Lab5_Team1_Report

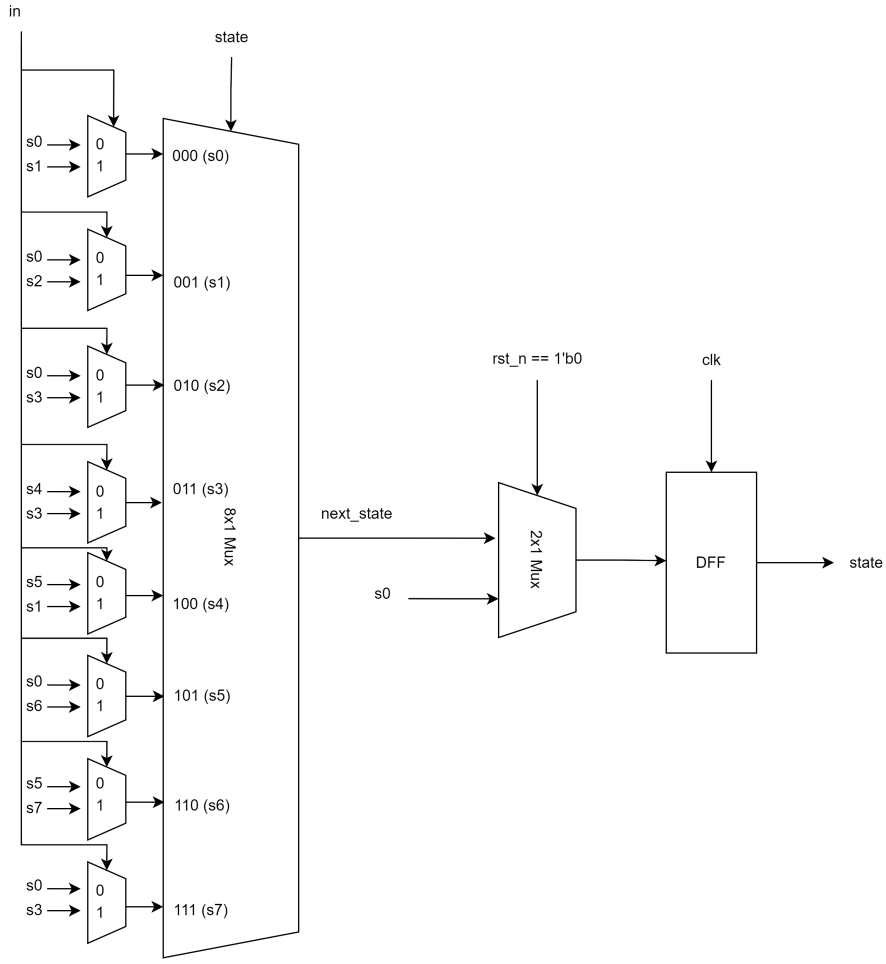
組員

107071016 施泳瑜
109062320 朱季葳

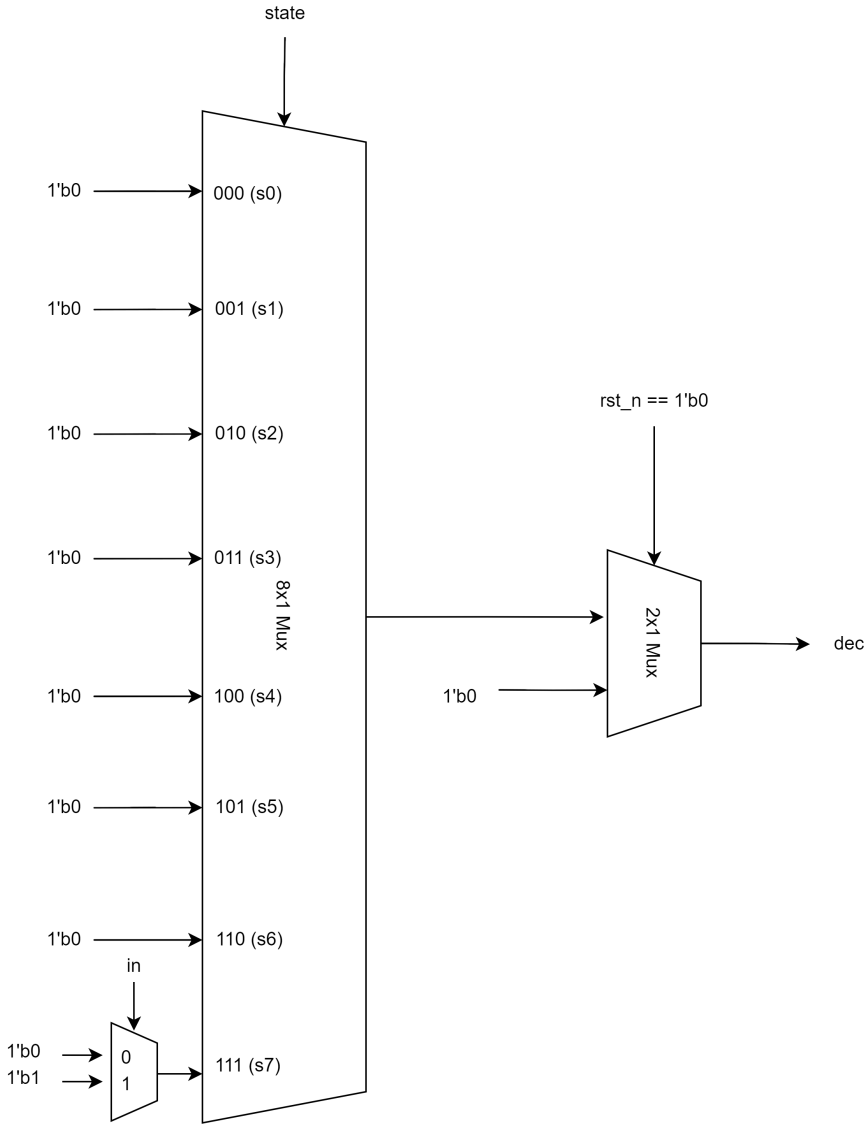
Advanced Question1

Drawing of the design of Verilog

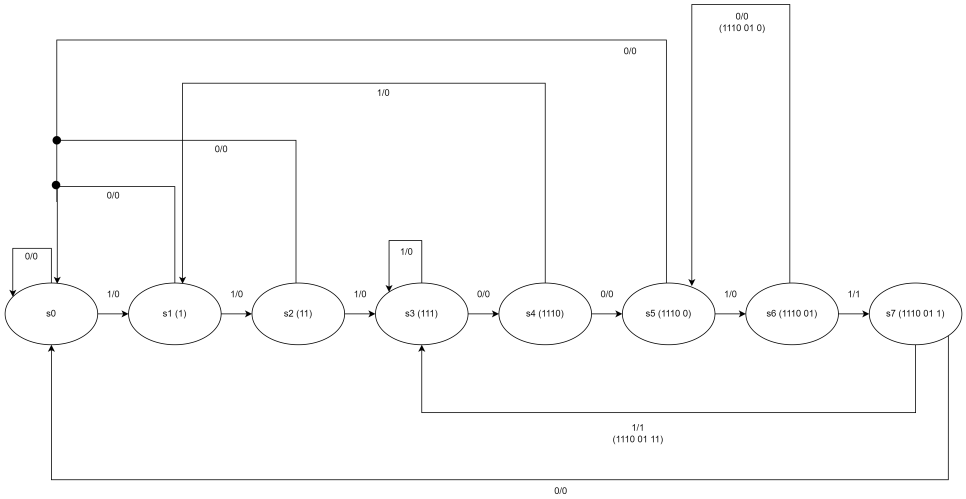
- state



- dec



State Diagram of the design of Verilog



Requirements

1. 利用 mealy machine 檢測 sliding window sequence

2. 目標: 1110(01)+11 in regular expression
3. (01) 至少要出現一次才合法
    - 111011 -> mismatch
    - 11100111 -> match
    - 1110010111 -> match
4. match 的狀況，設 dec = 1，否則 dec = 0

Design Explanation

1. s0 當作 initial state
2. 遇到 1 時進入下一個 state : s1
    - 否則回到 state : s0 (0 -> 視為 0)
3. 再遇到下一個 1 時進入下一個 state : s2 (11)
    - 否則回到 state : s0 (10 -> 視為 0)
4. 再遇到下一個 1 時進入下一個 state : s3 (111)
    - 否則回到 state : s0 (110 -> 視為 0)
5. 再遇到 0 時進入下一個 state : s4 (1110)
    - 否則回到 state : s3 (1111 -> 視為 111)
6. 再遇到下一個 0 時進入下一個 state : s5 (11100)
    - 否則回到 state : s1 (11101 -> 視為 1)
7. 再遇到下一個 1 時進入下一個 state : s6 (111001)
    - 否則回到 state : s0 (111000 -> 視為 0)
8. 再遇到下一個 1 時進入下一個 state : s7 (1110011)
    - 否則回到 state : s5 (1110010 -> 視為又再次進入 01 循環中)
9. 再遇到下一個 1 時達標 “11100111” ！ => 此時設定 dec = 1 & 再回到 state : s3
    - 否則回到 state : s0 (11100110 -> 視為 0)

Testbench Design & Result Explanation

TESTBENCH設計解釋

1. 模擬 ppt p.12 上半的 match case 情況
2. reset 檢驗
3. 後半模擬 ppt p.12 下半的 mismatch case 情況

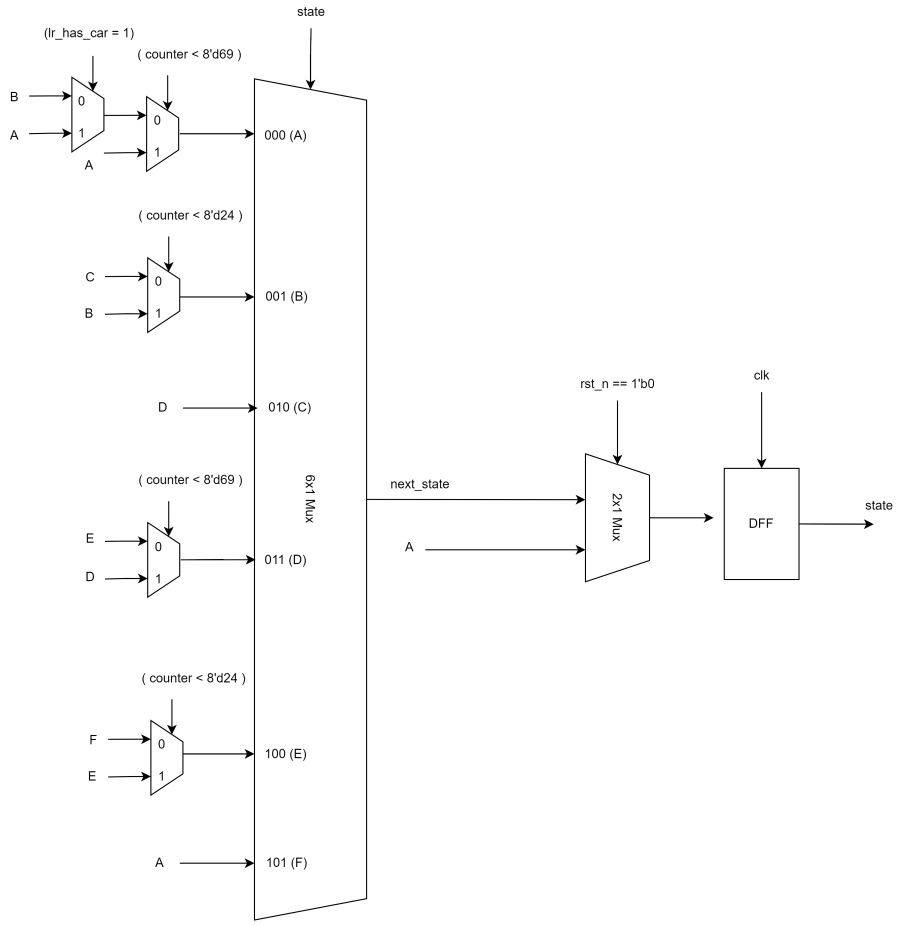
波形圖截圖

[圖一] 符合投影片的波形圖，並也在 reset 檢驗時 output dec = 0 (且維持 state A狀態)，結果與我們所預期的結果相吻合。

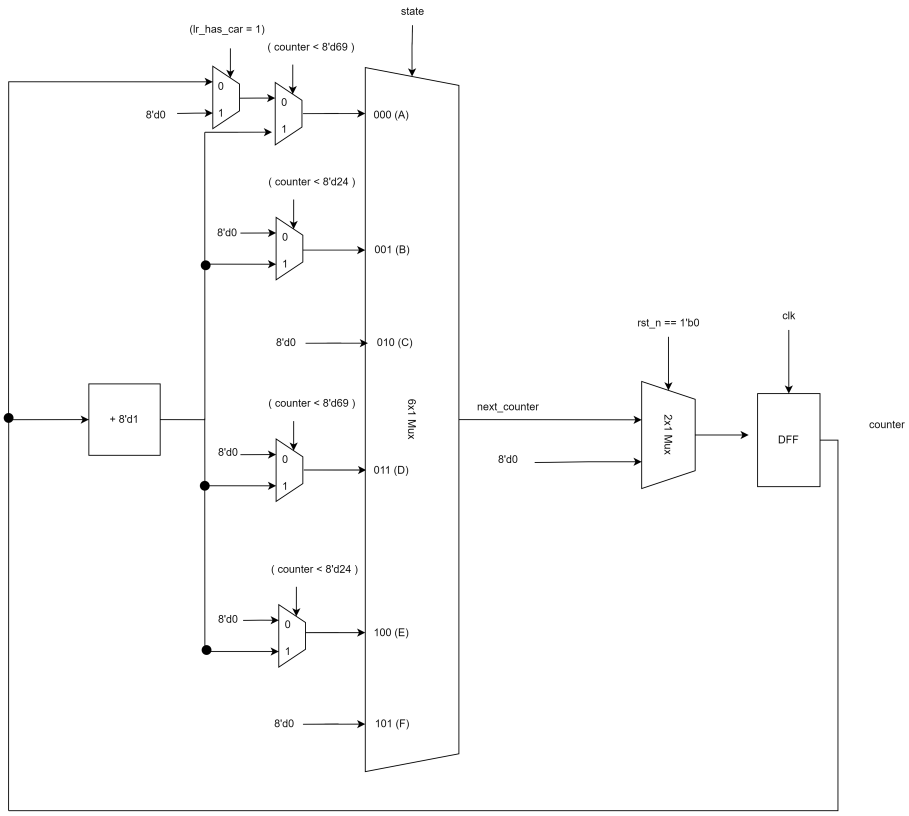
Advanced Question2

Drawing of the design of Verilog

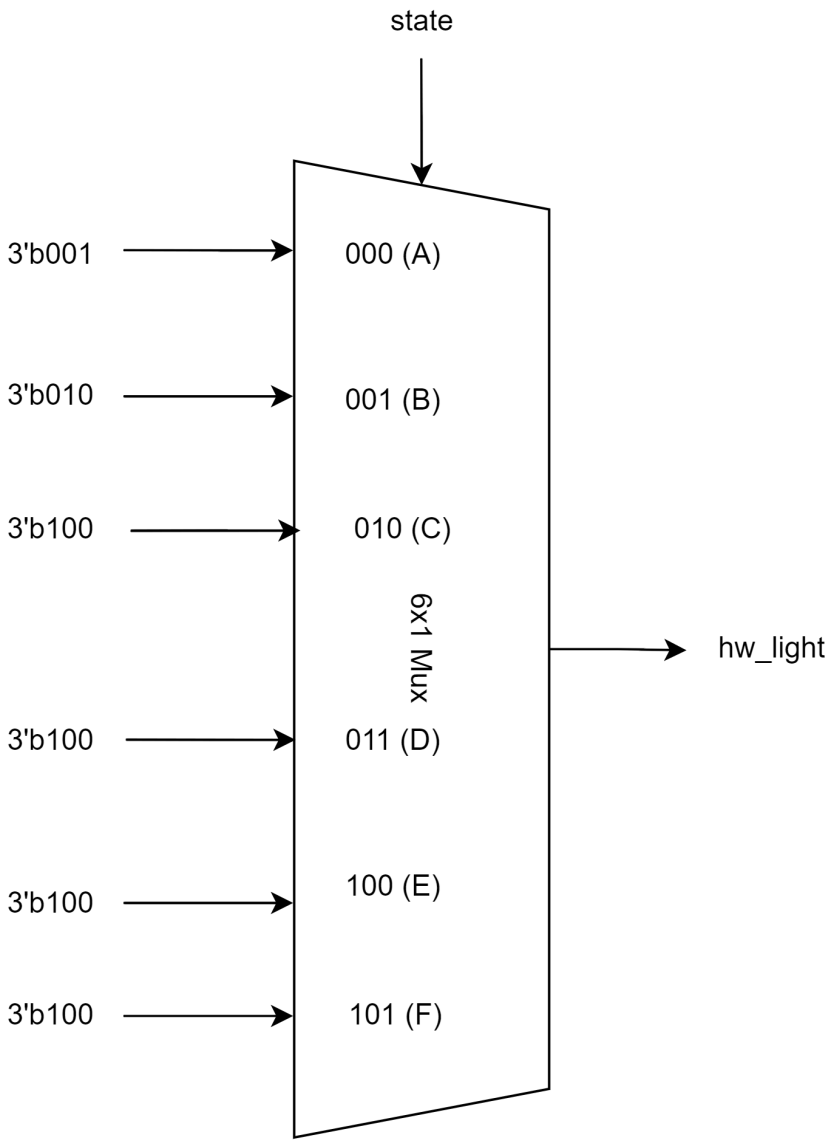
- state



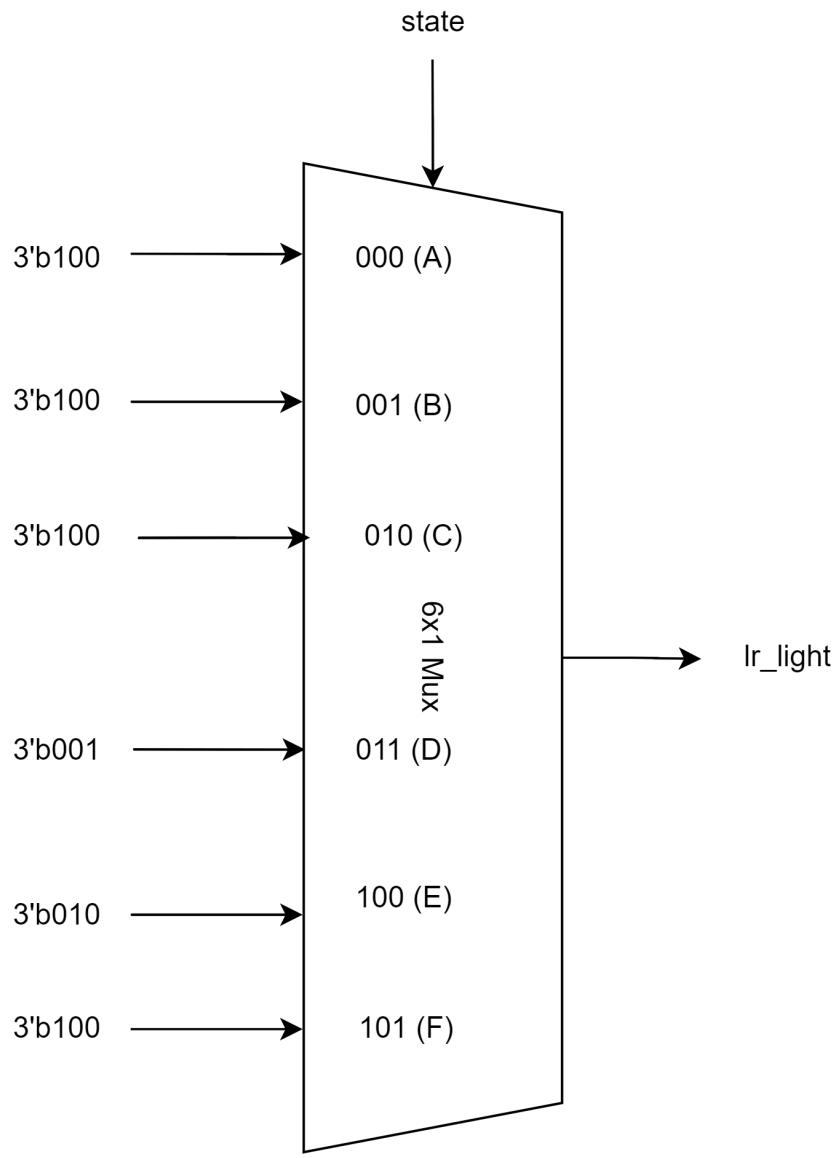
- counter



- hw_light



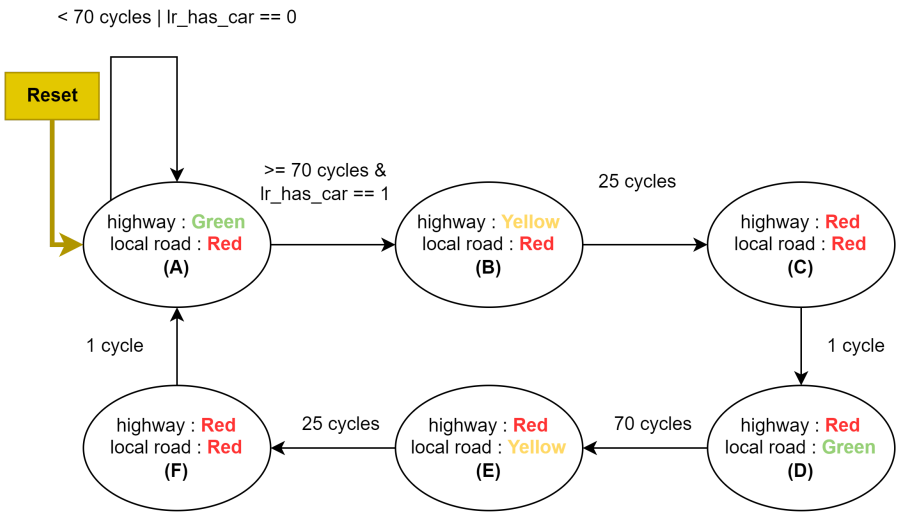
- lr_light



State Diagram of the design of Verilog

- PPT p.14 給得 state transition diagram 少了兩個箭頭 :
  - reset = 1 : 維持 state A
  - (高速公路的綠燈尚未超過 70 cycles) or (lr_has_car = 0) : 維持 state A
- 其他如同 PPT 圖所運作
  - state A 開始 : 高速公路為綠燈 / 地方道路為紅燈
  - 高速公路的綠燈維持 70 cycles + 地方道路有車 => 高速公路轉黃燈 (state B)
  - 高速公路的黃燈維持 25 cycles => 高速公路轉紅燈 (state C)
  - 高速公路的紅燈維持 1 cycles => 地方道路轉綠燈 (state D)

- 地方道路的綠燈維持 70 cycles => 地方道路轉黃燈 (state E)
- 地方道路的黃燈維持 25 cycles => 地方道路轉紅燈 (state F)
- 地方道路的紅燈維持 1 cycles => 高速公路轉綠燈 (state A)



Requirements

完成 Traffic light controller 之 FSM：

- 6 個 states
  - HW 比 LR 具有更高的 priority
  - LR 偵測到有車之後，HW 轉紅 + LR 轉綠 (如 State transition diagram 所示)
  - LR 在維持一段時間的 cycles 數之後，LR 轉紅 + HW 轉綠 (如 State transition diagram 所示)

Design Explanation

1. reset 時，狀態回到 state A (HW = Green & LR = Red)，且 counter = 0。

2. 使用一個 counter 去計算高速公路 / 地方道路的亮燈的 cycle 數。

3. 因高速公路綠燈轉黃燈有兩個判斷條件(cycle 數 + lr_has_car)，因此在 state A 確認是否已經滿足 70 cycles 之下，再去分支：
   - 是 (已經走過 70 cycles)：
     - lr_has_car = 0 : 維持 state A

        - lr_has_car = 1 : 高速公路轉黃燈 (state B)
    - 否 (尚未走過 70 cycles) : 維持 state A

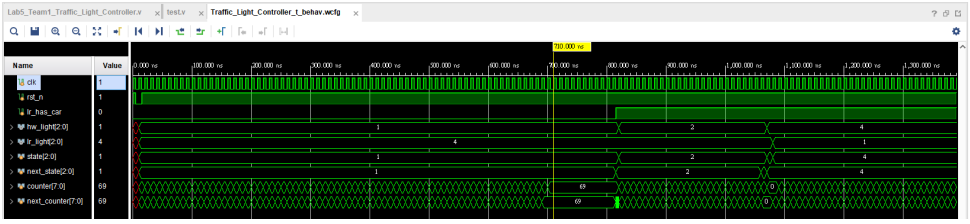
4. state B、C、D、E、F，在滿足 cycle 數之後，設立進入下個狀態。
    - state B --[25 cycles]-> state C
    - state C --[1 cycles]-> state D
    - state D --[70 cycles]-> state E
    - state E --[25 cycles]-> state F
    - state F --[1 cycles]-> state A

Testbench Design & Result Explanation

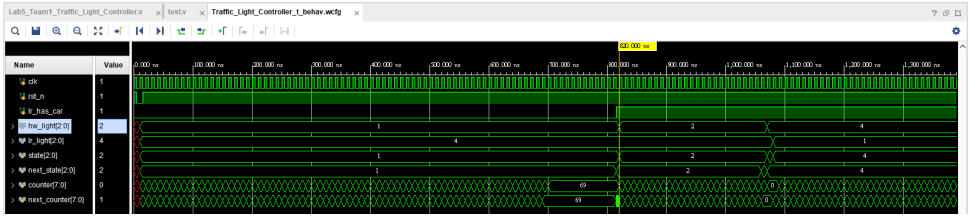
TESTBENCH設計解釋

1. 測試 state A -> B 時，至滿足 70 cycles 的條件，lr_has_car == 1 不滿足
2. 補上 lr_has_car == 1，讓 state -> B 轉換成功
3. state B -> C -> D -> E -> F -> A 在符合 cycles 的條件下正常轉換
4. Again！在 2820 ns 時測試 reset，檢查狀況是否符合預期 ~
5. state A -> B -> C 在符合 cycles 以及特殊條件下可如同上面一般正常轉換

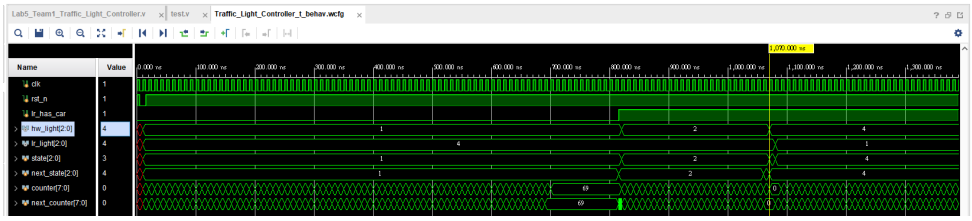
波形圖截圖



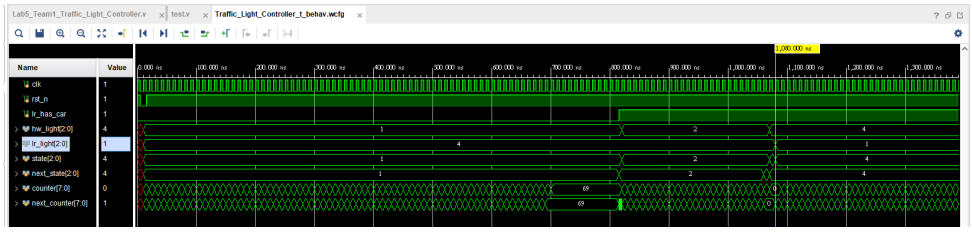
[圖一] 710 ns 時，過了 70 個 cycles，但此時 ls_has_car = 0，所以 HW 仍保持「綠」燈。



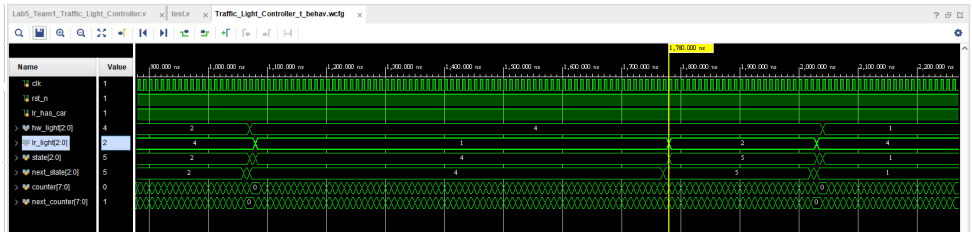
[圖二] 直到 820 ns 時，ls_has_car = 1 出現之後，HW 才轉為「黃」燈。



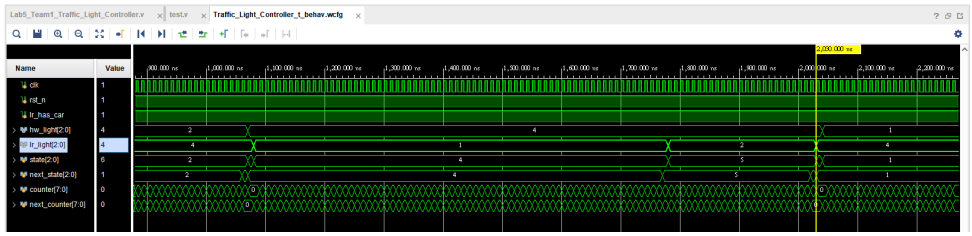
[圖三] 再過 25 cycles (1070ns時)，HW 轉為「紅」燈。



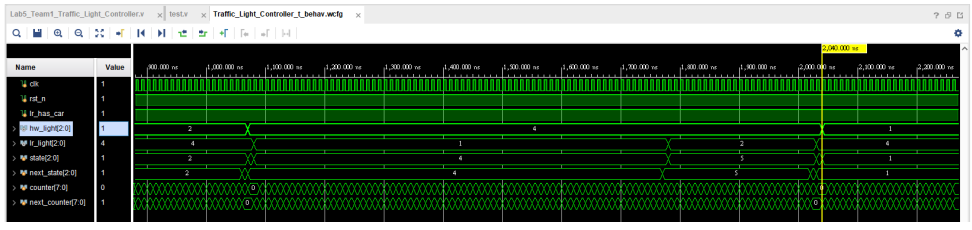
[圖四] 再過 1 cycles (1080ns時)，LR 轉為「綠」燈。



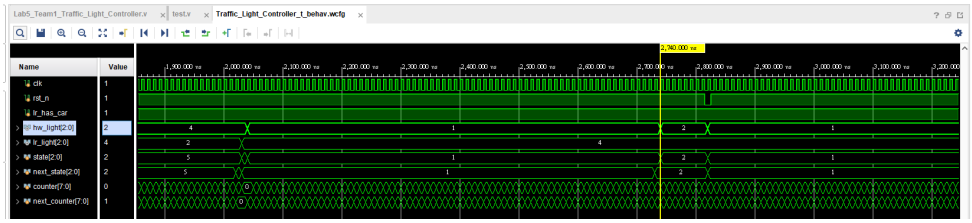
[圖五] 再過 70 cycles (1780ns時)，LR 轉為「黃」燈。



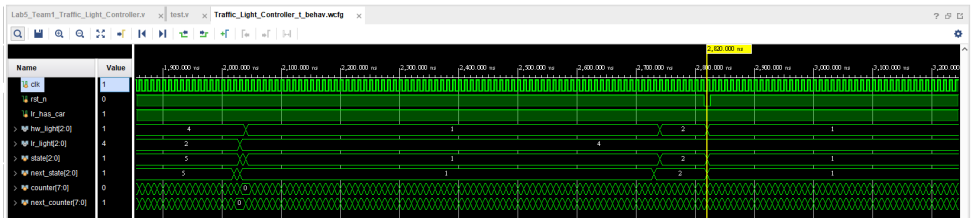
[圖六] 再過 25 cycles (2030ns時)，LR 轉為「紅」燈。



[圖七] 再過 1 cycles (2040ns時)，HW 轉為「綠」燈。 (又再回到了state A)



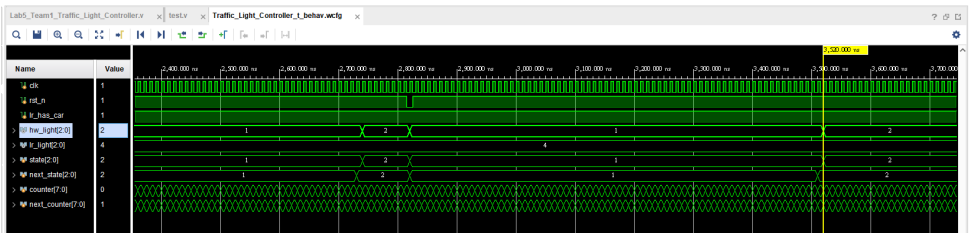
[圖八] 2740 ns 時，又過了 70 個 cycles，且此時 ls_has_car = 1，HW 轉為「黃」燈。



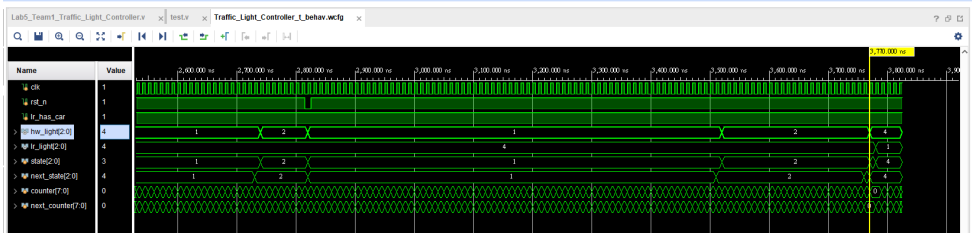
[圖九] Again~~~ 在2820 ns 時，測試 reset :

- counter = 0
- state = A (1)
- HW = Green (3'b001)
- LR = Red (3'b100)

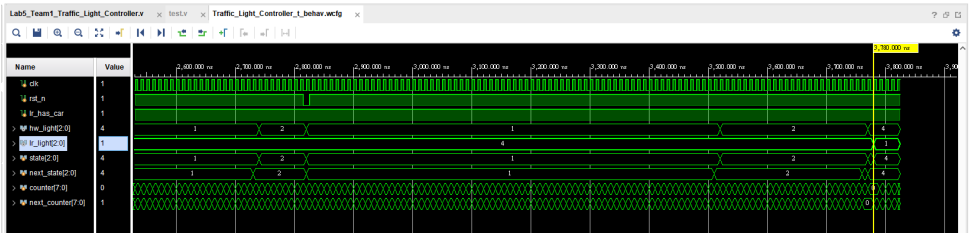
以上四者都檢查 ok，合法！



[圖十] 3520 ns 時，又過了 70 個 cycles，且此時 ls_has_car = 1，HW 轉為「黃」燈。



[圖十一] 再過 25 cycles (3770ns時)，HW 轉為「紅」燈。

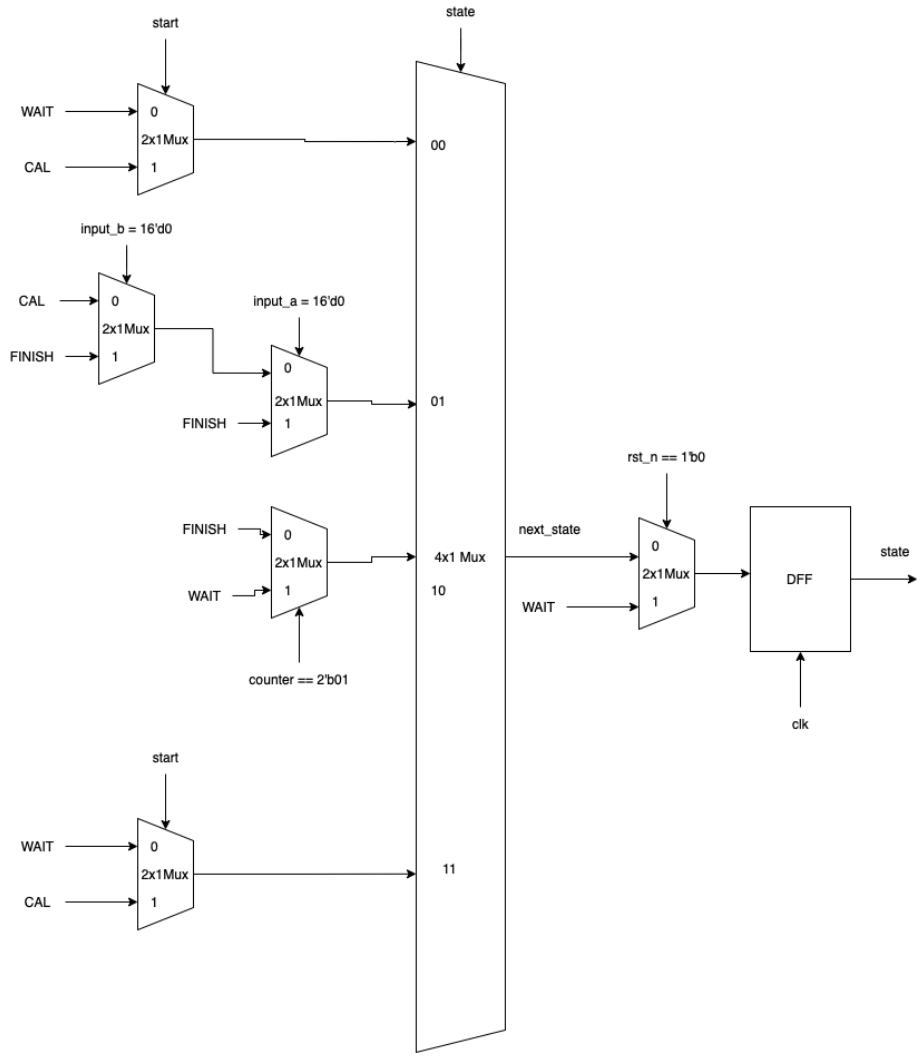


[圖十二] 再過 1 cycles (1080ns時)，LR 轉為「綠」燈。

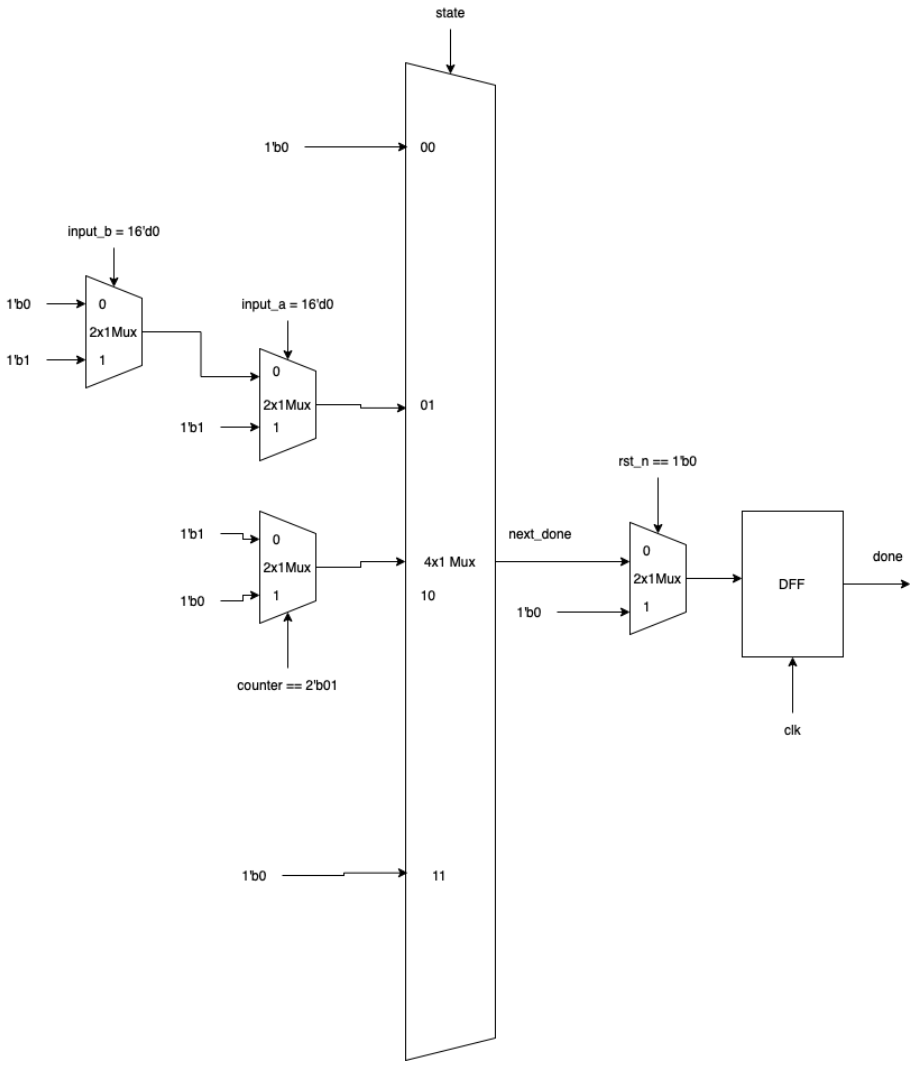
Advanced Question3

Drawing of the design of Verilog

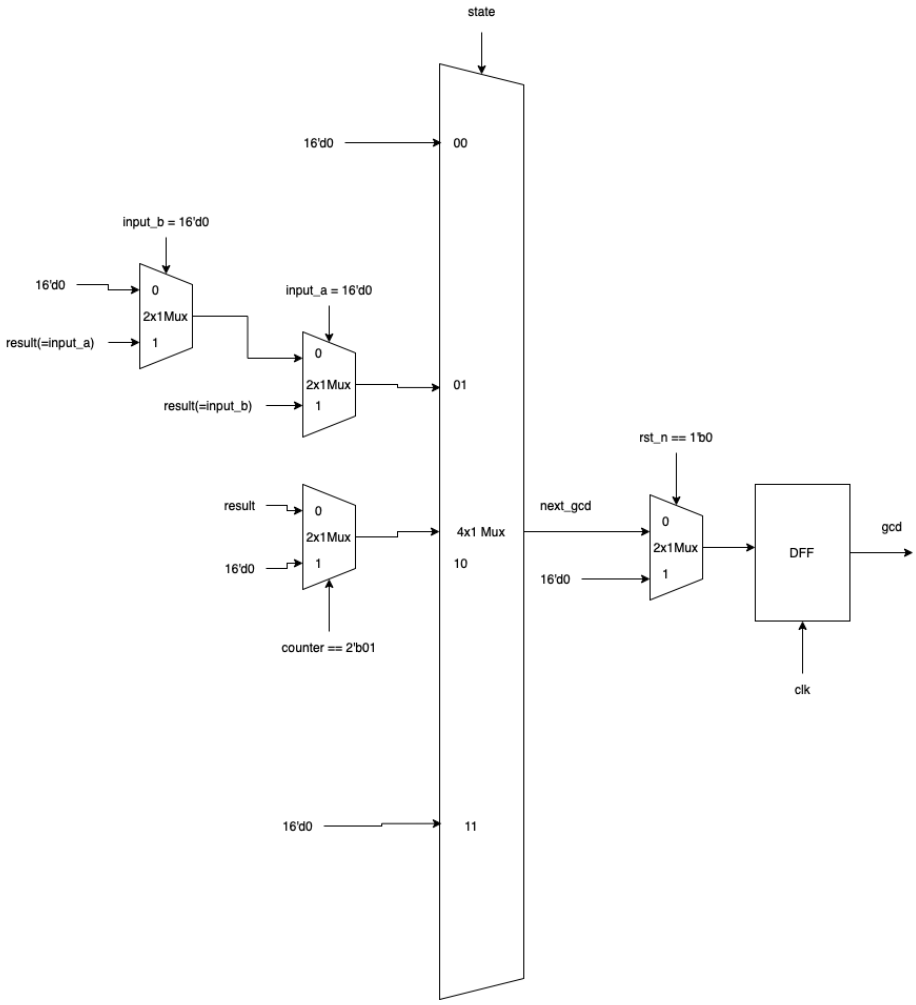
- state



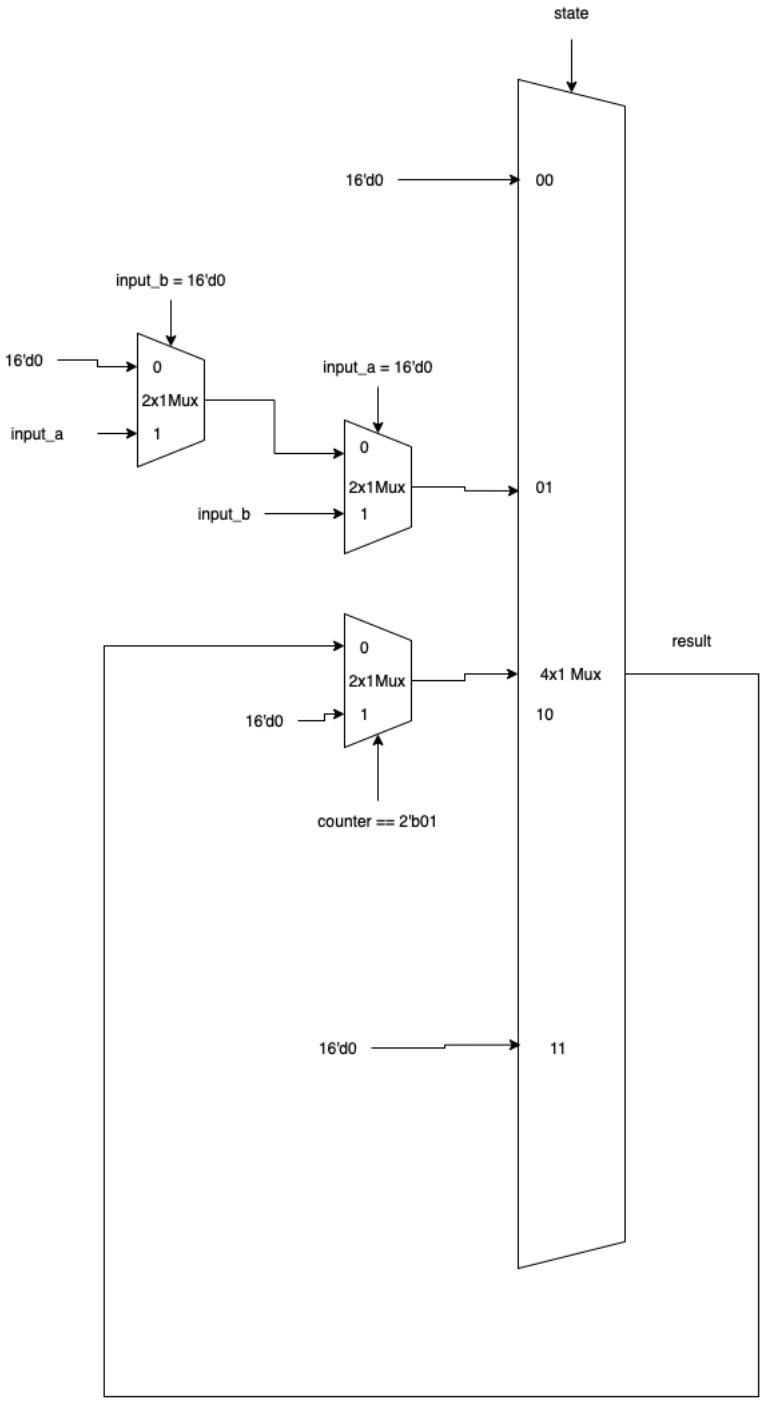
- done


state
1'b0
00
input_b = 16'd0
1'b0
0
2x1Mux
1
1'b1
input_a = 16'd0
0
2x1Mux
1
01
1'b1
rst_n == 1'b0
1'b1
0
2x1Mux
1
4x1 Mux
10
1'b0
counter == 2'b01
next_done
0
2x1Mux
1
1'b0
DFF
done
clk
1'b0
11

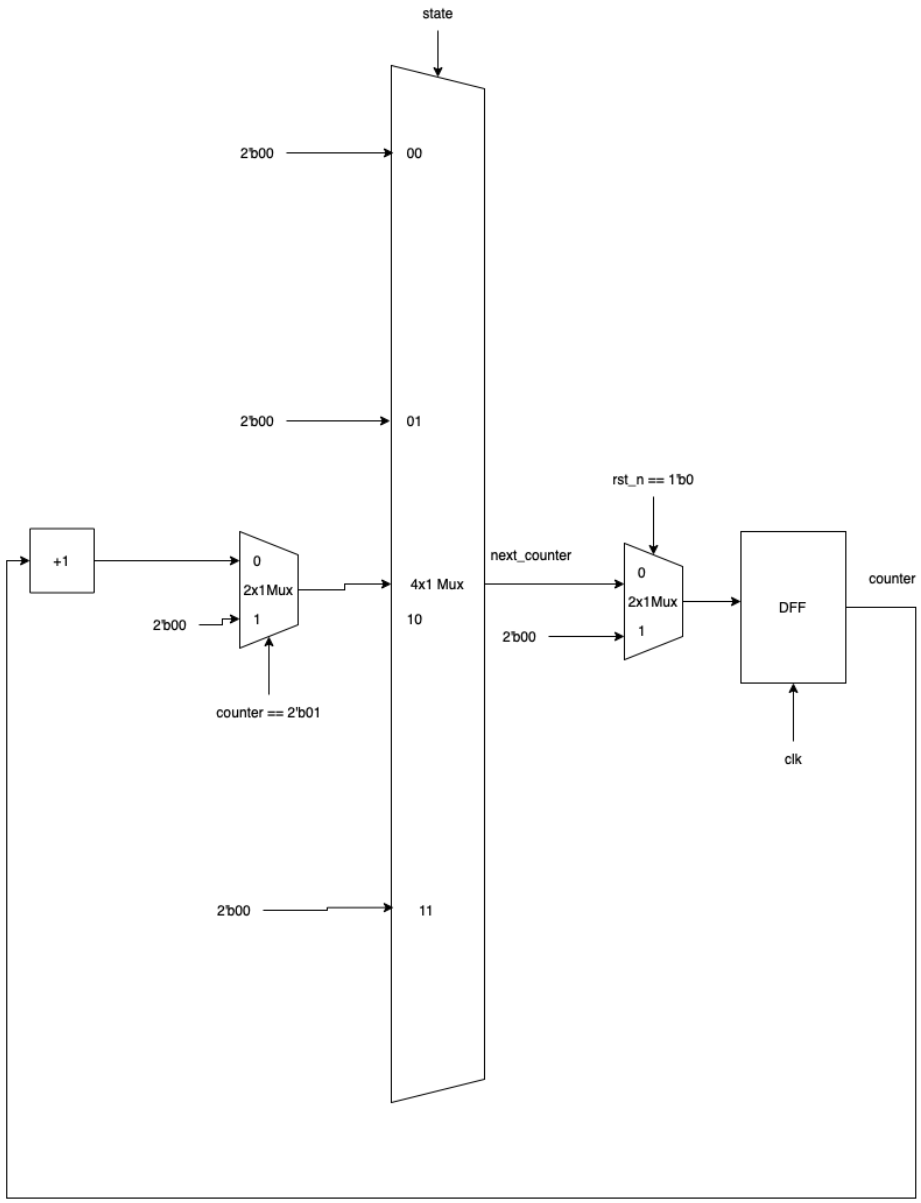
- gcd


state
16'd0
00
input_b = 16'd0
16'd0
0
2x1Mux
1
result(=input_a)
input_a = 16'd0
0
2x1Mux
1
01
result(=input_b)
rst_n == 1'b0
result
0
2x1Mux
1
4x1 Mux
10
16'd0
counter == 2'b01
next_gcd
0
2x1Mux
1
16'd0
DFF
gcd
clk
16'd0
11

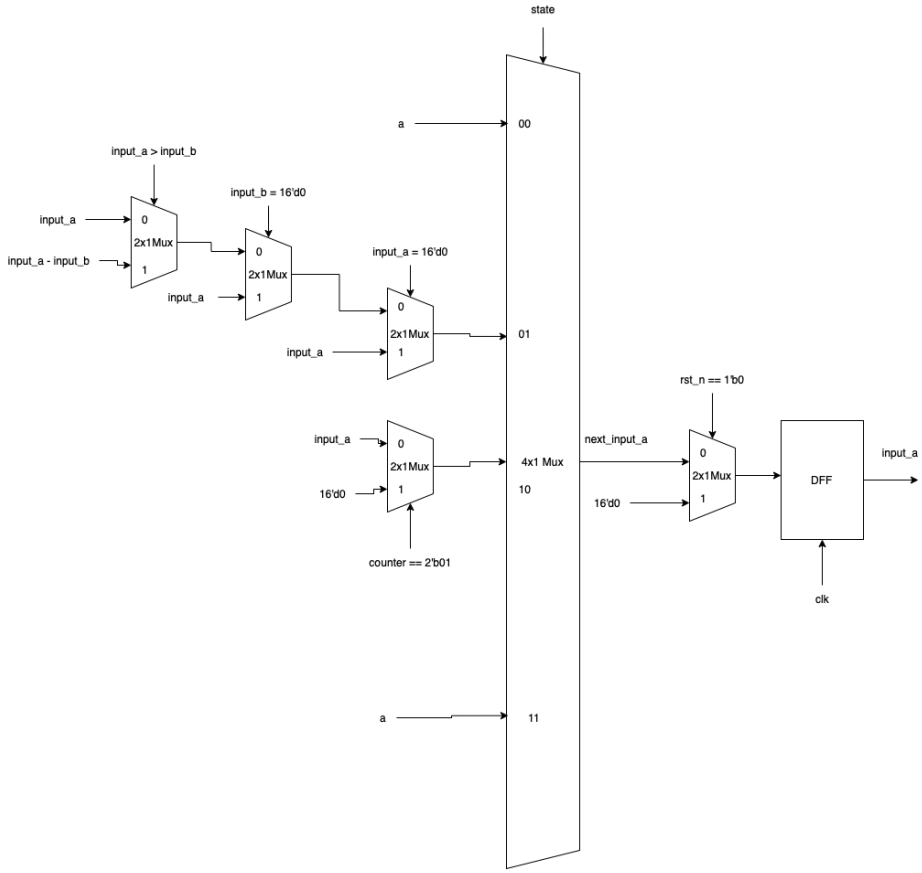
- result



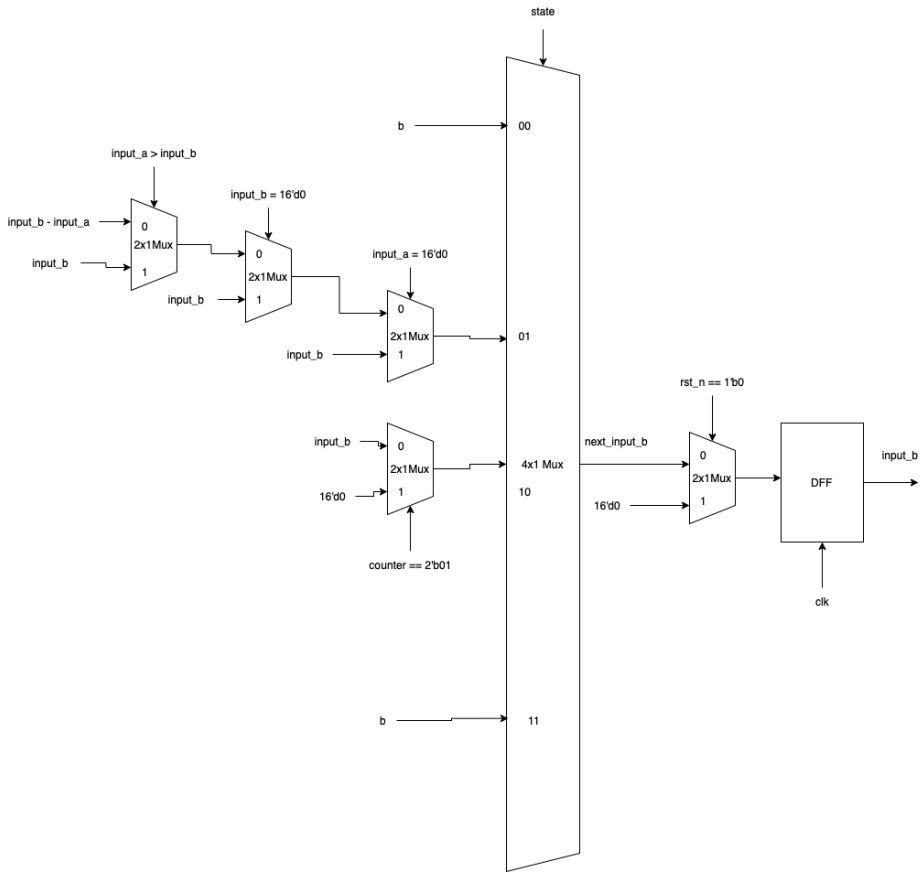
- counter



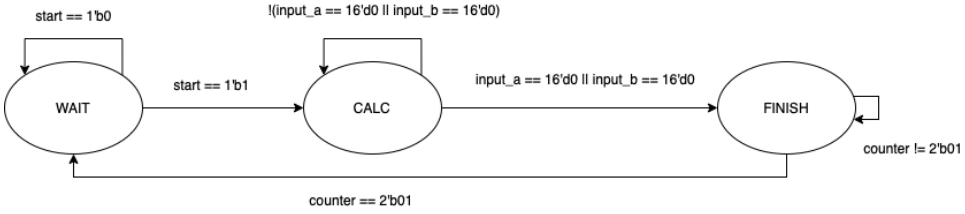
- input_a



- input_b



State Diagram of the design of Verilog



Requirements

1. 用輾轉相除法實作gcd
2. 有三個state分別為wait, calc, finish
3. 在state為wait, calc的時候,gcd = 16'd0, done = 1'b0，而在state為finish的時候gcd為計算結果,done = 1'b1
4. 在wait state時，當start = 1'b1, record the value of a and b, and change state to calc
5. 在calc state時，每一個cycle計算一次輾轉相除法的步驟，直到算出答案為止(a = 0 or b = 0)
6. 在finish state時，要維持前面所計算出來的結果2 cycle才能回到wait

Design Explanation

在我們的設計中，可以分為sequential以及combinational的部分，而我們將在以下分別作說明，而我們首先要介紹一些額外設置的registers

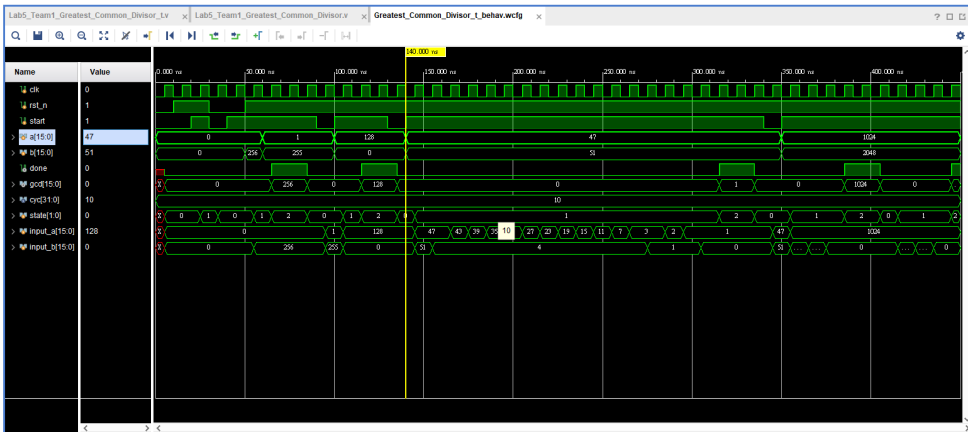
1. state:紀錄當下的state
2. counter:在state == finish的時候用來數cycle的數量
3. result:用來記錄最後的計算結果
4. input_a, input_b:在state為wait的時候接到a, b以記錄在start當下的input，否則a, b的值有可能會在計算過程中改變。
5. next_state, next_done, next_input_a, next_input_b:在positive edge trigger時用來更新值的registers

- sequential
  - reset
    - 把state設為wait
    - done, gcd設為0
    - counter設為0
    - input_a, input_b設為0
  - !reset

        - 把next_state的值傳入state
        - done, gcd, counter, input_a, input_b同理
- combinational
    - state == WAIT
        - 當start = 1時把next_state update 為CALC
        - fetch a, b的值
        - 其餘的result, next_done, next_gcd, next_counter皆update為0
    - state == CALC
        - 終止條件：input_a == 16'd0 || input_b == 16'd0
            - for input_a == 16'd0, result = input_b
            - for input_b == 16'd0, result = input_a
            - update next_state to FINISH
            - update done to 1'b1
            - update gcd to result
        - 輾轉相減的過程
            - input_a > input_b:把input_a的值扣除input_b作為update input_a的值
            - input_a <= input_b:把input_b的值扣除input_a作為update input_b的值
            - 下一個cycle仍要繼續做計算直到符合終止條件
            - gcd和done都update為0
    - state == FINISH
        - 用counter計算現在經過了幾個cycle
        - counter == 2'b01:代表已經是第二個cycle(從0開始計算)，所以下一個cycle要進入wait的狀態，將output在下一個cycle時update為wait state的狀態。
        - counter == 2'b00:代表是第一個cycle，所以下一個cycle維持一樣的state，所有output都照舊，並將counter+2'b01

Testbench Design & Result Explanation

波形圖截圖



TESTBENCH設計解釋

我們的測資如下列所示

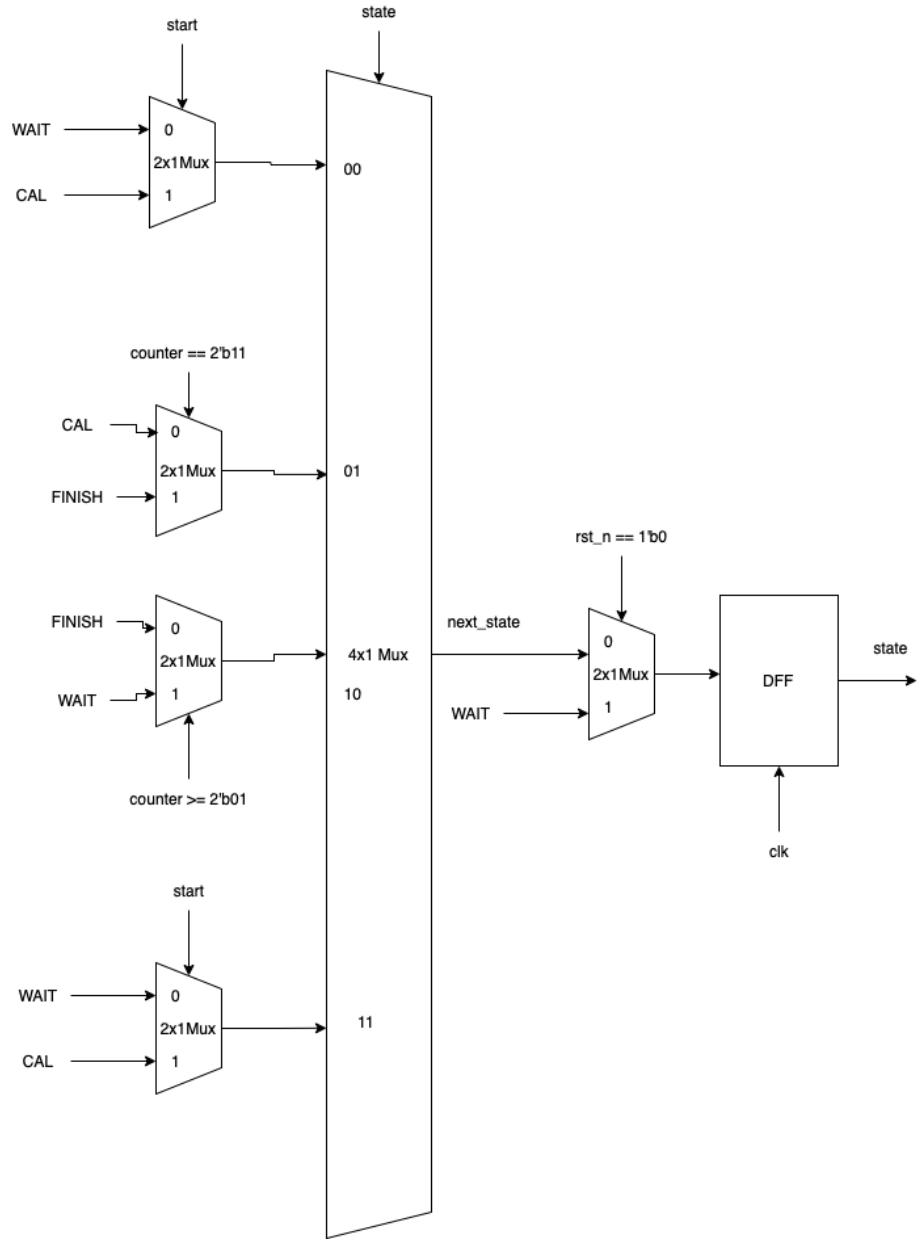
1. 在reset過後測試(a = 0, b = 256)來檢查input_a == 16'd0的終止條件是否奏效
2. 測試(a = 128, b = 0)來檢查input_b == 16'd0的終止條件是否奏效
3. 測試(a = 47, b = 51)來檢查兩個互質的數的最大公因數為1
4. 測試(a = 1024, b = 2048)來檢查其中一個是對方的因數的情形

由於窮舉沒有意義且我們只需要確認能不能正確的計算出最大公因數以及維持計算出的答案兩個cycle，所以我們確認完上面波形圖結果過後就確認設計無誤。

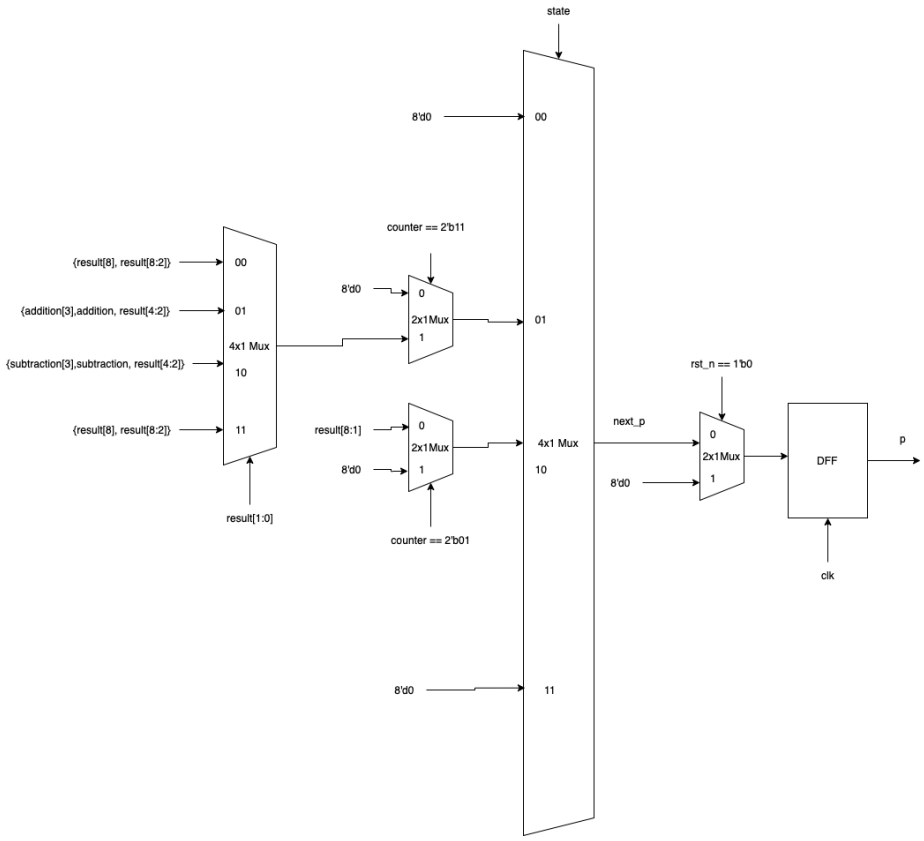
Advanced Question4

Drawing of the design of Verilog

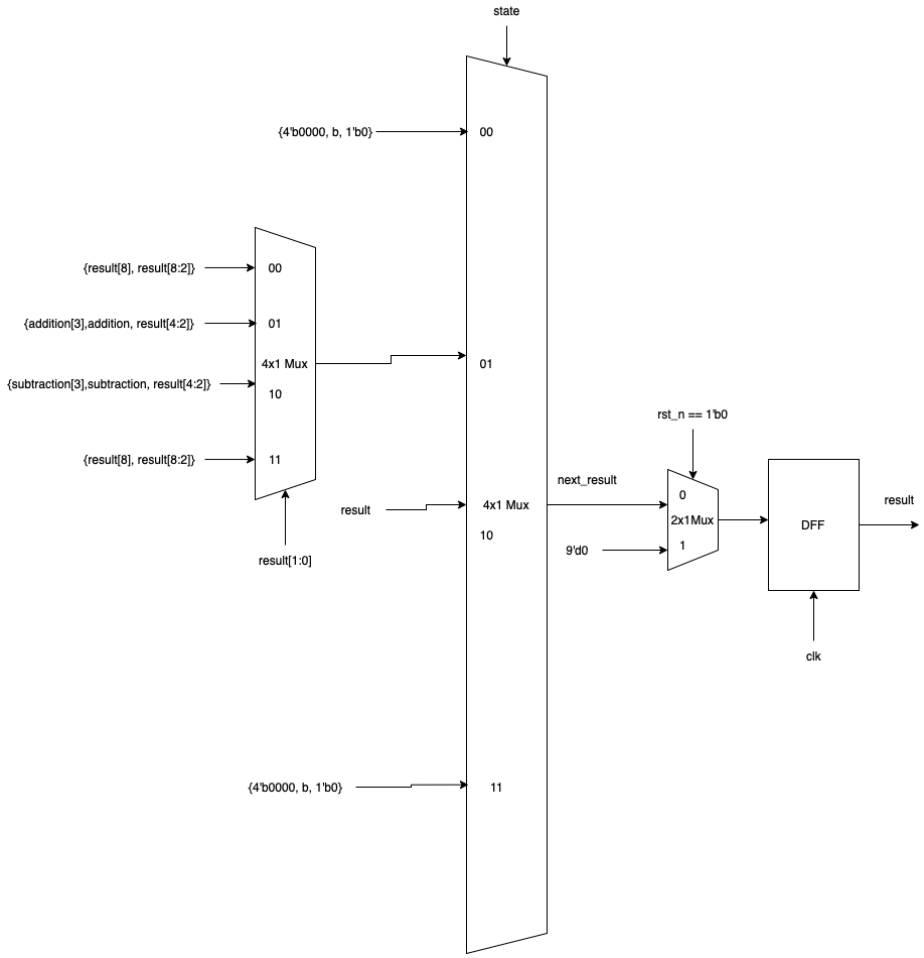
- state


start
state
WAIT
0
2x1Mux
1
CAL
00
counter == 2'b11
CAL
0
2x1Mux
1
FINISH
01
rst_n == 1'b0
FINISH
0
2x1Mux
1
WAIT
4x1 Mux
10
next_state
0
2x1Mux
1
WAIT
DFF
state
counter >= 2'b01
clk
start
WAIT
0
2x1Mux
1
CAL
11

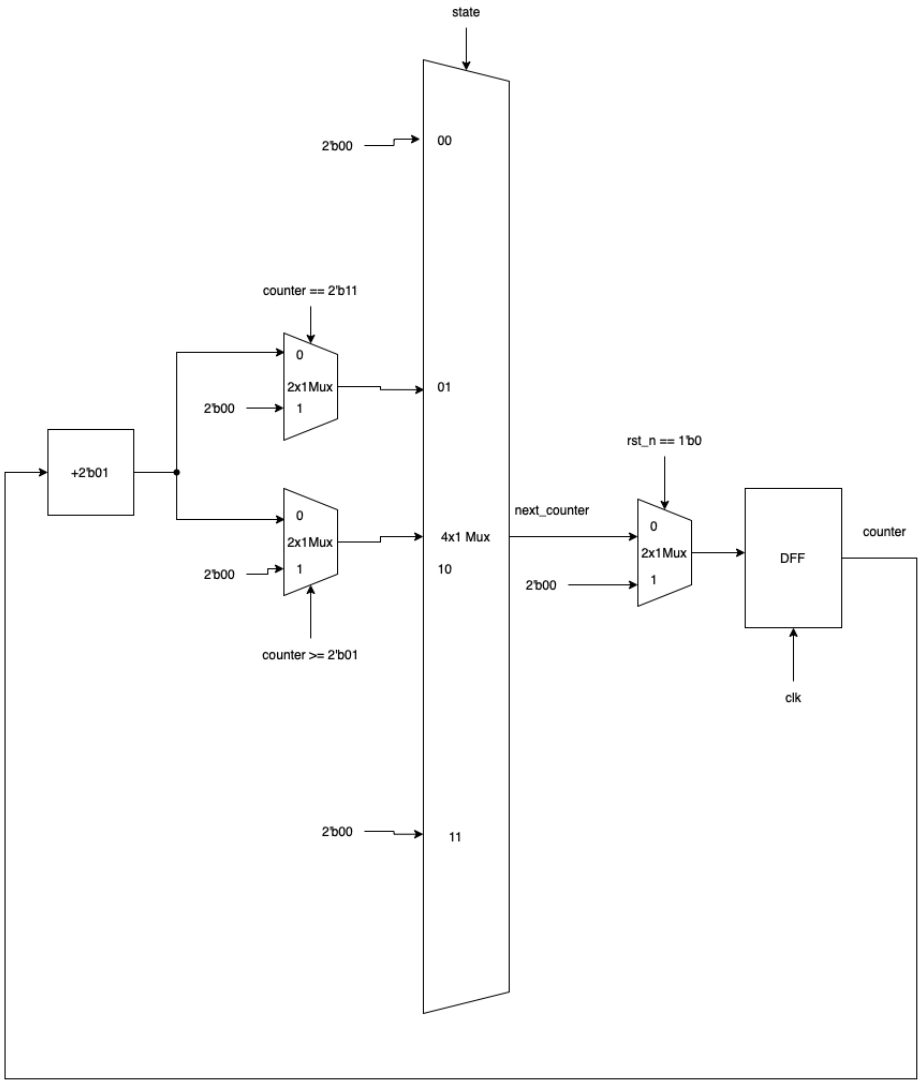
- p



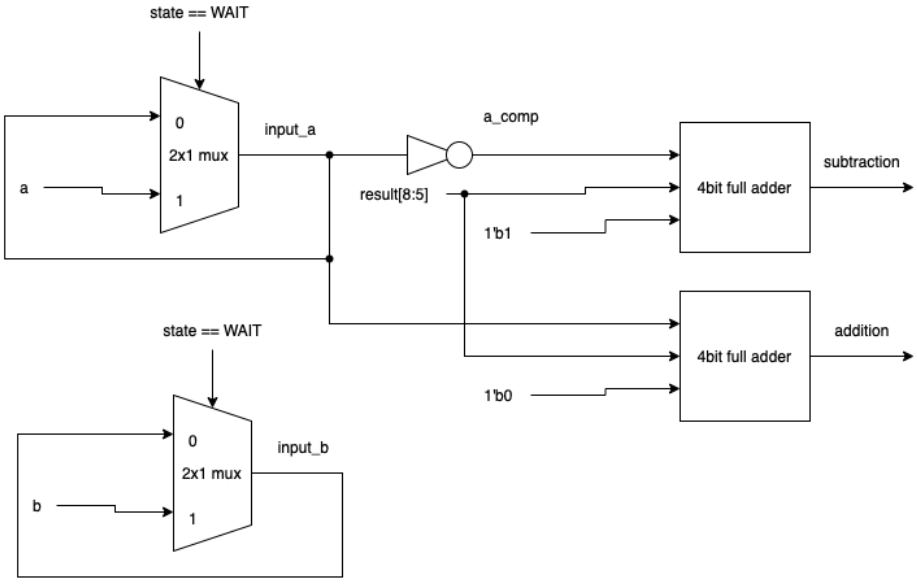
- result



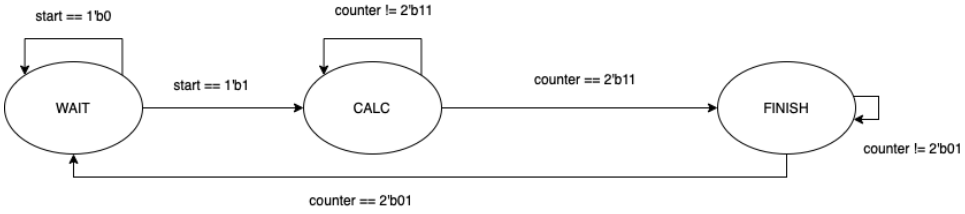
- counter



- input_a, input_b, addition, subtraction



State Diagram of the design of Verilog



Requirements

1. 實作一個Booth Multiplier
2. 有三個state分別為wait, calc, finish
3. 在state為wait, calc的時候,p = 8'd0，而在state為finish的時候p為計算結果
4. 在wait state時，當start = 1'b1, record the value of a and b, and change state to calc
5. 在calc state時，每一個cycle依據booth algorithm來做計算，持續四個cycle並得到最終答案
6. 在finish state時，要維持前面所計算出來的結果2 cycle才能回到wait

Design Explanation

在我們的設計中，可以分為sequential以及combinational的部分，而我們將在以下分別作說明，而我們首先要介紹一些額外設置的registers

1. state:紀錄當下的state
2. counter:在state == finish的時候用來數cycle的數量
3. result:用來記錄最後的計算結果
4. input_a, input_b:在state為wait的時候接到a, b以記錄在start當下的input，否則a, b的值有可能會在計算過程中改變。
5. next_state, next_done, next_input_a, next_input_b:在positive edge trigger時用來更新值的registers

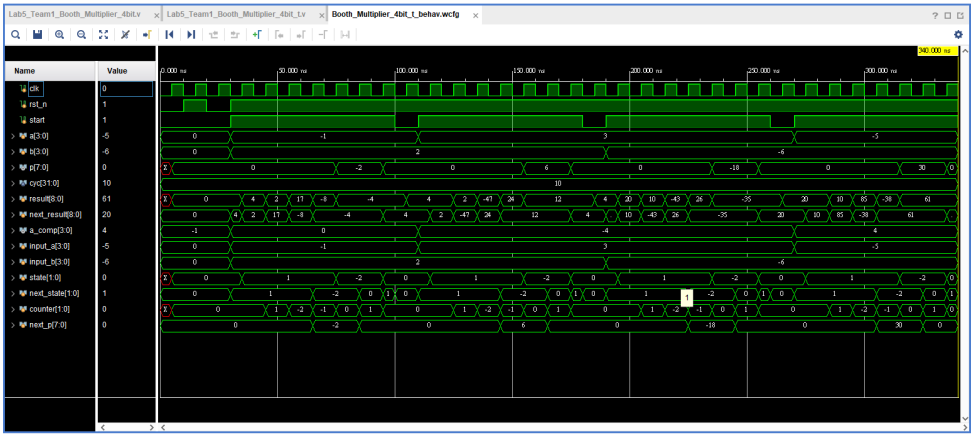
- sequential
  - reset
    - 把state設為wait
    - p設為0
    - counter, result設為0
  - !reset
    - 把next_state的值傳入state

        - p, counter, result同理

- combinational
    - state == WAIT
        - 當start = 1時把next_state update 為CALC
        - fetch a, b的值
        - 依據booth algorithm把result initialize為{4'b0000, b, 1'b0}
        - 其餘的next_p, next_counter皆update為0
    - state == CALC
        - 用counter計算現在經過了幾個cycle
            - counter == 2'b11:代表已經是第四個cycle，所以下一個cycle要進入wait的狀態，將output依據result[1:0]的值來進行最後一次的計算並在正緣觸發時update p(用左邊數來的8個bit)。
            - counter < 2'b11:代表尚未完成計算，所以下一個cycle維持一樣的state，所有output都照舊，並將counter+2'b01
        - 實作booth algorithm
            - result[1:0] == 2'b00, result[1:0] == 2'b11: shift right result by 1 bit
            - result[1:0] == 2'b01:(忽略overflow) 將result左半邊的4個bit加上a(用adder實作) 並進行right shift result by 1 bit
            - result[1:0] == 2'b10:(忽略overflow) 將result左半邊的4個bit減掉a(用adder實作) 並進行right shift result by 1 bit
    - state == FINISH
        - 用counter計算現在經過了幾個cycle
        - counter == 2'b01:代表已經是第二個cycle，所以下一個cycle要進入wait的狀態，將output在下一個cycle時update為wait state的狀態。
        - counter == 2'b00:代表是第一個cycle，所以下一個cycle維持一樣的state，所有output都照舊，並將counter+2'b01

Testbench Design & Result Explanation

波形圖截圖



TESTBENCH設計解釋

我們的tb測試了4種狀況如下

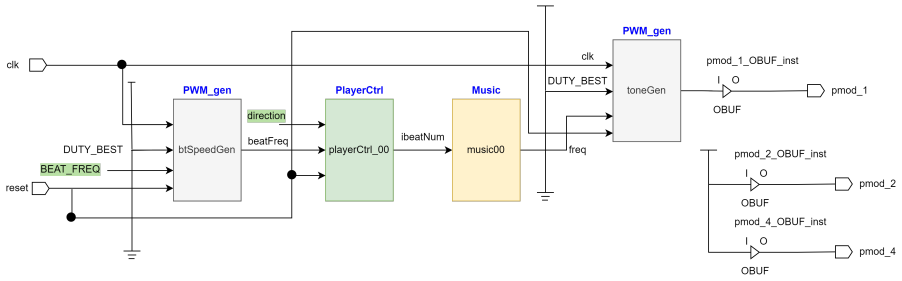
1. (a, b) = (負, 正):a = -1, b = 2
2. (a, b) = (正, 正):a = 3, b = 2
3. (a, b) = (正, 負):a = 3, b = -6
4. (a, b) = (負, 負):a = -5, b = -6

由於所有的情況大抵可依據a,b的正負來分類，所以我們測試上述四種情況來確認所得到的答案以及正負是否為正確，而根據波形圖所顯示的結果，確認我們的設計無誤。

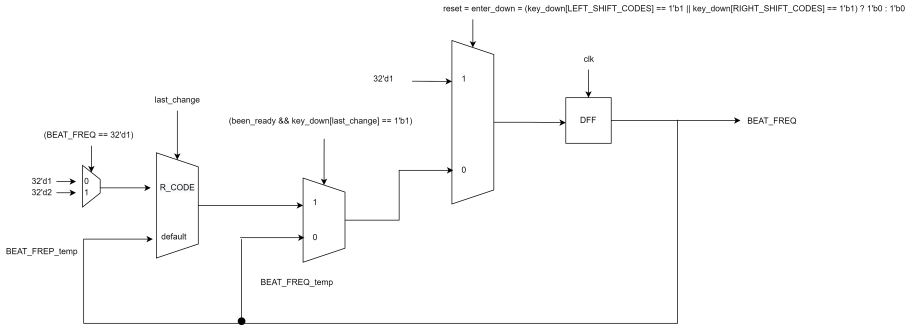
FPGA Demonstration 1

Drawing of the design of FPGA Demonstration 1

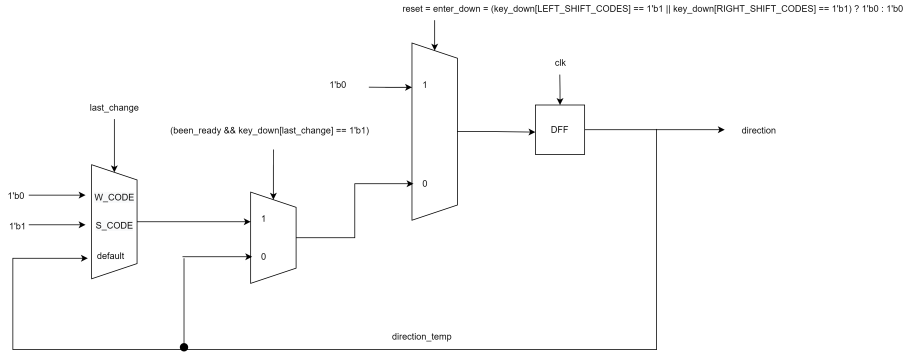
- Top module



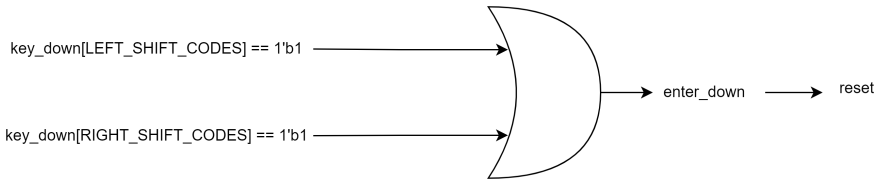
- BEAT_FREQ (Top module 綠底-1)



- direction (Top module 綠底-2)



- reset / enter_down (in BEAT_FREQ & direction module)



Design Explanation

[設計]

1. reset
    - direction = 0 : control the scale to ascend, ranging from C4 to high C8
    - BEAT_FREQ : 1 second per note

2. been_ready & key_down 合法

```
assign enter_down = (key_down[LEFT_SHIFT_CODES] == 1'b1 || key_down[RIGHT_SHIFT_CODES] == 1'b1) ? 1'b1 : 1'b0;
assign reset = enter_down;
always @(posedge clk) begin //manipulate beat and speed
    if(reset)begin
        BEAT_FREQ <= 32'd1;
        direction <= 1'b0;
    end

    else begin
        if(been_ready && key_down[last_change] == 1'b1)begin
```

    - 判斷是 w, s, r 何者被按下，依此做後續行為：
        - w : direction = 0
        - s : direction = 1

        - r : BEAT_FREQ
            - (1 seoncd -> 0.5 second)
            - (0.5 seoncd -> 1 second)
    - 都沒被按下 or 不合法： BEAT_FREQ、direction 不變

[延續使用 template 模組 + 解釋]

1. KeyboardDecoder Module：
   如同課堂中所介紹的演算法，使用 512-bit key_down 去追蹤誰被按下了，且 9-bit 的 last_change 紀錄是誰剛被按下，還有 1-bit key_valid 去看是否 active。

2. PWM_gen Module：

    - btSpeedGen：
      input 是 BEAT_FREQ (1 or 2) 和 Desired duty cycle (理解上是將它分成一半)，可產生 tempo of music。

    - ToneGen Module
      input 是 FREQ 和 Desired duty cycle ，如此一來，可以產生 repeated pulses with specified frequency (in HZ)，視為 tone。

      依此可以完成此題所規範的：
        - BEAT_FREQ = 1 則 period is 1 sec
        - BEAT_FREQ = 2 則 period is 0.5 sec

3. PlayerCtrl Module：
   輸出相對應的 ibeat 值，操縱節拍，可成功放出目標音樂。

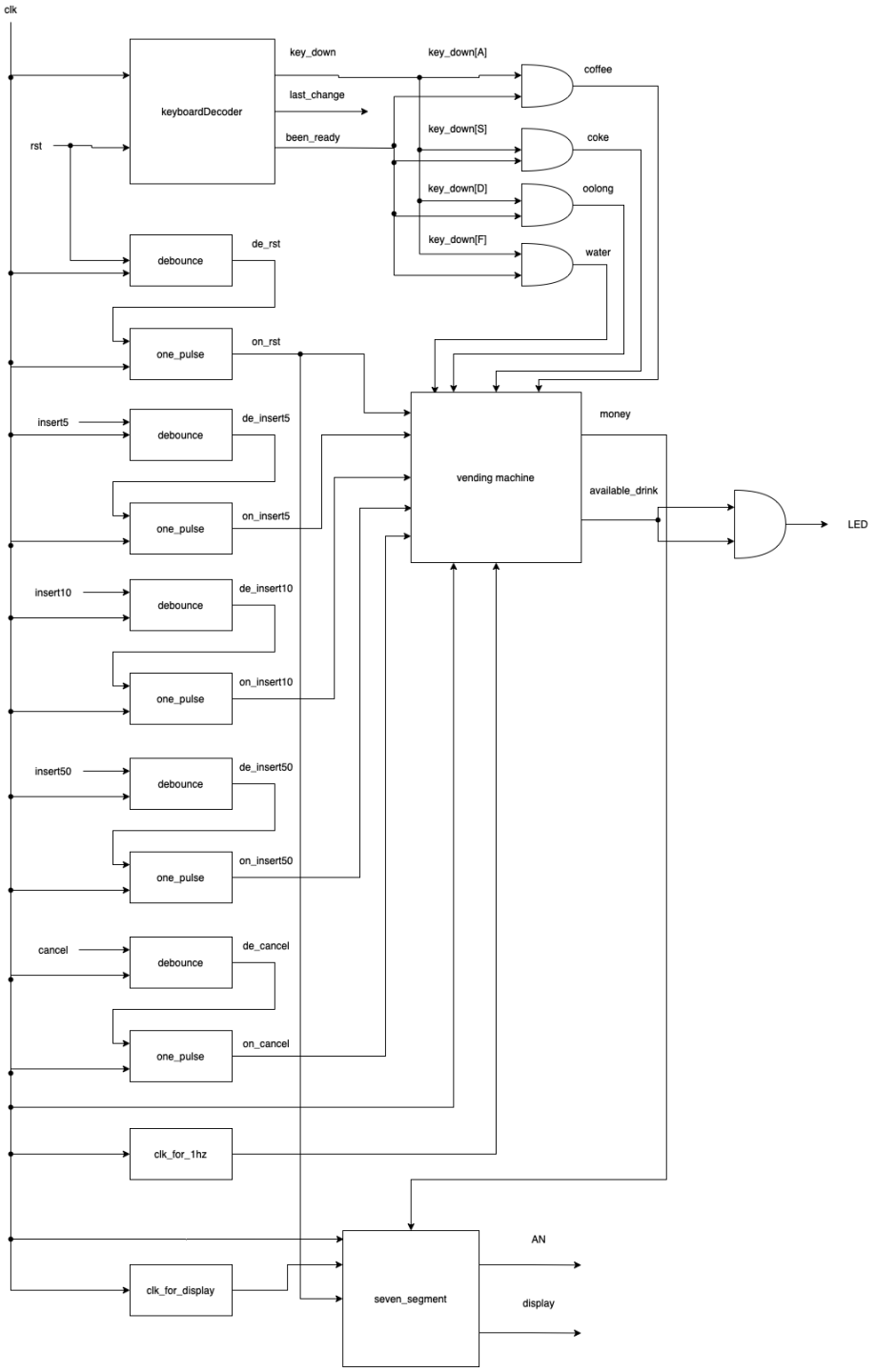
4. Music Module：
   如下，因已知要放出某個特定的音要去控制其 frequency，所以做音高撰寫，從 C5 = 523、D5 = 587、E5 = 659、F5 = 698、G5 = 784、A5 = 880、B5 = 988(Ref. 簡譜)，依樣畫葫蘆也刻出 C7~B7，再加上C8，以上是直接 define 的值；而 C6~B6 以及 C8~B8 則用 shift 的方法得到其值，如此一來填入對應的 frequency 可生成不一樣的頻率 / 音調。
    - C6~B6 ：double of C5~B5
    - C8~B8 ：double of C7~B7

```
`define NM8  32'd2093 //C7_freq
`define NM9  32'd2349 //D_freq
`define NM10 32'd2637 //E_freq
`define NM11 32'd2794 //F_freq
`define NM12 32'd3136 //G_freq
`define NM13 32'd3520 //A_freq
`define NM14 32'd3951 //B_freq

`define NM15 32'd4186 //C8_freq

`define NM0 32'd20000 //slience (over freq.)

always @(*) begin
    case (ibeatNum)
        8'd0 : tone = `NM1 >> 1; //C4_freq
        8'd1 : tone = `NM2 >> 1;
        8'd2 : tone = `NM3 >> 1;
        8'd3 : tone = `NM4 >> 1;
        8'd4 : tone = `NM5 >> 1;
        8'd5 : tone = `NM6 >> 1;
        8'd6 : tone = `NM7 >> 1;
```

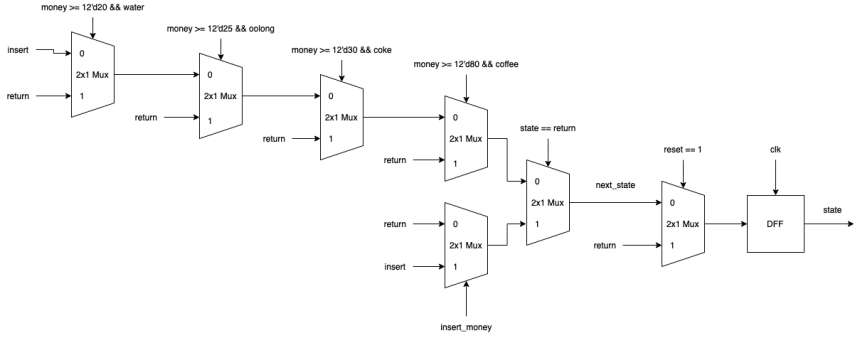
FPGA Demonstration 2

Drawing of the design of FPGA Demonstration 2

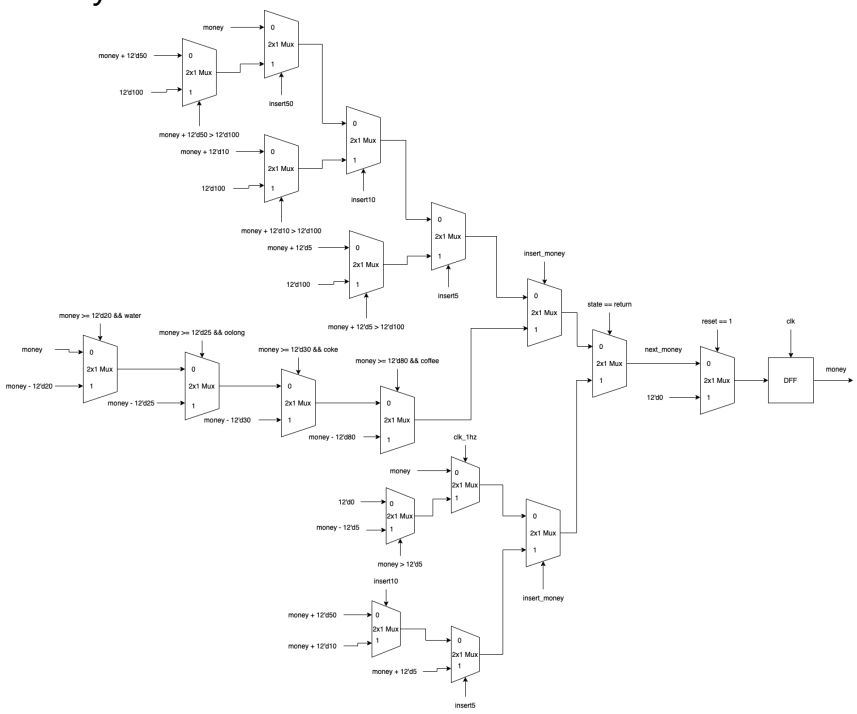
- Top module



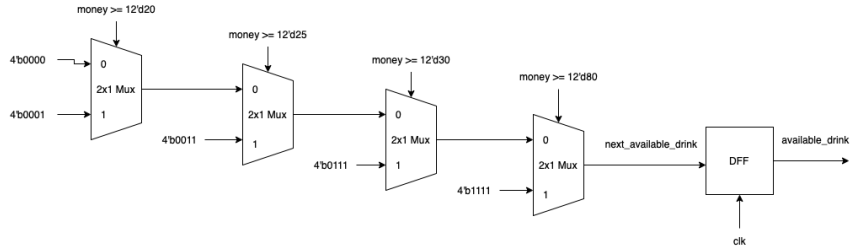
- vending machine
  - state



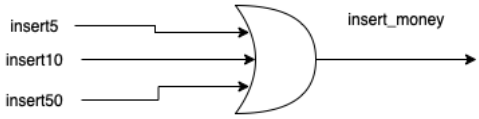
- money



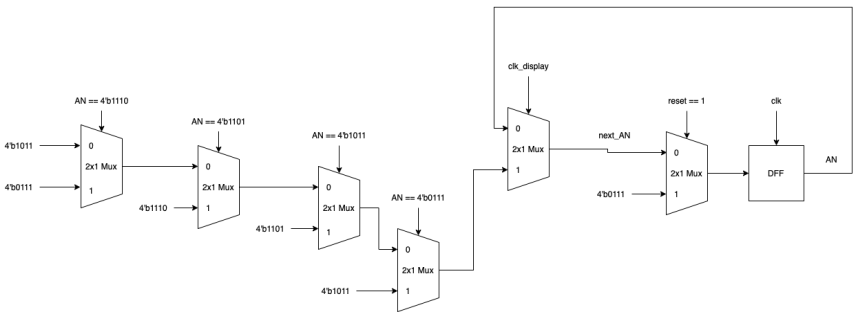
- available drink



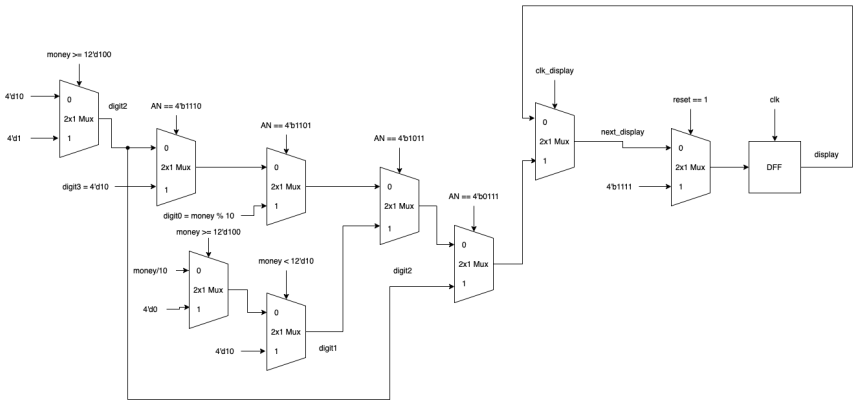
- insert_money



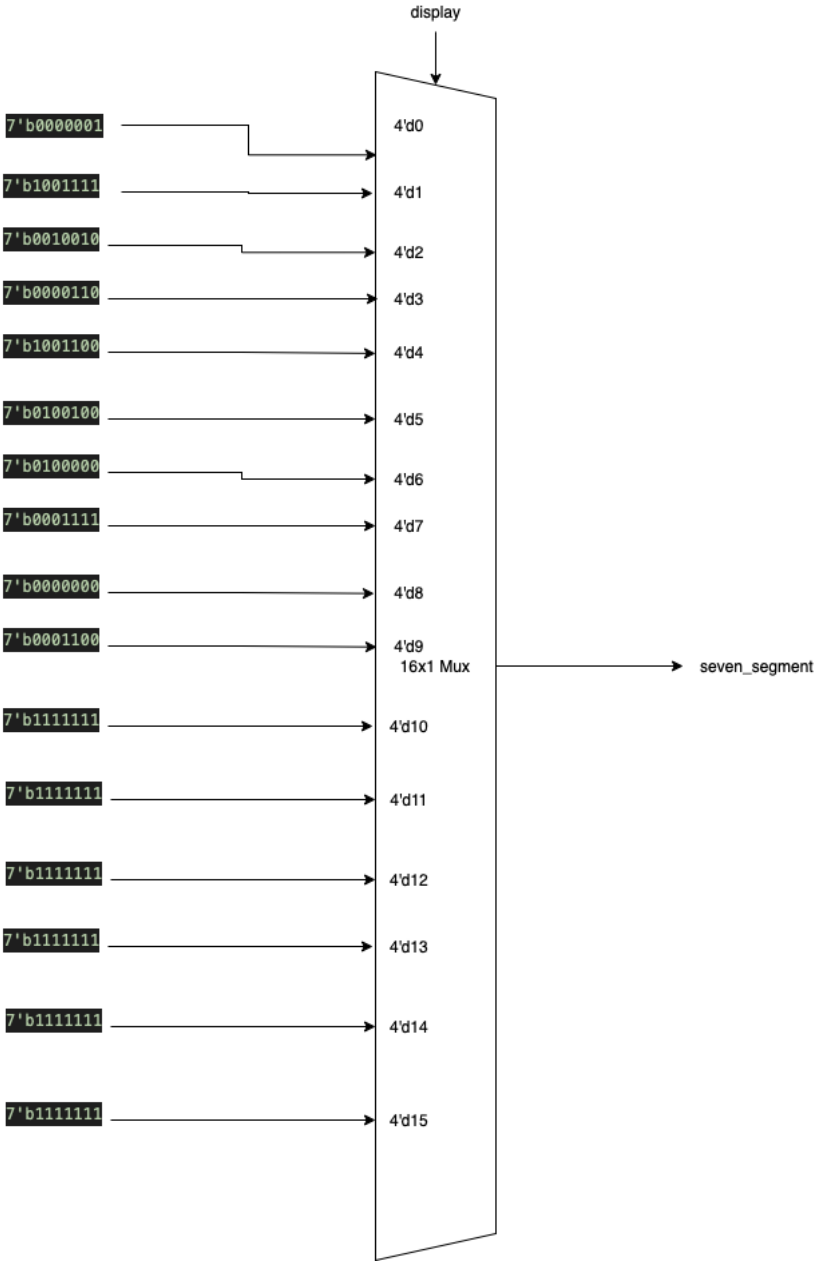
- seven_segment
  - AN



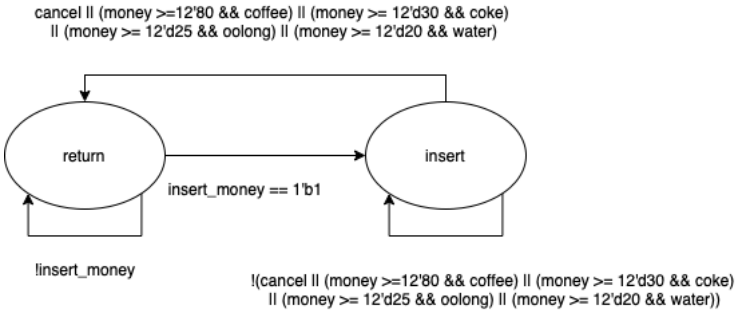
- display



- seven_segment



State Diagram of the design of Verilog



Design Explanation

以下我們將會分開解釋每個module的設計

- Top Module:
  - 把PS2_DATA, PS2_CLK, clk, rst接入模板所提供的keyboard_decoder來進行decode，並利用key_down來得知現在哪一個鍵盤按鍵是被按下的，例如A被按下，則key_down[9'b000011100]=1'b1，而我們依據A,S,D,F所對應的key_code以及key_down[key_code]的值來判斷A,S,D,F哪一個被按下了，並將該訊號以及been_ready做and的計算並分別接到coffee, coke, oolong, water。
  - 把reset, insert5, insert10, insert50, cancel做debounce以及one_pulse
  - 把clk, clk1hz(1hz的clk), on_rst, on_insert5, on_insert10, on_insert50, on_cancel, coffee, coke, oolong, water作為input接入vending machine module，並把money, available_drink接到相對應的output port
  - 把available_drink接到LED並在之後於fpga版上的LED[3:0]顯示哪個drink是available的
  - 把clk, clk_display(clock with a frequency of 1/2^17 clk), on_rst, money接到seven_segment module，並把display, AN接到相對應的output port，並於之後在fpga版上的七算顯示器上面顯示投入多少錢。
- Vending Machine:
  vending machine可分為sequential以及combinational的部分，而我們將分開做解釋

1. sequential:
   - reset:將state設為return, money以及available_drink都設為0

    - !reset:將next_state, next_money, next_available_drink的值用來update
2. combinational
    - state transition
        - state == return:
            - 在有錢幣被投進來的時候(insert_money == 1'b1)進到insert state
        - state == insert:
            - 在cancel被按下或者有飲料成功被購買後進到return state
    - change money
        - state == return:
            - 在錢的金額大於0的時候每秒 (利用clk1hz作為enable) 減少五塊
            - 在有錢投入的時候也要update money的value並於下一個cycle進到insert state
        - state == insert:
            - 如果投入硬幣的話，在錢的金額加上所投入的金額之後大於100則維持100，否則直接把所投入的金額加上原本的錢。
            - 如果飲料被買了且錢夠的話就扣掉飲料的錢
    - change available drink
        - available_drink的四個bit分別代表能不能買(coffee, coke, oolong, water)
        - 檢查錢的金額是否大於等於四種飲料
            - money >= 80:available_drink = 4'b1111
            - money >= 30:available_drink = 4'b0111
            - money >= 25:available_drink = 4'b0011
            - money >= 20:available_drink = 4'b0001
            - money < 20:available_drink = 4'b0000
- seven_segment:
    - AN:由4'b0111開始，每一個cycle更新一次
        - (4'b0111->4'b1011->4'b1101->4'b1110)
    - display:
        - AN = 4'b0111:display = 4'd10(不亮)

        - AN = 4’b1011:當money >= 100的時候display = 4’d1
        - AN = 4’b1101:
            - money >= 100:display = 4’d0
            - 10 <= money < 100:display = money/10
            - money < 10 :display = 4’d10(不亮)
        - AN = 4’b1110:display = money % 10
    - seven_segment:根據display的值去對應到七算顯示器，而在我們的設計中，當display的值大於10的時候，一率不亮七算顯示器

Contribution

朱季葳：advanced question 3、4; FPGA 2
施泳瑜：advanced question 1、2; FPGA 1

What we have learned from Lab5

1. 在fpga題目上，因為又加上了外接元件:speaker、keyboard，有更多需要去注意的點，因此有點手忙腳亂，而在最後找到方法解決所有問題(接線、加入模組...)之後格外有成就感。
2. 這次畫圖也因為牽涉到外接元件的因素而不太會畫，參考完講義之後，對原本上課給的 sample code 有更深一層的了解，也透過模仿成功把圖畫出來了。
3. 因這次牽涉很多要算 clock 的問題，所以在寫題目的時候算到不行，是蠻特別的體驗，然後希望有算對 QQ 。