRUCTION_DECODE.v:

- 因 lab3 寫法關係，在接收到 rst 訊號後，
  把 `REG[0]` `REG[1]` `REG[2]` `REG[3]` `REG[4]` 分別設為 0, 1, 2, 3, 5，其餘則設為 0
- 為了把剩下的 REG file 設為 0，故多宣告一個 `reg [31:0] i` ，去跑 for 迴圈

4. EXECUTION.v:

- 此塊與 lab3 相同

5. MEMORY.v:

- 接收到 rst 訊號時，把 `DM[0]` 設為 SW 值，但因 `DM` 為 32 bit，input 只用到 13 bit (0~12 個 SW)，所以高位補 0
- 為了把剩下的 DM file 設為 0，故多宣告一個 `reg [31:0] i` ，去跑 for 迴圈
- 為了避免 Combinational Loop，故把原本 lab3 中 XM_MemWrite 寫到同一個 always block 裡面

6. top.v:

- input 訊號為 clk, rst, 跟從 `MEM.DM` 來的 result1, result2
- output 為 AN, A~G(為了方便表示，故把 CA 改成 A，依此類推)
- 因為使用 0~12 個 switch 當成 input，故宣告 13 bit 的 answer_number1, answer_number2
- 由 counter 跟 state 去控制陰陽極，意即控制要亮哪個顯示器跟要亮哪個數字

7. Nexys4_DDR.xdc:

- 把 Clock signal 改名成 clk (為方便更改 lab3 的內容)
- Switches 開啟 0~12 當作 input， `SW[15]` 則改名成 rst 訊號
- 因為需要讓左右兩邊都亮，所以陽極 AN 全開，陰極 C 全開來顯示 0~9 數字

顯示器控制

- 最左邊 `SW[15]` 為 rst 訊號，往上撥後，用右邊 13 個指撥 `SW[12:0]` 設定 input，設定完後，把 `SW[15]` 往下撥，即可顯示最相近兩質數，左邊是較小的質數，右邊則為較大的質數