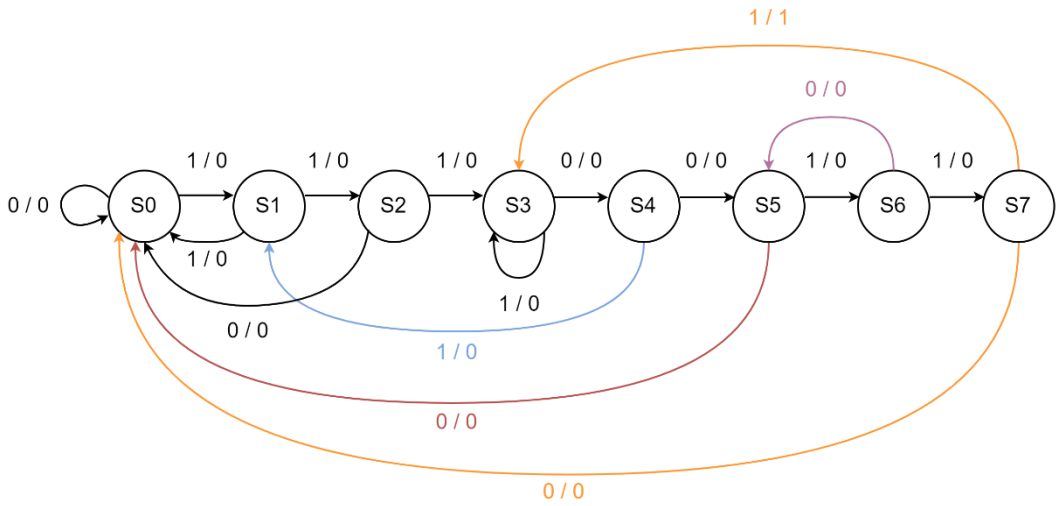
Lab 5 report

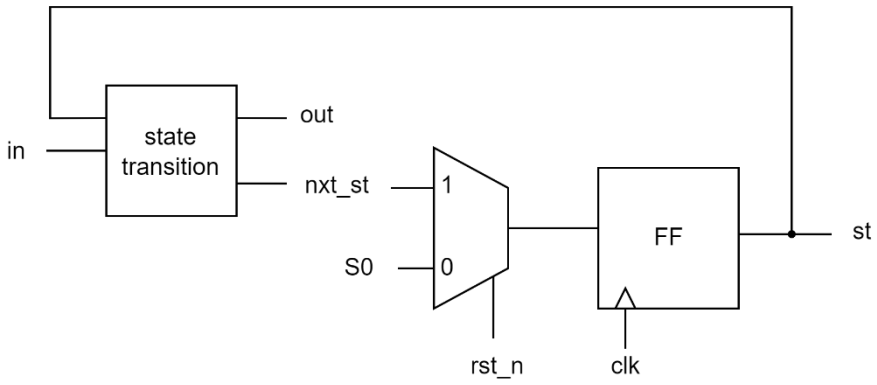
組員：110062221 李品萱
110062213 唐翊雯

I. Sliding window sequence detector

這題我們需要用 Mealy machine 作連續的 detect，找到 sequence 1110(01)+11，state transition diagram 如下圖。S0 到 S2 都是要收到 1 才能進入下一個 state 對應到 sequence 的前三個 1，在這個階段如果收到 0 都不是合法的，會直接回到 S0；但由於是用 sliding widow 的方式檢查，因此 S3 若收到 1 則會停留在 S3，一直到收到 0 才會進下一個 state；S4 則是要收到 0 才合法，因此若收到 1 則會視為 sequence 已 input 一個 1，因此進到 S1；在 S5 及 S6 的部分，由於只能存在 01 且可以存在多個 01，因此 S5 收到 1 進到 S6，S6 若收到 0 則回到 S5，檢查下一個是否為 1；S7 的部分則代表前面已經收到 1110(01)1，因此這邊若收到 1 則會 output dec 為 1，收到 0 是不合法的，則回到 S0。

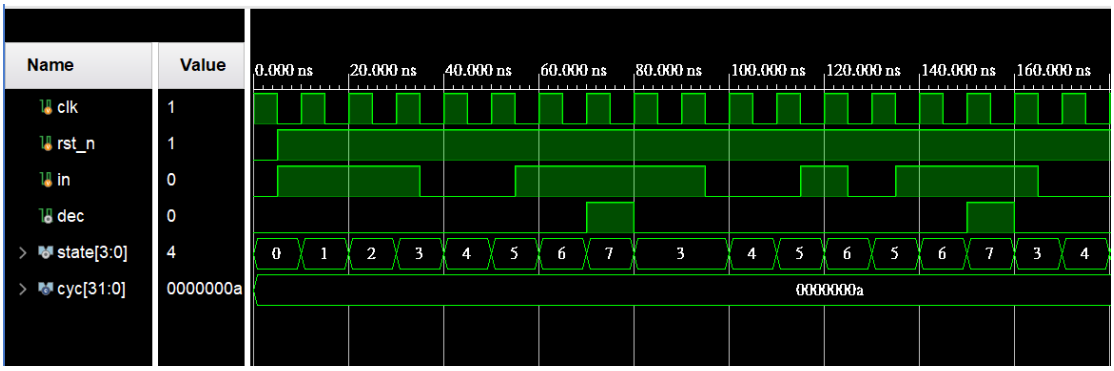


這題的 block diagram 如下。

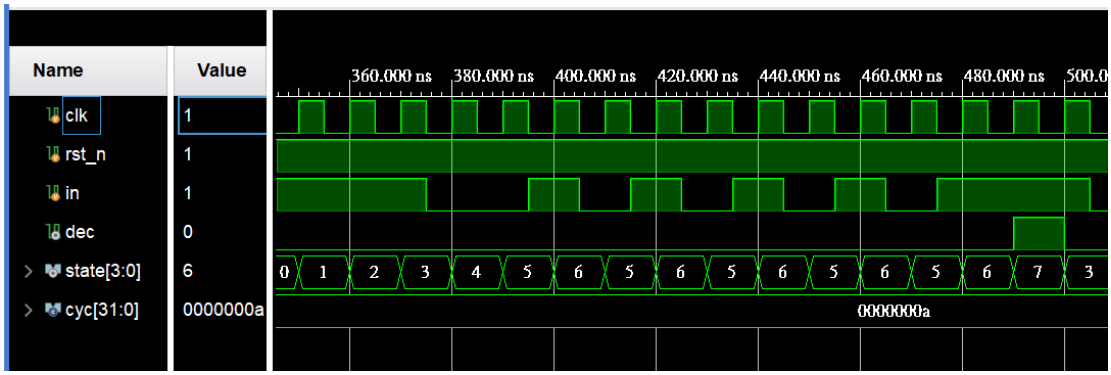


在 testbench 的部份，為了方便檢查我們將 state 也印出來，首先做出 spec

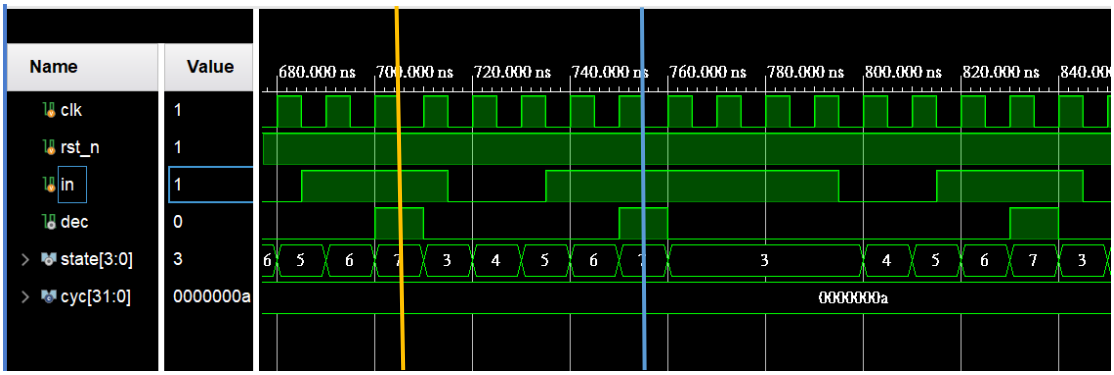
給的 sample waveform，如下圖。



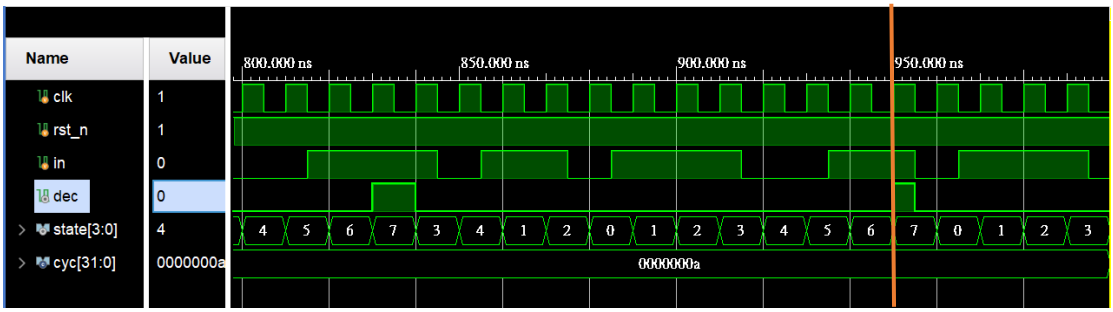
接下來我們測 01 在 1110 與 11 之間出現多次，確認符合預期結果。



接下來檢查我們 dec 為 1 之後的 sequence 開頭不給完整的三個 1，及持續給 1 的狀況，依序是下圖中黃色線及藍色線處，皆符合 state transition 的結果。

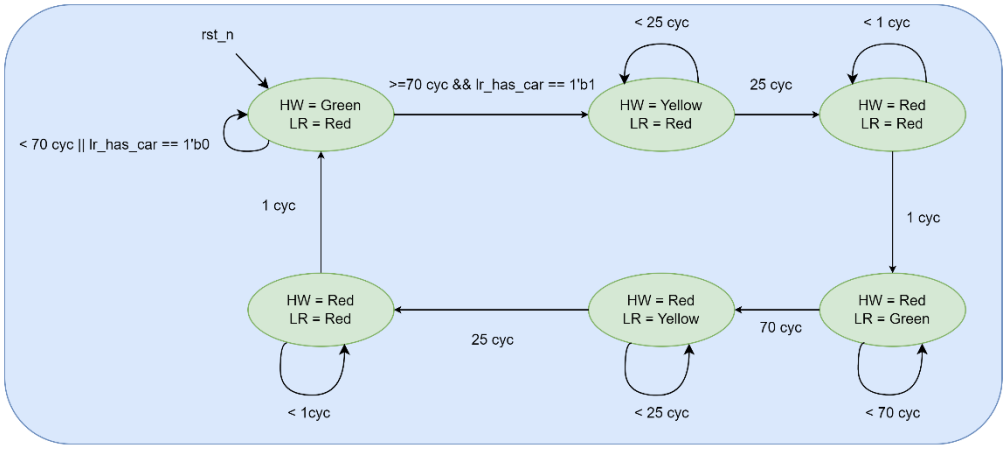


我們也測試不合法的輸入，皆符合預期結果，而在下圖橘色線處 dec 會產生一個 clock 的 1 是因為它在 s6 時讀到 1，而後在 clock 起來，state 變 7 時又讀到 1，但在該 clock low 時 input 為 0，不是合法的 sequence，因此 dec 此時為 0。

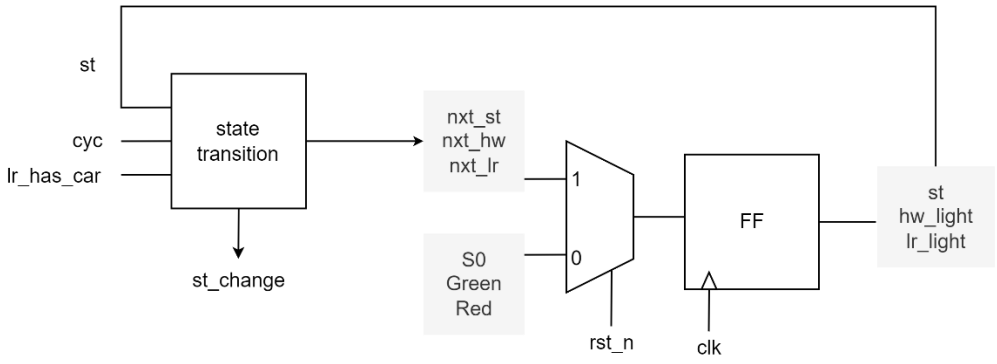


II. Traffic light controller

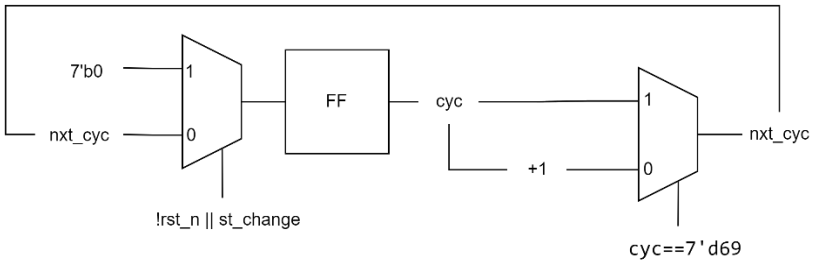
這題我們要做出一個紅綠燈，基本的 state transition 都在 spec 中有給了，我們基於 spec 再將 state transition diagram 處理得更完整，如下圖。



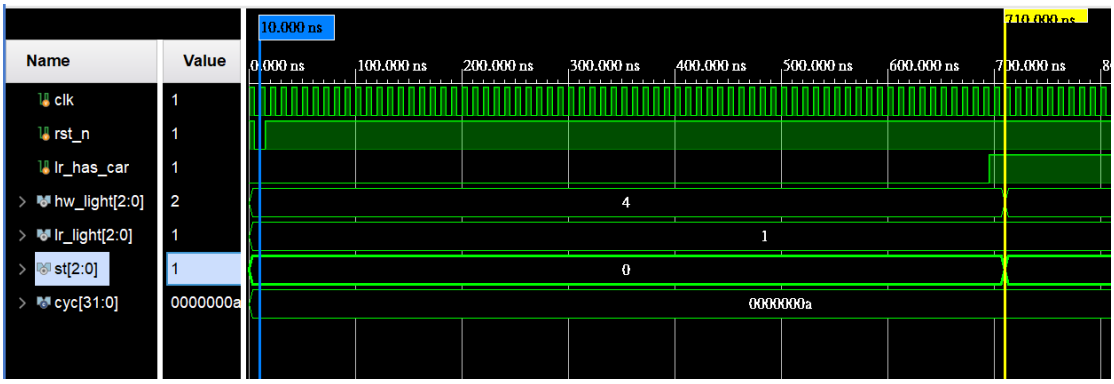
這題的 block diagram 如下圖，在這題的設計上，因為 state transition 是受 clock cycle 控制的，我們將燈號跟 state 一起變，並將 st_change 作為 FSM 的 output，也用來控制計算 clock cycle，clock cycle 控制的設計如下段所述。



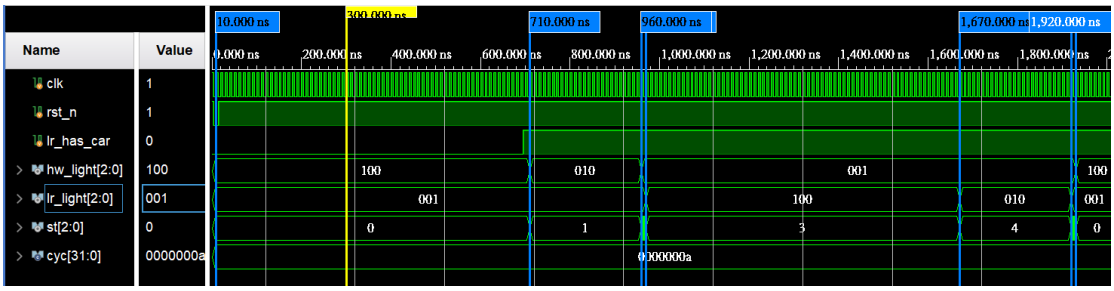
如下圖，在計算 clock cycle 上，我們讓 state change 發生時將 cyc 歸 0，否則加 1，因此只要 state 一變 cyc 就會開始數，並配合 FSM 的設計停留在原本的 state 直到 cyc 符合要求；又由於這題需要的 clock cycle 最大只到 70，因此我們讓 cyc 是 69 時就不繼續加 1，方便我們決定 cyc 的 bits 數。



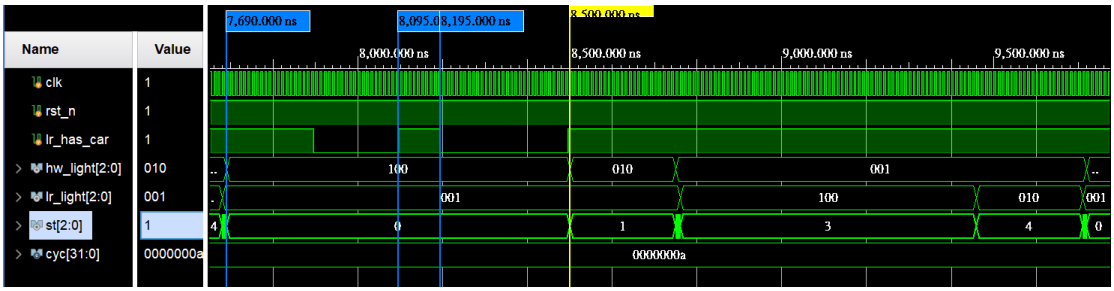
在 testbench 的部分，我們一樣印出 state 方便觀察。
首先確認 cycle 數是對的，如下圖，黃色線部分是 710ns，而我們是在藍色線，即 10ns 時 reset 的，因此可確認它經過了 70 個 cyc。



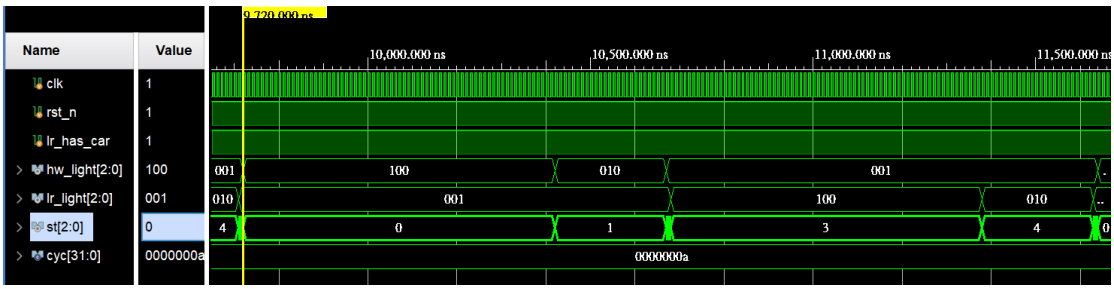
燈號變化的部份，我們按照 spec，紅燈為 001，黃燈為 010，綠燈為 100。
首先我們測在 70 cyc 前 lr_has_car 就為 1 持續到 70 cyc 後，確認 state transition 及燈號都是對的，下圖我們也用藍色線標出各個 state 發生變化的時間，確認從 state 0(HW=Green, LR = Red)到 state 5(HW = Red, LR = Red)之間經過的 cyc 依序為(70, 25, 1, 70, 25, 1)



而後我們測在 cyc 未達 70 前 lr_has_car 為 1，但在 cyc 不到 70 時 lr_has_car 就變 0，如下圖，可以看到在大約經過 50 個 cyc 時，lr_has_car 就有 1 到 0 的變化，但因為 cyc 未達 70，因此不會進入下個 state；而在 cyc 達 70 時，lr_has_car 為 0，一直到 lr_has_car 為 1 時，才進入下一個 state（黃線處）。

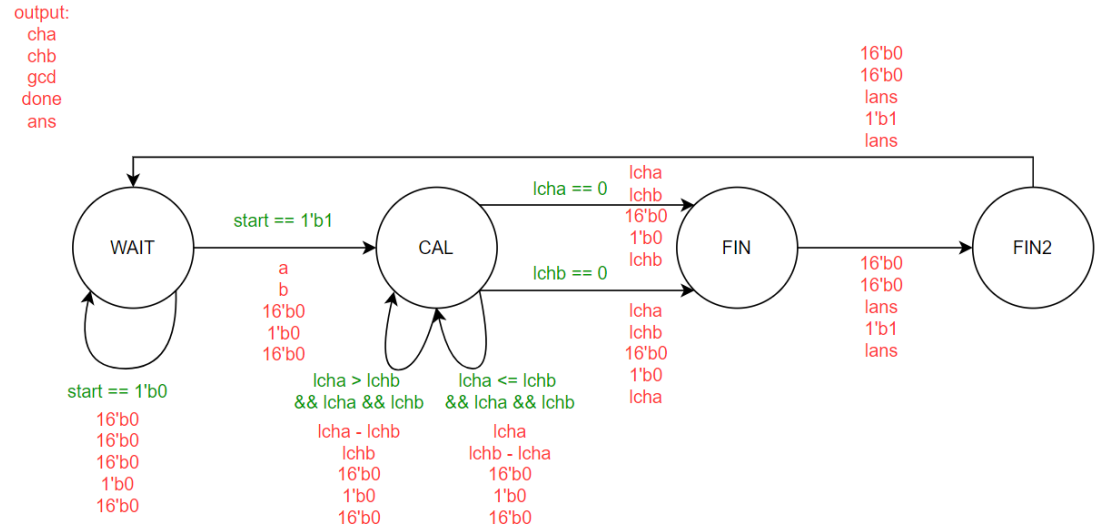


最後我們也對 lr_has_car 一直為 1 做確認，此情形義同在 cyc 為 70 之前 lr_has_car 就為 1，如下圖。



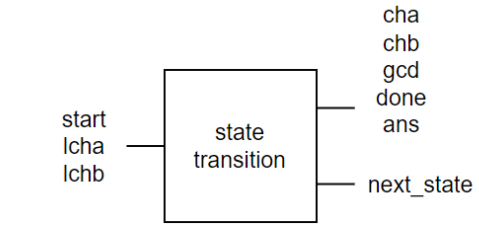
III. Greatest common divisor

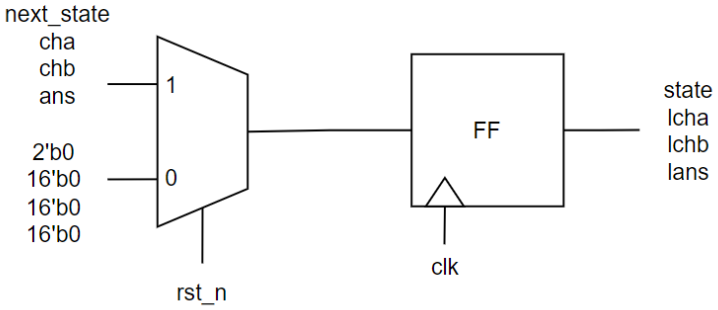
這題我們要對於給定的兩個數，利用輾轉相除法找出最大公因數。我們依照給定的 pseudo code 來實作，其 state diagram 如下：



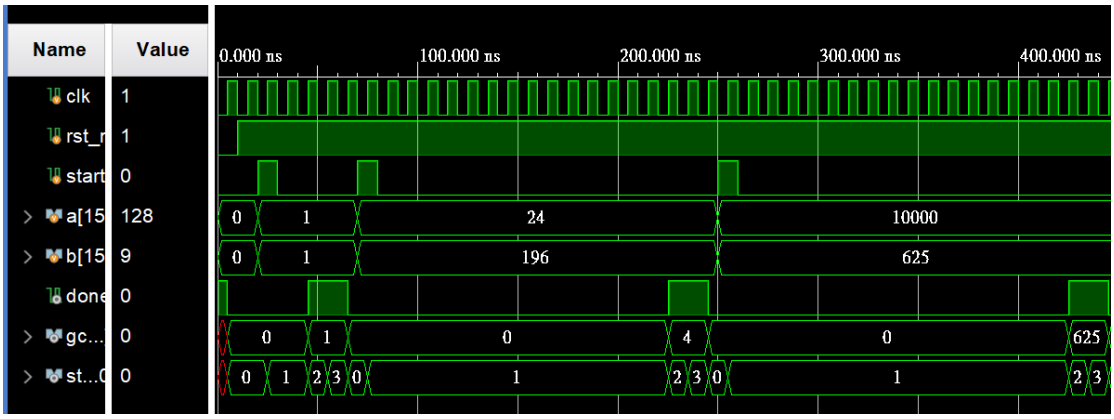
output 的部分有 cha（fetch 出來的 a）、chb（fetch 出來的 b），gcd，done 跟 ans（輾轉相除法的 return 值）。Input 的部分有 start、lcha（上一個 clock cycle 的 cha）、lchb（上一個 clock cycle 的 chb）。

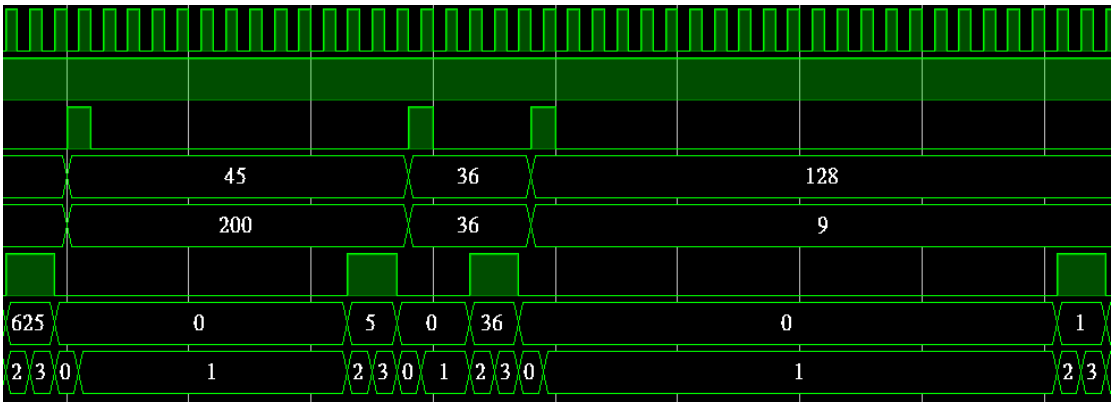
sequential circuit 的部分維護了 state 與前一個 clk 的 cha、chb、ans 的值，整體的 block diagram 如下：





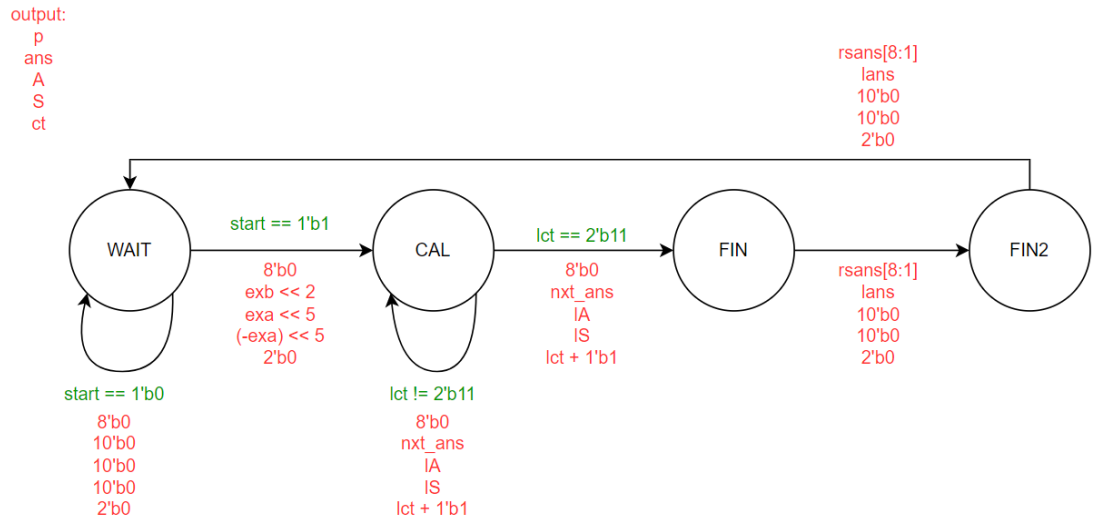
Testbench 的部分，我們測了幾組數字，盡量列舉各種情況（e.g. a == b、a 與 b 互質、a > b、b > a、gcd(a, b) == b…）以驗證其正確性。為了確認 code 有正常運作，我們將 state 印出來看，確定 state 的運作都是正確的。我們也事先推了每組數字應要在 CAL 這個 state 跑幾個 clock cycle，以確認 code 有正常運作。





IV. Bonus: Booth multiplier

這題要我們使用 Booth Multiplier，算出給定兩個數字的乘積。根據 spec 給定的資訊以及作為參考的 state diagram，我們做出的 state diagram 如下：

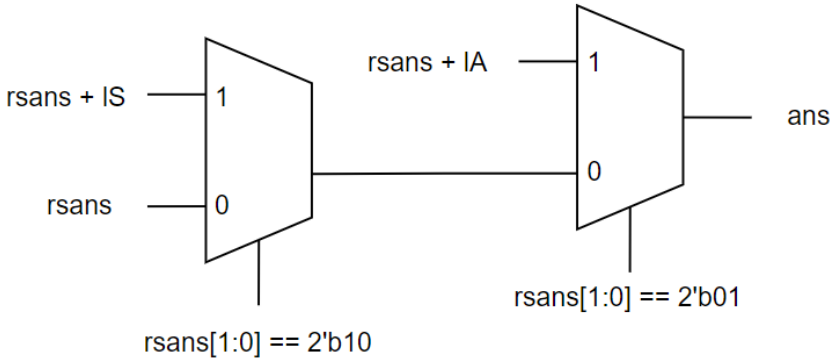


Output 的部分有 p、ans、A、S、ct。ct 代表的意思是在 CAL 這個 state 中，已經做過幾次運算。而 A 跟 S 即是維基百科定義的 A 與 S，而 ans 為維基百科定義的 P：

> Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values *A* and *S* to a product *P*, then performing a rightward arithmetic shift on *P*. Let **m** and **r** be the multiplicand and multiplier, respectively; and let *x* and *y* represent the number of bits in **m** and **r**.

Input 的部分則有 start、lct（上一個 clock cycle 的 ct）。

lct、lA、lS、lans 分別代表前一個 clk 的 ct、A、S、ans 的值。rsans 代表前一個 clk 的 ans 經過 arithmetic right shift 後的結果。在 CAL 這個 state 中，ans 的輸出 (上圖以 nxt_ans 表示) 較為複雜，沒有標註在 state diagram 上，其詳細的 block diagram 如下：

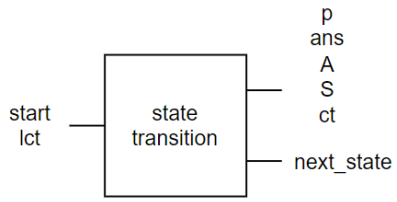


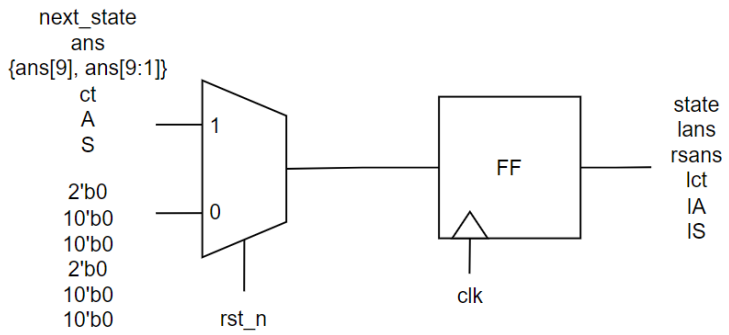
exa、exb 則分別為 input 的 a、b。但型態的部分，exa 宣告成 signed [4:0]，

宣告成 [4:0] 是因為在計算 S 的值時需要用到 -a，這時若 a 為 -8，則 -a 的值為 8，無法被 [3:0] 存起來，因此多宣告 1 個 bit 來避免這個情況發生。宣告成 signed 則是因為從 [3:0] 轉成 [4:0] 要保持存的數字相同。Exb 的部分，不使用 signed 宣告，這樣才不會在對 ans 初始化時改變 ans 的 MSB。

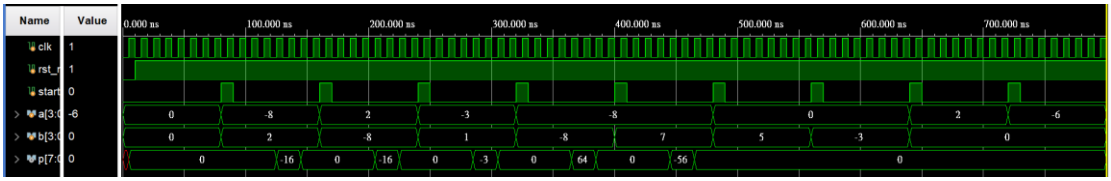
```
15 wire signed [4:0] exa;
16 wire [3:0] exb;
17 assign exa = a;
18 assign exb = b;
```

sequential circuit 的部分維護了 state 與前一個 clk 的 ans、ct、A、S、arithmetic right shift 後的 ans 的值，整體的 block diagram 如下：





Testbench 的部分，我們測了幾組數字，盡量列舉各種情況（e.g. a, b 皆為正、a, b 皆為負、a, b 一正一負、a = 0、b = 0、a = -8、b = -8…），以驗證其正確性。



Booth Multiplier 的原理是利用這個性質：

$$(\ldots 0\overbrace{1\ldots 1}^{n}0\ldots)_2 \equiv (\ldots 1\overbrace{0\ldots 0}^{n}0\ldots)_2 - (\ldots 0\overbrace{0\ldots 1}^{n}0\ldots)_2$$

即一段連續的 1 可以被表示成兩數的差，其中這兩數皆為 2 的冪次。
因此當讀到 01 時，代表為一段連續的 1 的開始；讀到 10 時，代表為一段連續的 1 的結束。在這兩個時候要分別加上和減掉被乘數 * 2^當前計算的次方。透過這樣的替換，可以比一般乘法器做更少的 operation。

V. FPGA: Mixed keyboard and audio modules together

這題要把 keyboard 跟 audio 的 module 合在一起，並實作出 spec 上提到的功能。我們將 basic 提供的 keyboard 與 musicbox 的模板，做適當的修改並合在一起以完成這題。

Audio 的部分，要修改的地方有要播出來的 note 以及更改 note 的速度。原本模板中的 beatFreq 為 32'd8，對應到 1 beat = 0.125 sec。根據題目要求，我們開了兩個 parameter，32'd1 與 32'd4，分別對應到 1 beat = 1 sec 與 1beat = 0.5 sec。

```
parameter BEAT_FREQ_0 = 32'd1;  //one beat=1sec
parameter BEAT_FREQ_1 = 32'd2;  //one beat=0.5sec
```

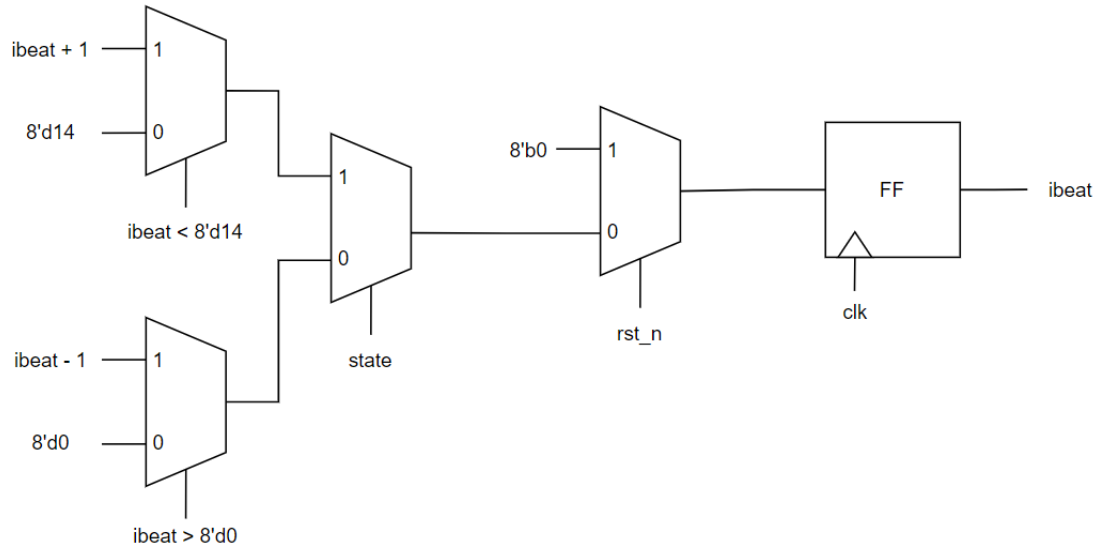
同時我們也要更改傳入 module 的 freq，我們另外開了一個 reg 並傳入，這樣便能根據按下的鍵更改其值。
Note 的部分，我們修改了 Music、PlayerCtrl 兩個 module。PlayerCtrl 的部分要修改總 note 數，只要把這項 parameter 改成題目給的 15 即可（總共有 15 個 note）：

```
parameter BEATLEAGTH = 15;
```

除此之外，由於題目要求 "When it reaches C4 or C8, stay on the note"，以及 note 的變化要分為遞增及遞減，而不像原本的 module 只有遞增並且循環播放。因此我們對這邊的 always block 做了下列調整：

```
always @(posedge clk, posedge reset) begin
    if (reset)
        ibeat <= 0;
    else if(state == 1'b1) begin
       if(ibeat < 8'd14) ibeat <= ibeat + 1;
       else ibeat <= 8'd14;
    end
    else begin
       if(ibeat > 8'b0) ibeat <= ibeat - 1;
       else ibeat <= 8'b0;
    end
end
```

這部分的 block diagram 如下，其中 state 代表的是現在是遞增或遞減：



Music 的部分，我們把原本小蘋果的旋律改掉，改成題目要求的 C4~C6：

```
always @(*) begin
    case (ibeatNum)
        8'd0 : tone = `C4;
        8'd1 : tone = `D4;
        8'd2 : tone = `E4;
        8'd3 : tone = `F4;
        8'd4 : tone = `G4;
        8'd5 : tone = `A4;
        8'd6 : tone = `B4;
        8'd7 : tone = `C5;
        8'd8 : tone = `D5;
        8'd9 : tone = `E5;
        8'd10 : tone = `F5;
        8'd11 : tone = `G5;
        8'd12 : tone = `A5;
        8'd13 : tone = `B5;
        8'd14 : tone = `C6;
        default : tone = `sli;
    endcase
end
```

接著是 Keyboard 的部分，只需要對 Top module 做修改即可，首先我們會用到的鍵如下：

```
parameter [8:0] KEY_CODES_w = 9'h1d;
parameter [8:0] KEY_CODES_s = 9'h1b;
parameter [8:0] KEY_CODES_r = 9'h2d;
parameter [8:0] KEY_CODES_enter = 9'h5a;
```

我們需要用鍵盤控制的 reg 有：state（現在為遞增或遞減）、speed（現在的速度為 0.5 sec 還是 1 sec per note）與 rst_n。我們根據題目，在按鍵按下去時給予相對應的值：

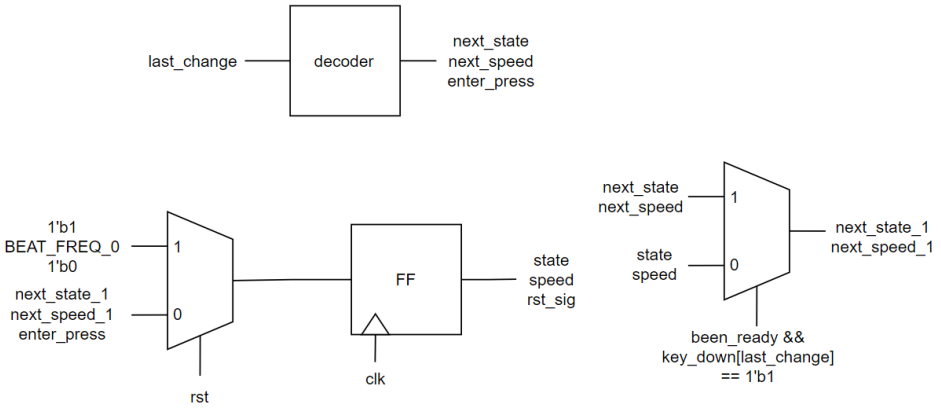
```
always@(*) begin
    case (last_change)
        KEY_CODES_w : begin
            next_state = 1'b1;
            next_speed = speed;
            enter_press = 1'b0;
        end
        KEY_CODES_s : begin
            next_state = 1'b0;
            next_speed = speed;
            enter_press = 1'b0;
        end
        KEY_CODES_r : begin
            next_state = state;
            next_speed = speed == BEAT_FREQ_0 ? BEAT_FREQ_1 : BEAT_FREQ_0;
            enter_press = 1'b0;
        end
        KEY_CODES_enter : begin
            next_state = 1'b1;
            next_speed = BEAT_FREQ_0;
            enter_press = 1'b1;
        end
        default: begin
            next_state = state;
            next_speed = speed;
            enter_press = 1'b0;
        end
    endcase
end
```

接著我們判斷 last_change 時確認 next_state 與 next_speed 的值，然後在 sequential 的部分，我們再把這些值傳給 state、speed 與 rst_n：

```
always @ (*) begin
    if (been_ready && key_down[last_change] == 1'b1) begin
        next_state_1 = next_state;
        next_speed_1 = next_speed;
    end
    else begin
        next_state_1 = state;
        next_speed_1 = speed;
    end
end
```

```
always @ (posedge clk, posedge rst) begin
    if (rst) begin
        state <= 1'b1;
        speed <= BEAT_FREQ_0;
        rst_sig <= 1'b0;
    end else begin
        state <= next_state_1;
        speed <= next_speed_1;
        rst_sig <= enter_press;
    end
end
```

這些部分的 block diagram 如下：



將 state 與 speed 接回剛剛與 audio 有關的 module 後，即完成整份 code。

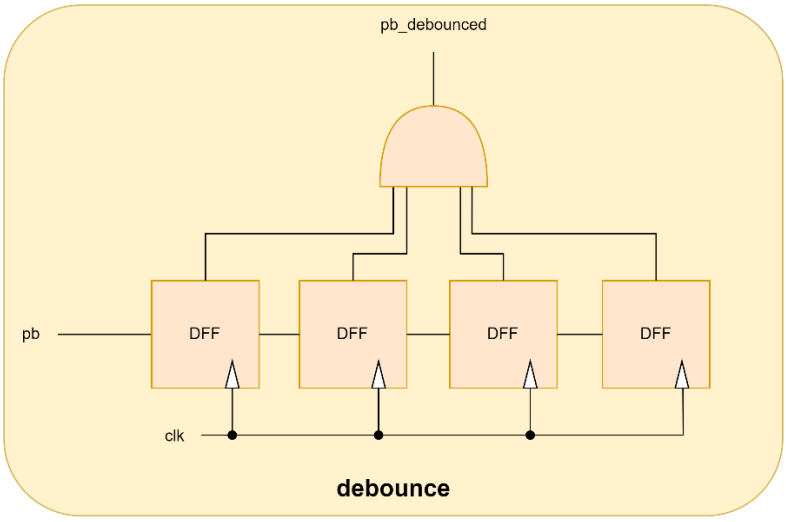
VI. FPGA: Vending machine

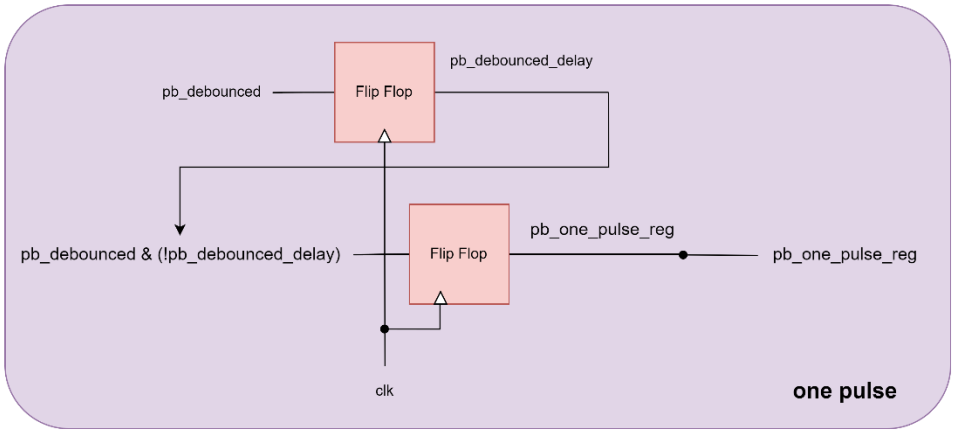
這次的 FPGA 我們需要用鍵盤跟板子實做一個販賣機，我們大致可分為以下部分：

- 投錢
- 選擇可以買的飲料或取消購買
- 找錢
- 7-segment display

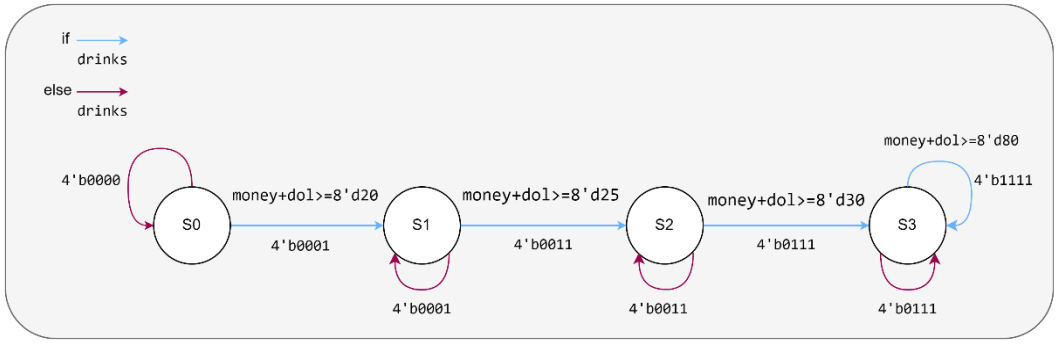
投錢

投錢是以按按鈕的方式，分別可以投 5、10、50 元，與前幾次 lab 一樣，我們需要對 bottom 做 debounce 跟 one pulse 的處理，分別如下兩張圖。





由於投進的錢幣數值會決定可選擇的飲料種類，因此我們用 FSM 的方式處理，如下圖，dol 代表這次投進的錢，money 代表從開始投錢到現在累積的錢，drinks 代表現在可以買的飲料種類，(0 不能買，1 可以買)。



選擇可以買的飲料或取消購買

接下來我們會用鍵盤的 a, s, d, f 鍵選擇要買什麼飲料，或藉由板子上的按鈕取消購買，按鈕的基本處理如前面所述，而鍵盤我們參考上課講義及 basic 的 SampleDisplay 檔案處理。

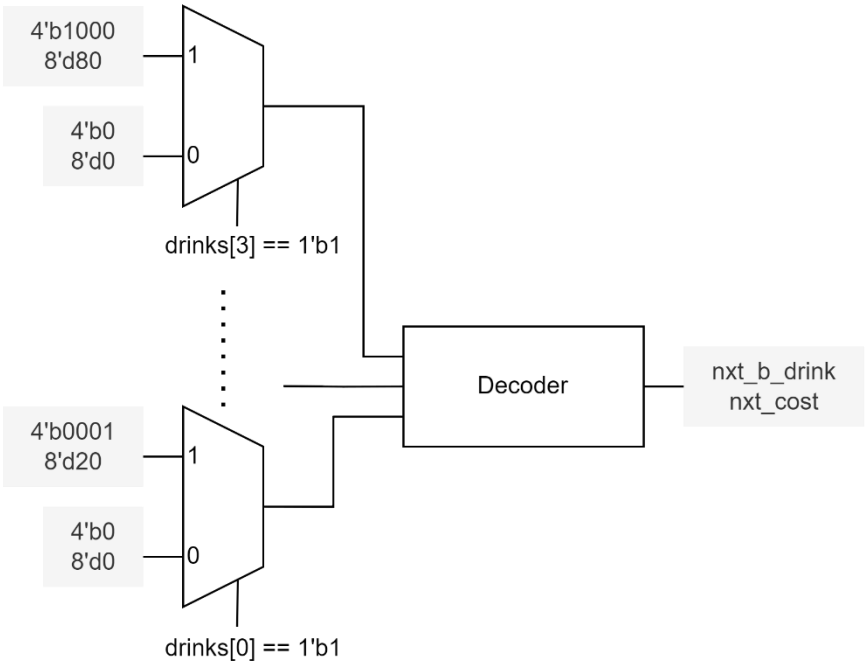
我們先將鍵盤的 code 對應出來，如下圖，每個 parameter 會是 last_change 的判斷依據。

```
parameter KEY_CODES_A = {1'h0, 4'h1, 4'hc}; // coffee
parameter KEY_CODES_S = {1'h0, 4'h1, 4'hb}; // coke
parameter KEY_CODES_D = {1'h0, 4'h2, 4'h3}; // oolong
parameter KEY_CODES_F = {1'h0, 4'h2, 4'hb}; // water
```

在判斷 last_change，即鍵盤按下什麼鍵時，我們也同時確認現在這個飲料是可以買的，以 KEY_CODE_A 舉例如下圖，這部份我們會得到買了什麼飲料及飲料所需的價錢。

```
case(last_change)
        KEY_CODES_A: begin
            if(drinks[3] == 1'b1)begin
                nxt_b_drink = 4'b1000;
                nxt_cost = 8'd80;
            end
            else begin
                nxt_b_drink = 4'b0;
                nxt_cost = 8'd0;
            end
        end
```

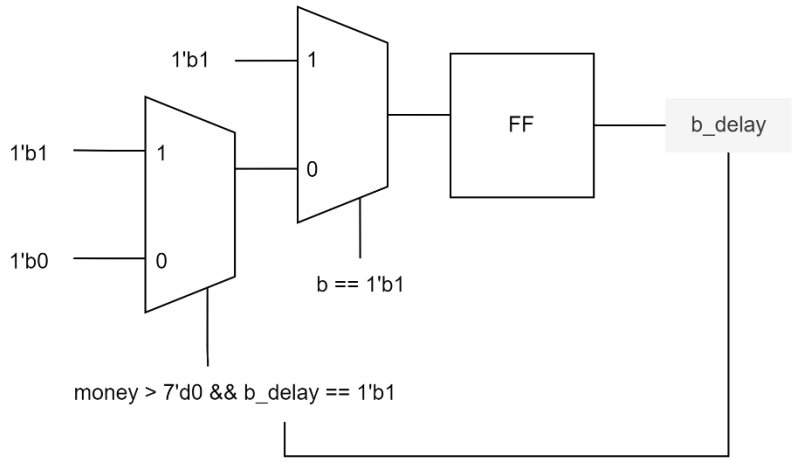
Block diagram 如下圖。



在判斷鍵盤有按下按鍵（即判斷 key_valid 及 key_down[last_change]）的部分，我們會多記一個 b，b 為 1 代表有成功買東西，0 反之。因為 b 會拿來判斷現在是在投錢還是找錢，因此 b 如果是 1 的話，會一直維持 1 直到找過程結束，我們運用之前 lab 的 ping pong counter 中 flip 訊號 delay 的想法實現，設計如下，cancel 也有做相同的處理。

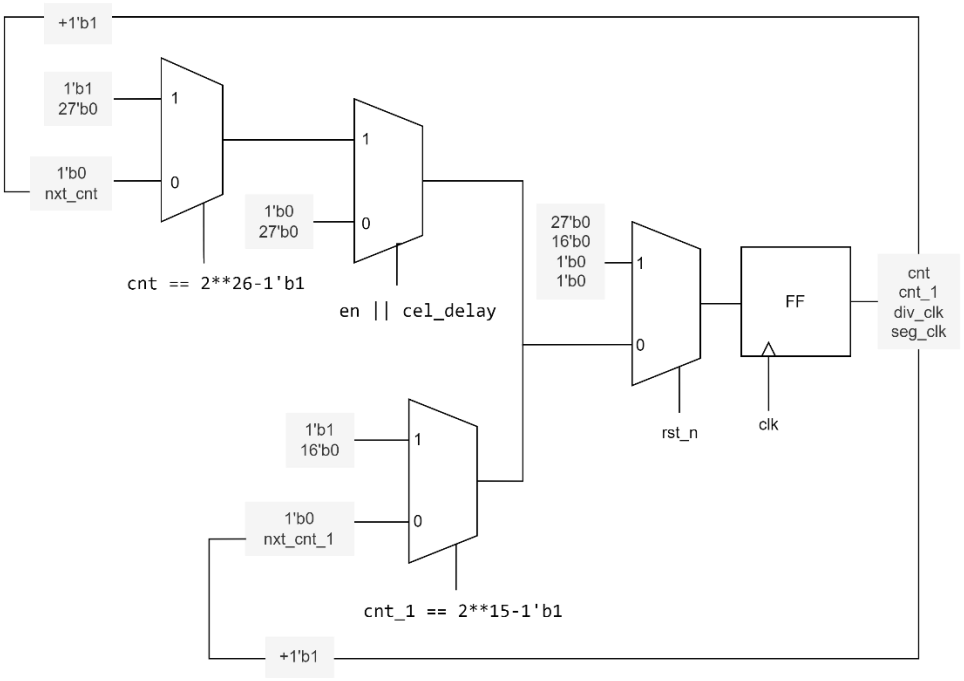
```
always@(posedge clk)begin
    if(b == 1'b1) b_delay <= 1'b1;
    else if(money > 7'd0 && b_delay == 1'b1) b_delay <= 1'b1;
    else b_delay <= 1'b0;
end
```

b_delay 的 block diagram 如下。

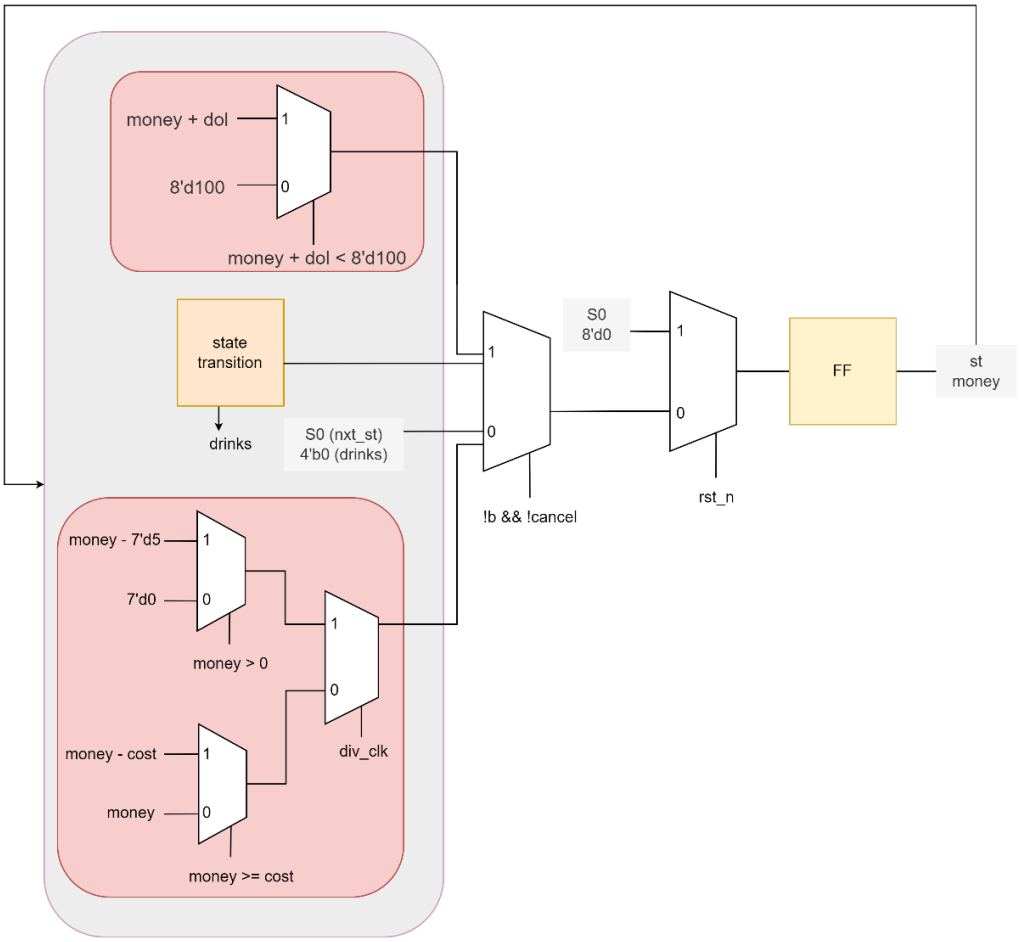


找錢

接下來是 vending machine 內部的運作部分，當 b 為 1 或 cancel 為 1 時（下圖的 en||cel_delay），會進入到找錢的環節，當這個環節被觸發後，state 就會回到 S0，drinks 也會全部歸零，也就是代表什麼飲料可以買的 LED 燈會全部熄滅。如果是買飲料的話，會先顯示剩餘的錢再做每秒減五，為了做到這件事，我們讓負責做 div_clk 的 counter 只有在 b 為 1 時才會做計算，這樣便可以讓我們顯示完剩餘的錢再做每秒減五；而 cancel 則是單純每秒減五，用 clock divider 控制，clock divider 的 block diagram 如下圖，div_clk 控制每次退五元，seg_clk 控制 7-segment 的顯示。



而 vending machine（不包含 display）整體運作的 block diagram 如下。



7-segment display

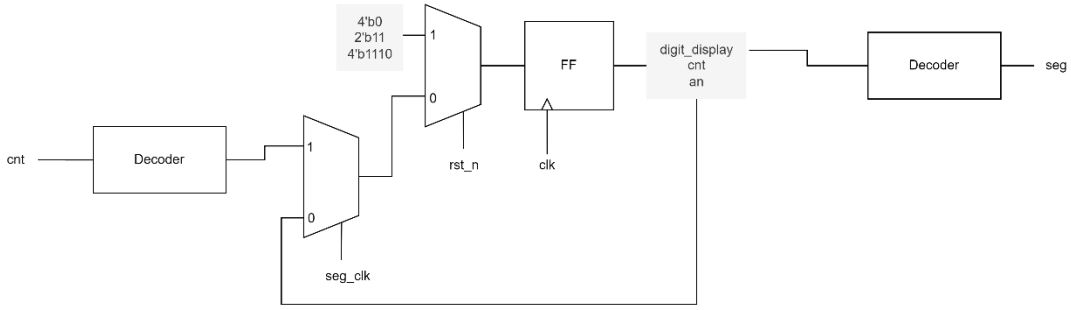
Spec 中有提到不能 prepend 0，因此在顯示上我們需要做一些處理。首先，在接各個數字的 7-segment 對應亮什麼時，我們多寫一個全都不亮的狀況，然後在輪流亮 an 時做判斷。

亮 an[0]時，我們要亮的數值會是 money - (money/7'd10)*7'd10；

亮 an[1]時，如果小於 10，an[1]要全暗，否則亮 money / 7'd100，但若 money 是 100 的話就不適用，而是直接亮 0；

亮 an[2]時，如果小於 100，an[2]要全暗，否則亮 money / 7'd100；

7-segment display 的 block diagram 如下。



以上即為 vending machine 的實作呈現。

VII. Summary

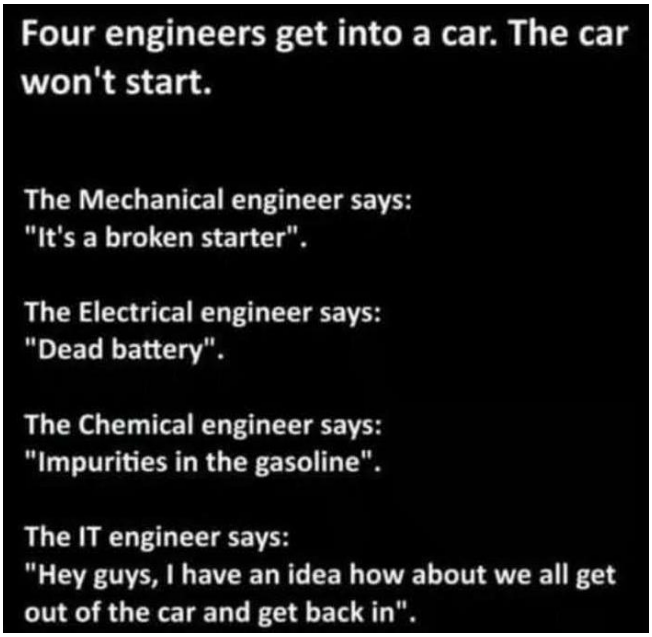
這次的 Lab 我們學習到如何運用聲音模組及鍵盤，這也讓 FPGA 的實作更有貼近生活的感覺，也讓我們對平常使用的鍵盤有更完整的理解，對音訊的處理也有了近一步的認識。

在 advanced 中，我們複習了 FSM 的使用，這次設計的 mealy machine 也比上次複雜，讓我們更了解 mealy machine 的運作。在紅綠燈的控制設計中，也了解到 FSM 在生活中的運用會是什麼樣子的呈現。

GCD 跟 Booth Multiplier 的部分其實很相似，我們學到如何好好運用 FSM 來處理問題。這兩題只要將 state diagram 定義完整便很容易實作，告訴我們先把 state 好好定義出來的重要性。

至於販賣機的實作部分，我們學到如何從 0 design 一個 FSM，在不同的 state transition 的設計方式中，想到最適合的方式。一開始以為這一題會較為複雜，但實作下來發現除了一些細節要注意之外，沒有想像中的難，而最後的成果也的確作出了我們平常所知道的販賣機，非常有趣。

這次在跑 vivado 的時候，偶爾會遇到看不懂錯誤訊息的問題，花了不少時間，後來才發現大部分時候重開就好了，學到了一課：



VIII. Contributions

- **Code**

  Sliding window sequence detector by 唐翊雯

Traffic light controller by 唐翊雯
Greatest common divisor by 李品萱
Bonus: Booth multiplier by 李品萱
FPGA: Mixed keyboard and audio modules together by 李品萱
FPGA: Vending machine by 唐翊雯

- **Report**

  各自描述負責的題目