Lab 5 Keyboard and Audio Modules

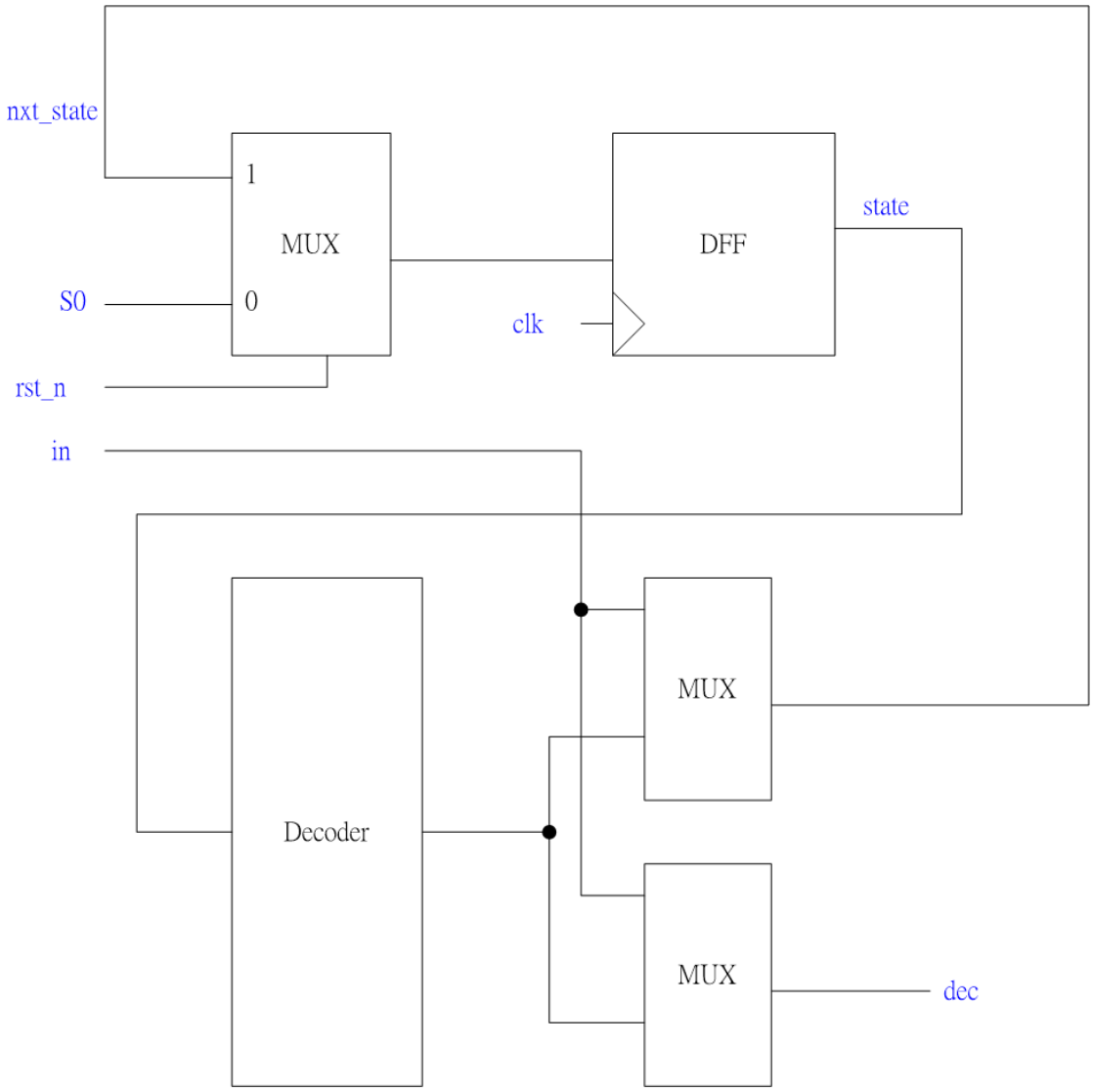
實驗報告

組長:劉奇泓 109033135

組員:洪聖祥 109062315

1. Sliding window sequence detector (mealy machine)

Block diagram:

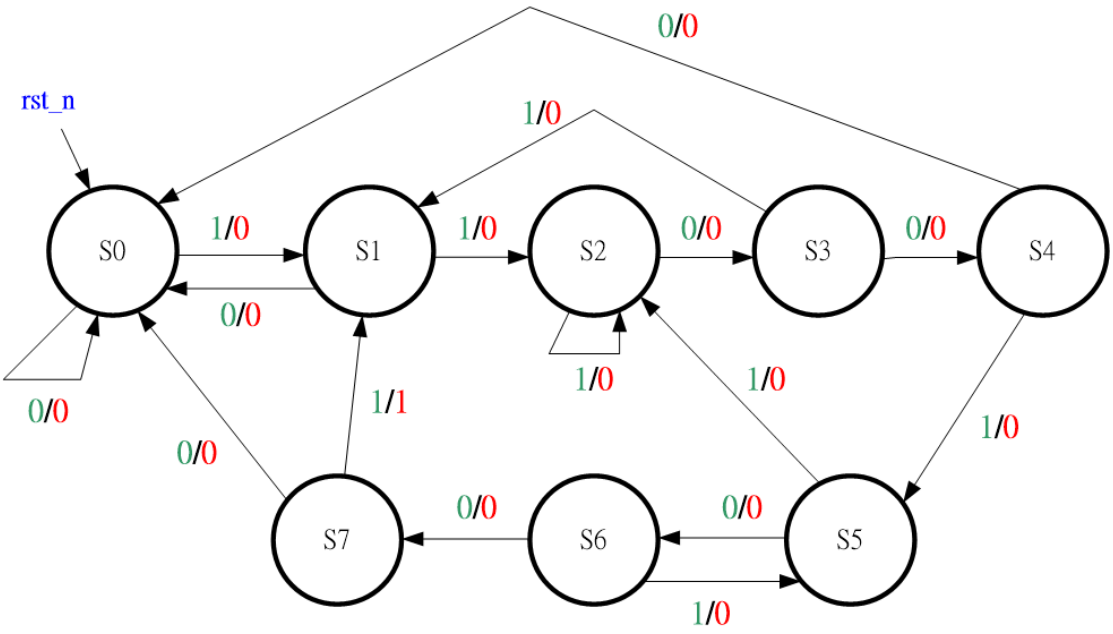


sliding window sequence detector 要偵測的 sequence 是 1100(10)+01，中間的 10 可以重複，但至少要出現一次，並且可以重複偵測，只要在任何時候出現以上的組合都會 output 1。input 有 clk、rst_n、in，output 是 dec，module 內部有 state、nxt_state

來運作 mealy machine。一開始用一個 MUX 選擇如果 rst_n = 0 的話輸入 S0 到 DFF，反之輸入 nxt_state，接著 DFF 將 state 輸入到一個 decoder 判斷目前在哪個 state，最後再輸入到 MUX 由 in 判斷輸出 dec 的值與下個 cycle 的 nxt_state。

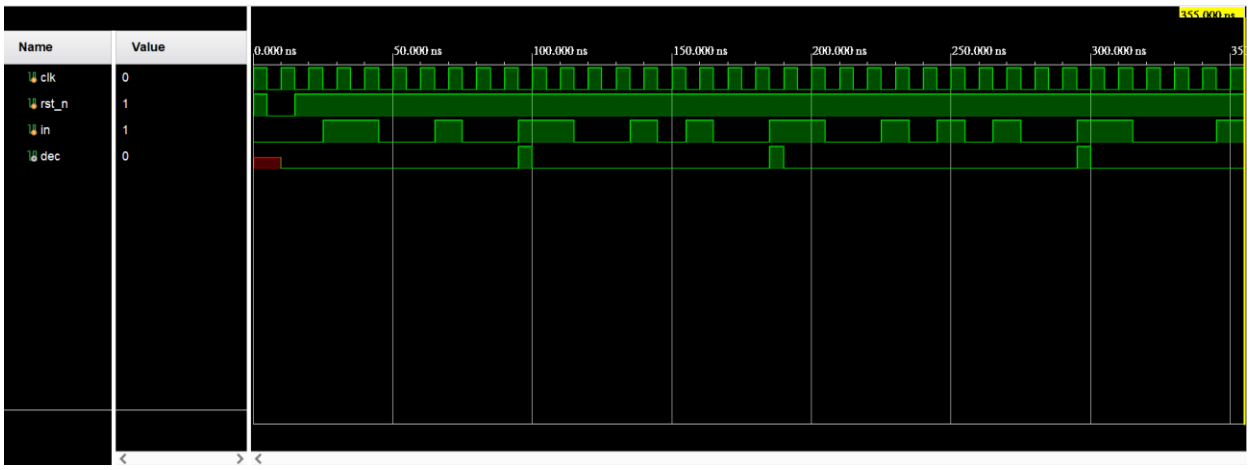
```
module Sliding_Window_Sequence_Detector (clk, rst_n, in, dec);
input clk, rst_n;
input in;
output reg dec;
reg [3:0] state;
reg [3:0] nxt_state;

parameter S0 = 3'b000;
parameter S1 = 3'b001;
parameter S2 = 3'b010;
parameter S3 = 3'b011;
parameter S4 = 3'b100;
parameter S5 = 3'b101;
parameter S6 = 3'b110;
parameter S7 = 3'b111;

always @ (posedge clk) begin
if(rst_n == 1'b0) state <= S0;
else state <= nxt_state;
end
```

state diagram:



rst_n == 0 時 state 會到 S0，然後開始接收 input sequence 並隨著 clk 變換 state，如果在連續的 cycle 接收到 1100(10)+01 的話會 output 1，不是以上的 sequence 則 output 0，而因為是 mealy machine，output 會由 state 和 input 決定，所以輸出是 asynchronous 的。如果沒有接收到 1100(10)+01 的數列，則會根據在先前的 cycle 輸入的數列來判斷要進入哪一個 state，因此可以在任何時候偵測數列(continuous detection)

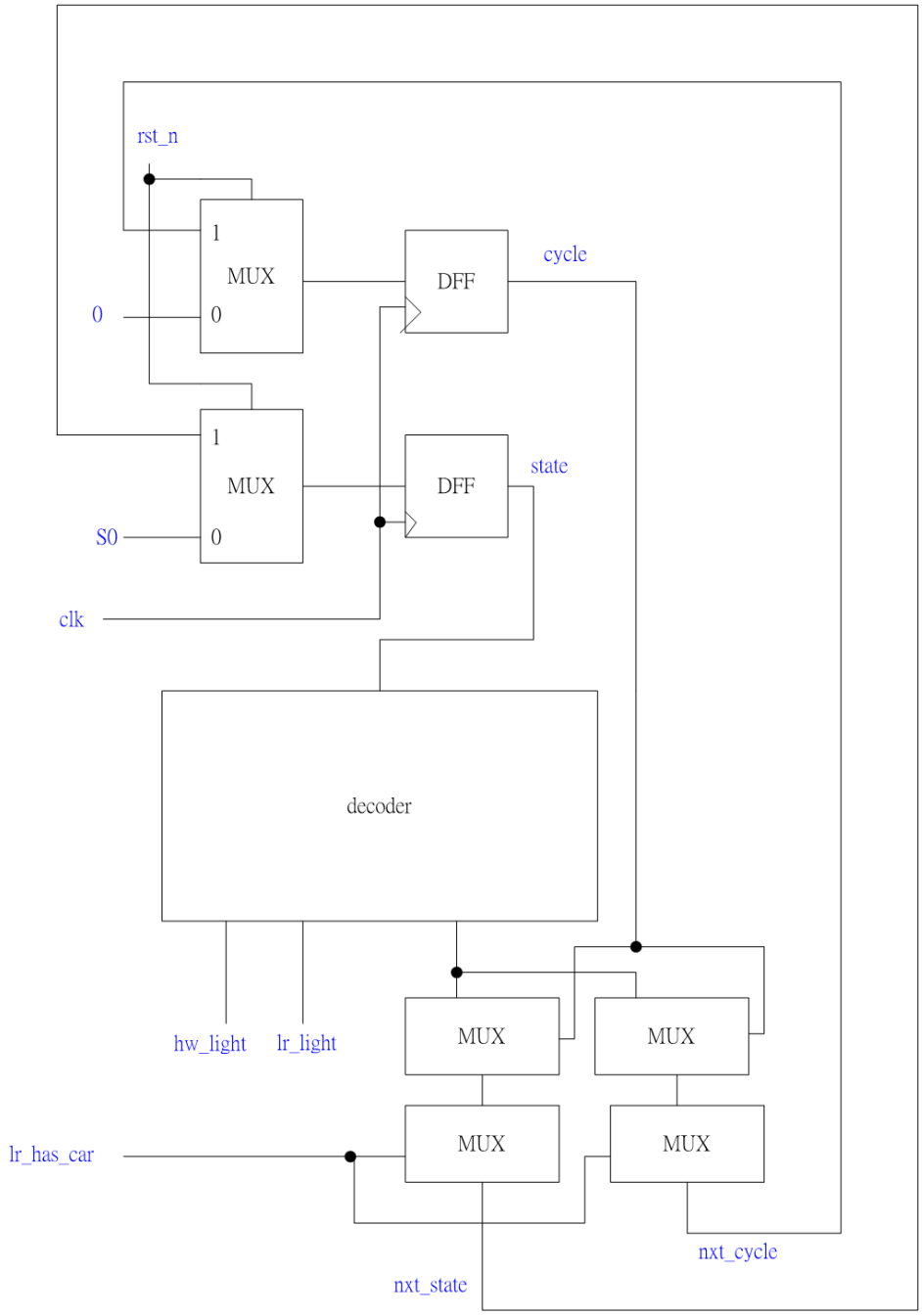
波形圖:



一開始輸入 1100(10)01，接收數列後 dec 輸出 1，再來輸入 1100(1010)01、1100(101010)01 確認數列中重複輸入 10 時是否 dec 也會輸出 1，最後輸入 110001，確認中間沒有輸入(10)時 dec 不會輸出 1。

2. Traffic light controller

Block diagram:



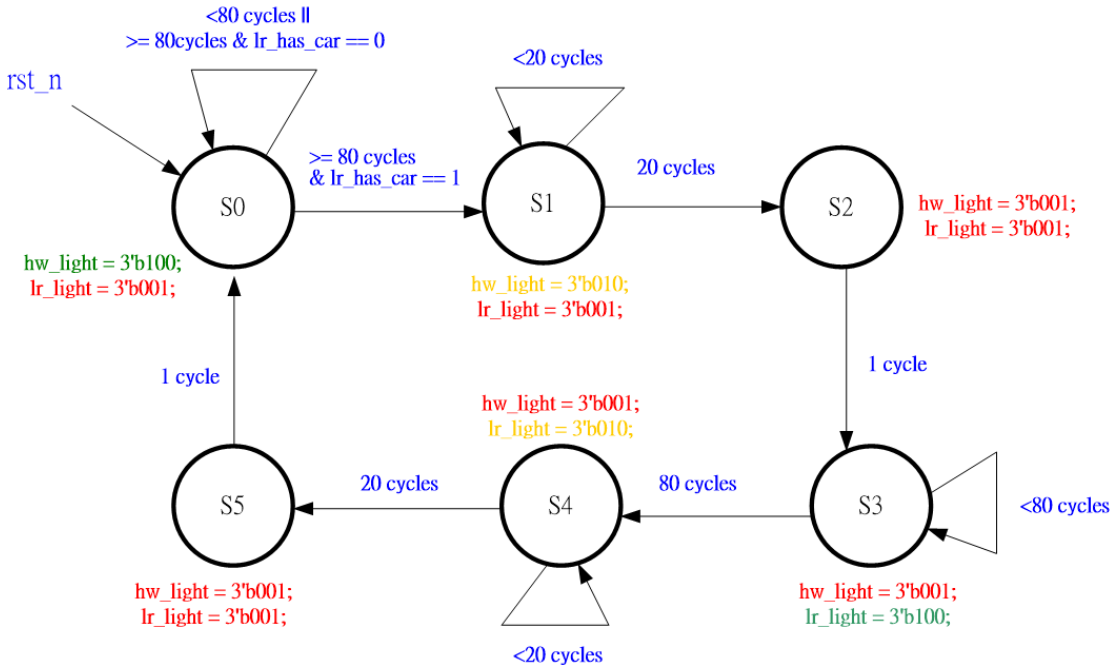
Traffic light controller 的 input 有 clk, rst_n 以及 lr_has_car(表示 local road 有車子經過)，output 有 hw_light 和 lr_light，用 3-bit 來表示 highway 和 local road 的紅綠燈(100

綠燈、010 黃燈、001 紅燈)。module 中有 cycle，nxt_cycle 表示在 state 中經過幾個 clk cycle，以及 state，nxt_state 表示 state machine 的狀態。

一開始用兩個 MUX 選擇，如果 rst_n = 0 的話輸入 S0 到 state 的 DFF、0 輸入到 cycle 的 DFF，反之輸入 nxt_state、nxt_cycle，接著 DFF 將 state 輸入到一個 decoder 判斷目前在哪個 state，然後根據 state 的狀態輸出 hw_light 和 lr_light，並使用兩層 MUX 和 lr_has_car 和 cycle 來決定 nxt_cycle 和 nxt_state。

```
module Traffic_Light_Controller (clk, rst_n, lr_has_car, hw_light, lr_light);
 input clk, rst_n;
 input lr_has_car;
 output reg [2:0] hw_light;
 output reg [2:0] lr_light;
 reg [6:0] cycle, nxt_cycle;
 reg [2:0] state, nxt_state;
 parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100, S5 = 3'b101;

always @ (posedge clk) begin
if (rst_n == 1'b0) begin
     cycle <= 7'd0;
     state <= S0;
     end
else begin
     cycle <= nxt_cycle;
     state <= nxt_state;
     end
end
```

state diagram:



hw_light 和 lr_light 在每個 state 中改變 output，我們在 state 旁標註出他們的 output 所以及代表的顏色。一開始 reset 後 state 跑到 S0，hw_light 變成綠燈、lr_light 變成紅燈，如果 cycle < 80 則 cycle 會在每個 posedge clk + 1；如果 cycle = 80 則 cycle 不會再改變數值，會一直等到 lr_has_car = 1 時 state 就會變成 S1，並把 cycle 歸零。

```
S0: begin
    hw_light = 3'b100;
    lr_light = 3'b001;
    if(cycle == 7'd80) begin
        if(lr_has_car) begin
            nxt_cycle = 7'd0;
            nxt_state = S1;
            end
        else begin
            nxt_cycle = cycle;
            nxt_state = S0;
            end
        end
    else begin
        nxt_cycle = cycle + 7'd1;
        nxt_state = S0;
        end
    end
```

State = S1 時，hw_light 變成黃燈、lr_light 變成紅燈，cycle 會

在每個 posedge clk + 1 直到 cycle = 20，state 就會變成 S2。

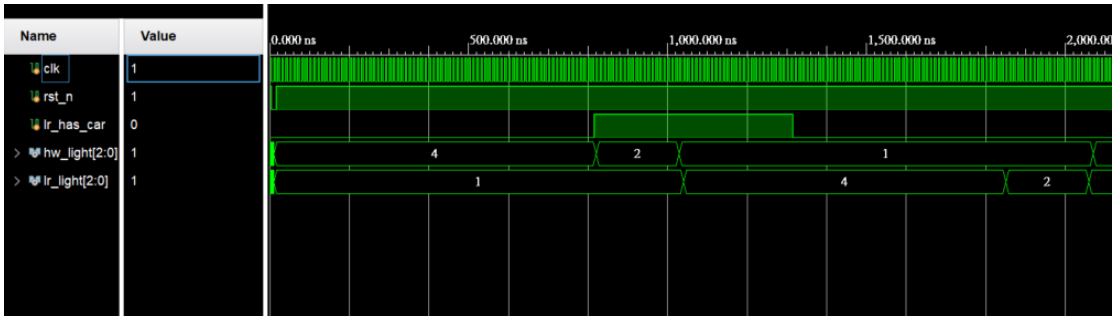
```
S1: begin
    hw_light = 3'b010;
    lr_light = 3'b001;
    if(cycle == 7'd20) begin
        nxt_cycle = 7'd0;
        nxt_state = S2;
        end
    else begin
        nxt_cycle = cycle + 7'd1;
        nxt_state = S1;
        end
    end
```

State = S2 時，hw_light、lr_light 都變成紅燈，經過一個 cycle 後 state 就會變成 S3。

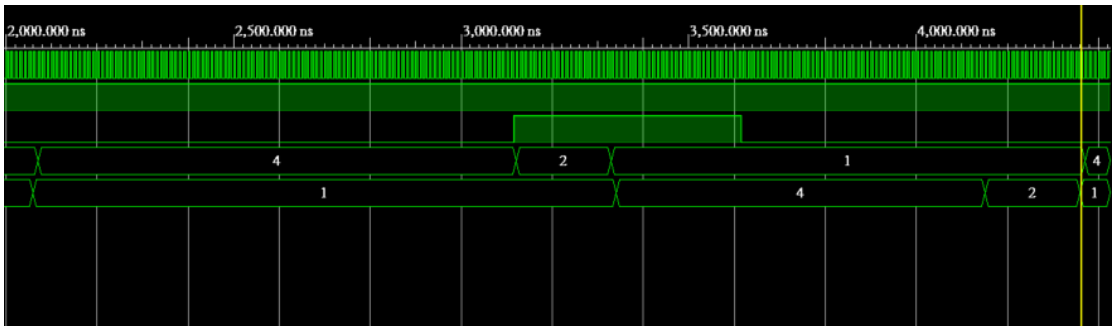
```
S2: begin
    hw_light = 3'b001;
    lr_light = 3'b001;
    nxt_state = S3;
    nxt_cycle = 7'd0;
    end
```

以此類推，S3~S5 都是類似的運作方式。

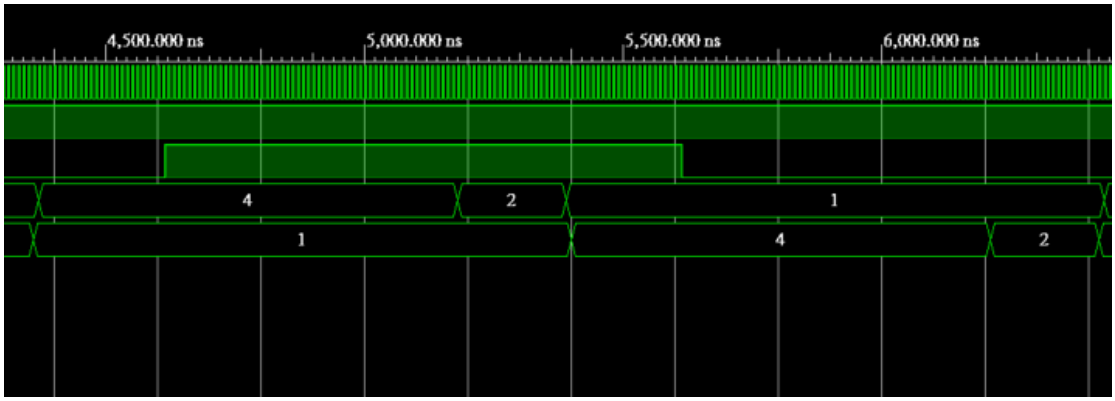
波形圖：



一開始嘗試在 80 個 cycles 後輸入 lr_has_car = 1，觀察 state transition，以及 hw_light 及 lr_light 轉換的時間是否符合，



回到 S0 後，經過超過 80 cycles 才輸入 lr_has_car = 1，觀察各狀態是否正確，



再次回到 S0 後，提早輸入 lr_has_car = 1，觀察 S0 是否是否保持 80 cycles 才變到下個 state。

3.Greatest Common Divisor

GCD 的功用是計算兩個 16 bit input a 和 b 的最大公因數。GCD 有 3 個 state，WAIT、CAL、和 FINISH，並且兩個 output done 和 gcd 是隨著 state 的改變而改變，因此是一個 Moore Machine。

- Code Detail:

    1. Sequential Block:

```
always@(posedge clk)begin
    if(rst_n==1'b0)begin
        state <= WAIT;
        count <= 2'b00;
    end
    else begin
        state <= next_state;
        count <= next_count;
        in_a <= next_in_a;
        in_b <= next_in_b;
    end
end
```

在 positive edge 更新 state、count、in_a、in_b。state 處理的是所屬哪一個 state。count 處理的是在進入 FINISH state，我們要計算 2 個 cycle 才能回到 WAIT state。因此我們需要知道 count 的值。in_a、in_b 處理的是我們在 WAIT state 我們要讀入的 input a 和 b，以及在 CAL state 的 1 個 cycle 只會執行一次減法。因此 in_a、in_b 不但可以知道輾轉相除法執行到時麼程度，還可以符合 spec 中說一個 cycle 執行 1 次減法運算。

2. Combinational Block

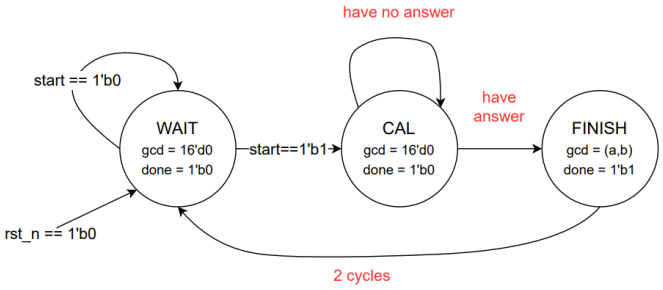
```
always@(*)begin
    case(state)
        WAIT:begin
            gcd = 16'd0;
            done = 1'b0;
            next_count = 2'b00;
            if(start==1'b0)begin
                next_state = WAIT;
            end
            else begin
                next_state = CAL;
                next_in_a = a;
                next_in_b = b;
            end
        end
        CAL:begin
            gcd = 16'd0;
            done = 1'b0;
            next_count = 2'b00;
            if(in_a == 16'd0)begin
                next_state = FINISH;
                ans = in_b;
            end
            else if (in_b == 16'd0)begin
                next_state = FINISH;
                ans = in_a;
            end
            else begin
                next_state = CAL;
                if(in_a > in_b)begin
                    next_in_a = in_a - in_b;
                end
                else begin
                    next_in_b = in_b - in_a;
                end
            end
        end
        FINISH:begin
            gcd = ans;
            done = 1'b1;
            if(count == 2'b01)begin
                next_state = WAIT;
                next_count = 2'b00;
            end
            else begin
                next_state = FINISH;
                next_count = count + 2'b01;
            end
```

WAIT state:等待 start == 1’b1，並開始讀入兩個 input a 和 b，並進入下一個 state CAL。若 start == 1’b0，還是在 WAIT state。Output gcd = 16’d0、done = 1’b0。

CAL state:利用輾轉相除法計算 a 和 b 的最大公因數(輾轉相除法:比較兩數大小，較大的數字減掉較小的數字並更新較大的數為這個值，直到其中一個數為 0，另一個數及是答案。)若有答案，把答案存入變數 ans 並進入下一個 state FINISH。沒有的話，還是在 CAL state。Output gcd = 16’d0、done = 1’b0。

FINISH state:在這個 FINISH state 持續 2 個 cycle 並把 output gcd 為所計算的最大公因數 done 則為 1’b1，2 個 cycle 之後回到 WAIT state。

State diagram:



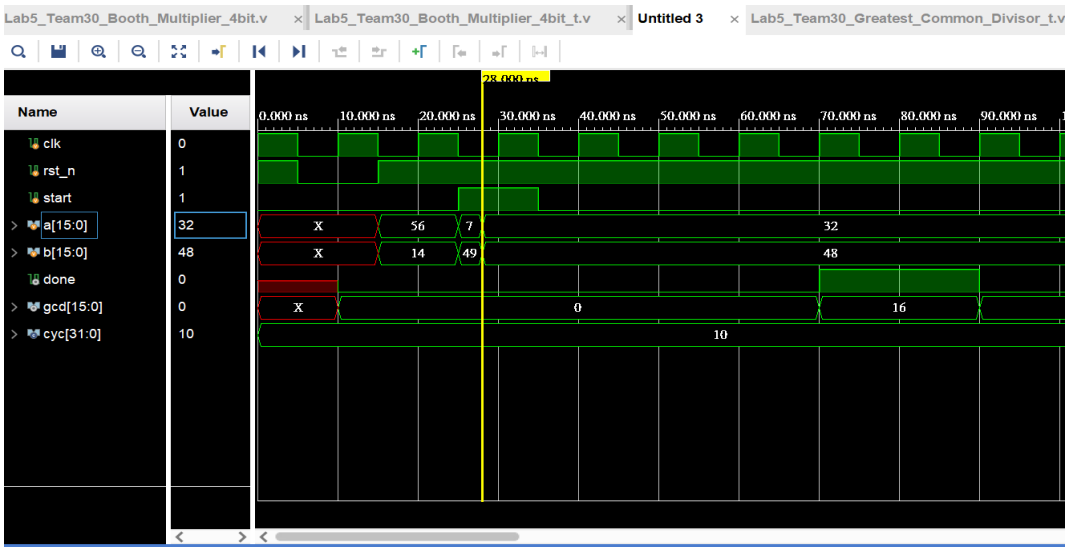
- Testbench

我先想幾個可能會有的錯誤，並針對可能的錯誤設計 testbench。

可能有的錯誤:

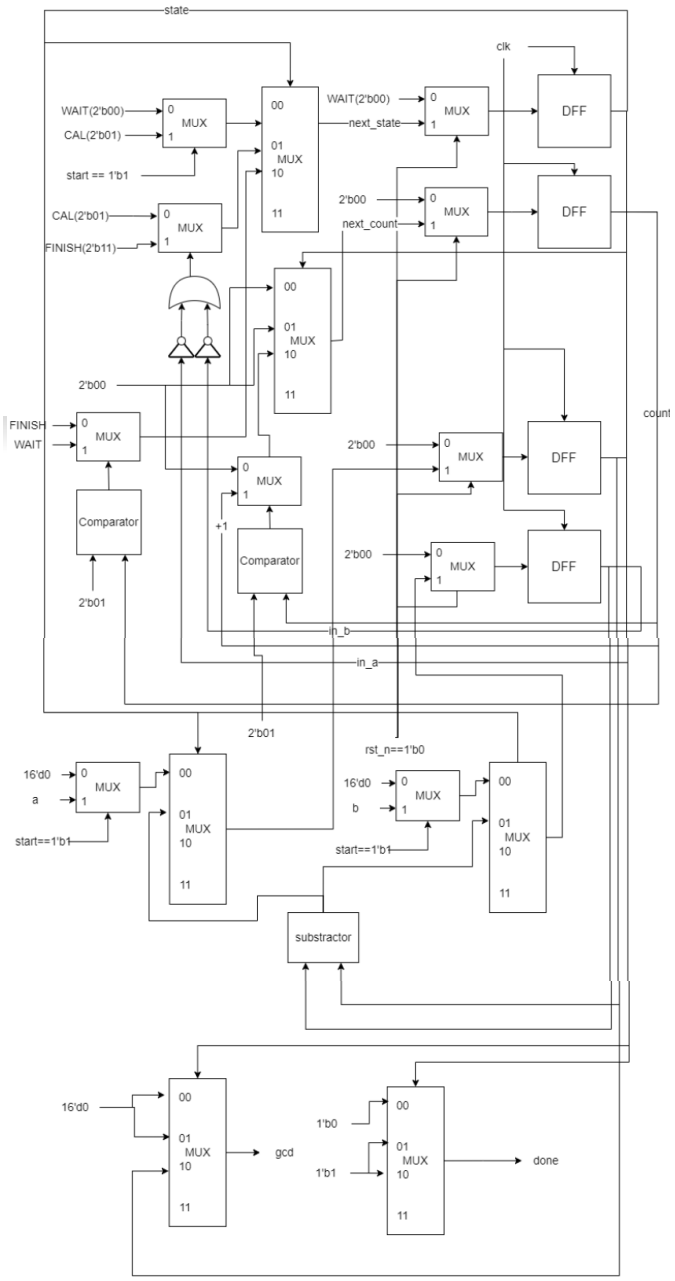
1. 在 WAIT state，當 start==1’b1，spec 上說 input a 和 b 可能會改變。因此確認是否有正確讀入 input。
2. Output gcd 和 done 要隨著 state 改變
3. FINISH state 只會持續 2 個 cycle
4. Output 是否正確

波形圖



可以看到在 WAIT state，當 start==1’b1，input a 和 b 本來是 7 和 49，但隨後改成 32 和 48。所以應該讀入 32 和 48 而不是 7 和 49。再來看到在 70ns 時，進入 FINISH state，可以看到 output 確實是隨著 state 改變而改變，而且只維持 2 個 cycle，結果也正確(32 和 48 的最大公因數為 16)。

- Block diagram



4.Booth multiplier

Booth multiplier 的功用是計算兩個 4 bit 2’s complement 相乘的值

- Code detail

    1. Sequential Block

```
always@(posedge clk)begin
    if(!rst_n)begin
        state <= WAIT;
        count <= 2'b00;
    end
    else begin
        state <= next_state;
        count <= next_count;
        P <= next_P;
    end
end
```

在 positive edge 更新 state、count、P。state 處理的是所屬哪一個 state。count 處理的是 CAL state 中，我們需要 4 個 clock cycle 就會進入下一個 state FINISH。因此我們需要知道 count 的值才知道何時進入下一個 state。因為兩個 input a、b 都是 4 bit 2’s complement，所以我們只需要 4 個 clock cycle。P 處理的是我們在 CAL state 的運算是如何進行的並且 1 個 cycle 只會一次運算。

    2. Combinational Block

```
always@(*)begin
    case(state)
        WAIT:begin
            p = 8'd0;
            next_count = 2'b00;
            if(start)begin
                next_state = CAL;
                A = 10'd0;
                A[9:5] = a;
                S = 10'd0;
                S[9:5] = -a;
                next_P = 10'd0;
                next_P[4:1] = b;
            end
            else next_state = WAIT;
        end
```

WAIT state: 等待 start == 1’b1，並開始讀入兩個 4 bit input a 和 b，變數 A 的第 6 個 bit 到第 10 個 bit 設值為 a、變數 S 的第第 6 個 bit 到第 10 個 bit 設值為 a 的 2’s complement、變數 next_P 的第 2 個 bit 到第 5 個 bit 設值為 b 其他補上 0，並進入下一個 state CAL。若 start == 1’b0，還是在 WAIT state。Output p = 16’d0。

```
        CAL:begin
            p = 8'd0;
            case(P[1:0])
                2'b00: next_P = P >>> 1;
                2'b01:begin
                    next_P = (P + A)>>>1;
                end
                2'b10:begin
                    next_P = (P + S)>>>1;
                end
                2'b11:next_P = P >>> 1;
            endcase
            if(count == 2'b11)begin
                next_count = 2'b00;
                next_state = FINISH;
            end
            else begin
                next_count = count + 2'b01;
                next_state = CAL;
            end
        end
```

CAL state:如果 P 的最右邊兩個 bit 是 01，next_P 的值更新為 P + S，如果 P 的最右邊兩個 bit 是 10，next_P 更新為 P + A，如果 P 的最右邊兩個 bit 是 00 或 11，維持原來的 P 也就是 next_P 設值為 P。判斷為後將所有的 bit 往右移，補上的 bit(MSB)則是依據是複數還是正數來補上 1 或 0。

```
            FINISH:begin
                p = P[8:1];
                next_count = 2'b00;
                next_state = WAIT;
            end
        endcase
```

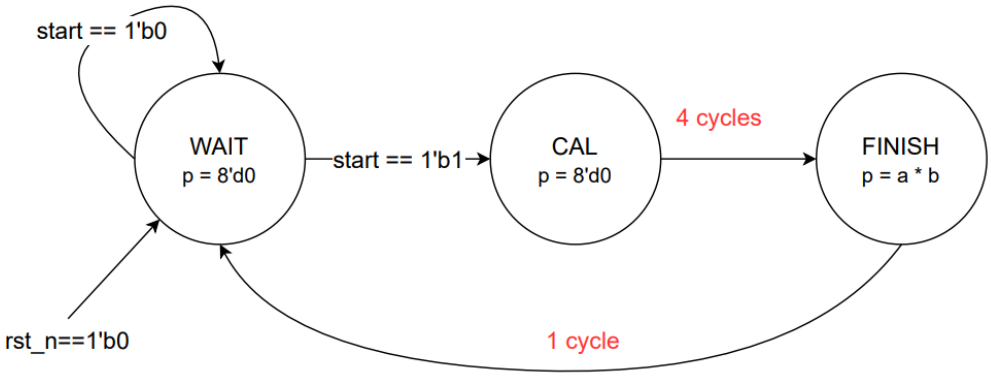
FINISH state: FINISH state 持續 4 個 cycle 就會進入下一個 state WAIT state。FINISH state 期間，output p 為算出的答案(兩數相乘)。

- Testbench

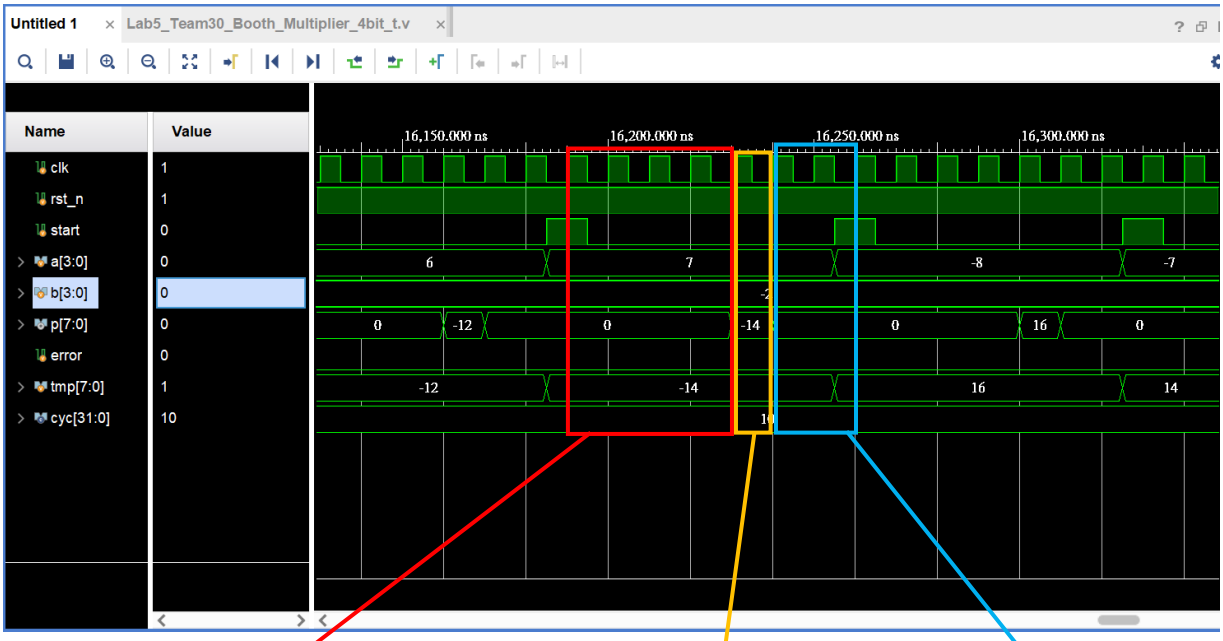
```
    Booth_Multiplier_4bit bm(clk, rst_n, start, a, b, p);

    initial begin
        @(negedge clk)
        error = 1'b0;
        rst_n = 1'b0;
        @(negedge clk)
        rst_n = 1'b1;
        a = 4'b0000;
        b = 4'b0000;
        repeat(2**4)begin
            repeat(2**4)begin
                tmp = a*b;
                $display("ans : %d", p);
                start = 1'b1;
                #(cyc);
                start = 1'b0;
                #(4*cyc)
                if(tmp!==p)error = 1'b1;
                else error = 1'b0;
                #(2*cyc);
                a = a + 8'd1;
            end
            b = b + 8'd1;
        end
    #(cyc);
```

因為 2 個 input 都只有 4 個 bit，因此我便把所有可能的 input 列舉出來。其中，我多設 2 個變數 error 和 tmp 用來測試所計算相乘的結果是否正確。並且我用$display 好用來印出我 multiplier 的結果。首先，我用兩層 repeat 來一一測試每個 input 組合。內層是美執行一次 a 加 1，外層是美執行一次 b 加 1。每次測試新的組合，我先計算新的組合相乘的結果並把結果存入我設的變數 tmp，error 設為 1’b0，並且 start==1’b1 維持 1 個 cycle。而因為我們知道計算需要 4 個 clock cycle，因此經過 4 個 clock cycle 之後，測試 tmp 是否等於 booth multiplier 所計算出的結果。如果不相等，將 error 拉成 1’b1，否則 error 為 1’b0。

- State diagram



- 波形圖(部分)

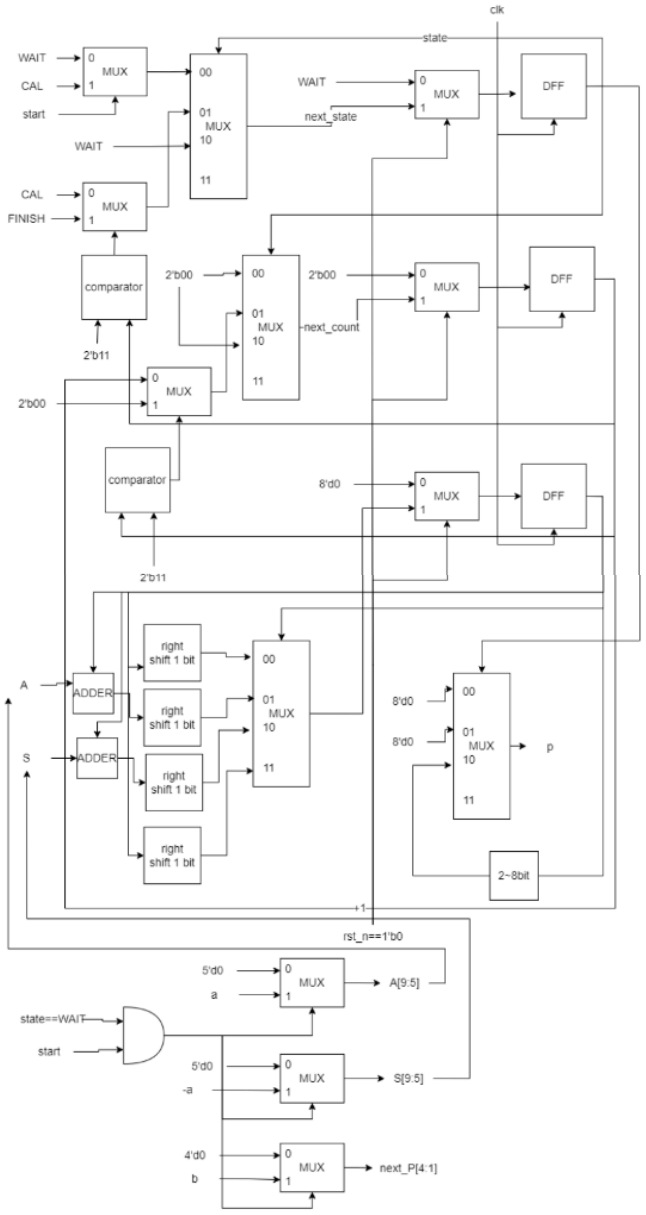


CAL state:經過 4 個 clock cycle 之後，進入下一個 state。Output p 為 7'd0。

FINISH state:經過 1 個 clock cycle 之後，進入下一個 state。Output 為 booth multiplier 所計算的結果。

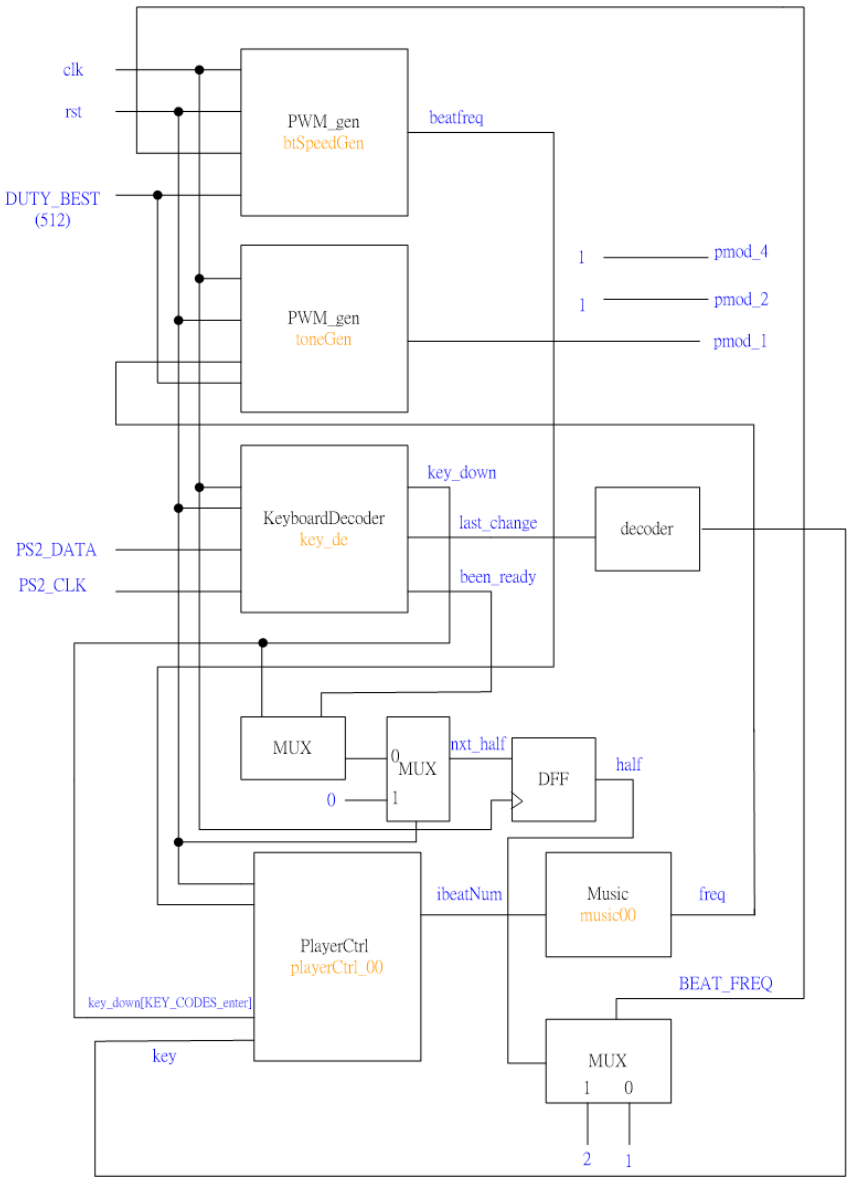
WAIT state:在 start==1'b1 後下一個 clock cycle 進入下一個 state。Output p 為 7'd0。

- Block diagram


clk
state
WAIT
CAL
start
MUX
00
01
10
11
WAIT
next_state
DFF
CAL
FINISH
comparator
2'b00
next_count
2'b11
8'd0
ADDER
right shift 1 bit
A
S
p
2~8bit
+1
rst_n==1'b0
5'd0
a
A[9:5]
state==WAIT
start
-a
S[9:5]
4'd0
b
next_P[4:1]

5. FPGA 1 - Mixed keyboard and audio modules together

Block diagram:



此 module 的 input 有 clk、rst、PS2_DATA 以及 PS2_CLK(來自 keyboard 的訊號)，output 是 PS2_DATA 和 PS2_CLK ，以及 pmod_1、pmod_2，以及 pmod_4，連接到 audio 輸出音樂。在 submodule 方面我們使用一些在這次 basic question 中助教上傳的

modules，我們未更動 PWM_gen 和 KeyboardDecoder 直接拿來使用，而 PlayerCtrl 和 Music 我們更動了一些功能來符合題目的規定。在 module 中我們使用兩個 MUX 及一個 DFF，第一個 MUX 是一個 combinational always block，以 been_valid 和 key_down 作為 selection bits，如果 been_ready && enter 按下，則 half 設為 0；如果 been_ready && r 鍵按下，則 half 反轉，如果以上條件都沒有符合則 half 不變。接下來的 MUX 和 DFF 在 seqential block 中，如果 rst = 1 則 half 設為 0，反之則 half 在 posedge clk 改變值。

```
always @ (posedge clk, posedge rst) begin
if (rst)
    half <= 1'b0;
else
    half <= nxt_half;
end

always @ (*) begin
if(been_ready && key_down[KEY_CODES_enter])
    nxt_half <= 1'b0;
else begin
    if(been_ready && key_down[KEY_CODES_R])
        nxt_half = ~half;
    else
        nxt_half = half;
    end
end
```

half 接下來再連接到一個 MUX，以 half 作為 selection bit，來選擇音樂的頻率(1HZ or 2HZ)，

```
always @ (*) begin
if(half)
    BEAT_FREQ = 32'd2; // 0.5 sec
else
    BEAT_FREQ = 32'd1; // 1 sec
end
```

另外還有一個 decoder 可以把 last_change[7:0]轉換成 key[2:0]提供給 PlayerCtrl 做判斷。

```
parameter [8:0] KEY_CODES_W = 9'b0_0001_1101;    // W => 1D
parameter [8:0] KEY_CODES_S = 9'b0_0001_1011;    // S=> 1B
parameter [8:0] KEY_CODES_R = 9'b0_0010_1101;    // R => 2D
parameter [8:0] KEY_CODES_enter = 9'b0_0101_1010;    // enter => 5A
```

```
always @ (*) begin
        case (last_change)
            KEY_CODES_W : key = 3'b000;
            KEY_CODES_S : key = 3'b001;
            KEY_CODES_R : key = 3'b010;
            KEY_CODES_enter : key = 3'b011;
            default       : key = 3'b100;
        endcase
    end
```

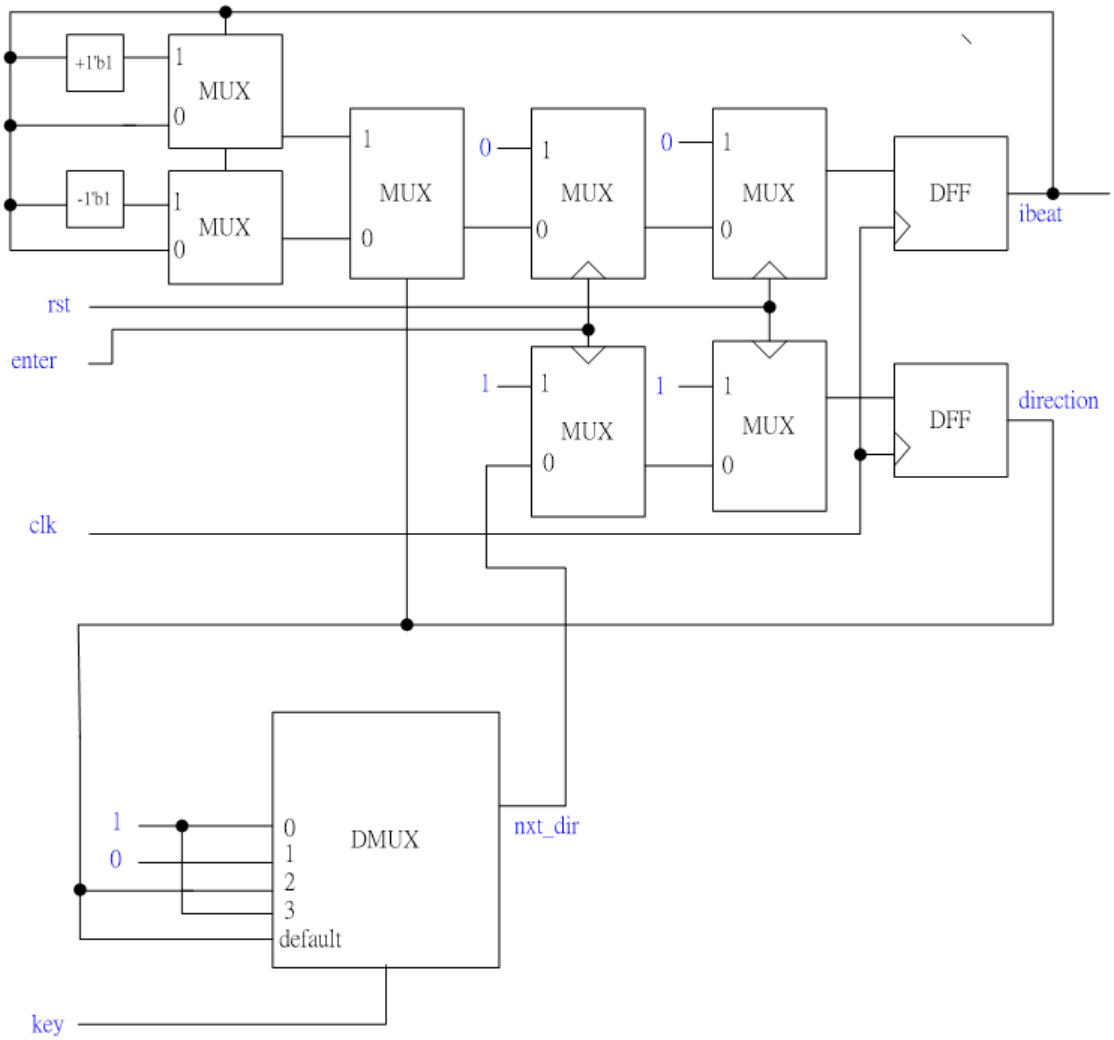
- MUSIC

```
module Music (
    input [4:0] ibeatNum,
    output reg [31:0] tone
);

always @(*) begin
    case (ibeatNum)     // 1/4 beat
        5'd0 : tone = `C4;
        5'd1 : tone = `D4;
        5'd2 : tone = `E4;
        5'd3 : tone = `F4;
        5'd4 : tone = `G4;
        5'd5 : tone = `A4;
        5'd6 : tone = `B4;
        5'd7 : tone = `C5;
        5'd8 : tone = `D5;
        5'd9 : tone = `E5;
        5'd10 : tone = `F5;
        5'd11 : tone = `G5;
        5'd12 : tone = `A5;
        5'd13 : tone = `B5;
```

我們在 code 的開頭 define 了 C4~C8 的頻率，因此在這個 module 中我們只需要用一個 case，根據 ibeatNum 排列出對應的 tone 即可。

- PlayerCtrl

Block diagram:



PlayerCtrl 的 input 有 clk, rst, enter, key, output 是 ibeat, module 中有 direction 表示 ibeat 每個 cycle 要增加或減少。

```
always @(posedge clk, posedge rst, posedge enter) begin
    if (rst)begin
        ibeat <= 0;
        direction <= 1'b1;
        end
    else begin
        if (enter) begin
            ibeat <= 0;
            direction <= 1'b1;
        end
        else begin
            direction <= nxt_dir;
            if (direction) begin
                if(ibeat < BEATLEAGTH)
                    ibeat <= ibeat + 1;
                else
                    ibeat <= ibeat;
                end
            else begin
                if(ibeat > 5'd0)
                    ibeat <= ibeat - 1;
                else
                    ibeat <= ibeat;
                end
            end
        end
    end
```

我們在 sequential always block 中用兩個 DFF 前面串上幾個 MUX 來進行判斷，這邊的 rst、enter 都是 asychronous reset，如果沒有要 reset 的話就會判斷 direction 如果是 1 的話表示音階要上升，就讓 ibeat 隨著 clk 增加直到 BEATLEAGTH，如果是 0 音階會下降，就讓 ibeat 減少直到 0。

```
always @ (*) begin
    case(key)
    3'b000:  // w
        nxt_dir = 1'b1;
    3'b001: // s
        nxt_dir = 1'b0;
    3'b010: // r
        nxt_dir = direction;
    3'b011: // enter
        nxt_dir = 1'b1;
    default:
        nxt_dir = direction;
    endcase
end
```

另外，在 combinational block 中我們使用一個 DMUX，以 key 做為 selection bits，key = 0 表示 w 鍵、1 為 s 鍵、2 為 r 鍵、3 為 enter 鍵。按下 w 和 enter 時 direction 會變成 1，按下 r 時 direction 不變。

6.Fpga 2 Vending machine

Vending machine 在 fpga 上模擬。在 fpga 上有 5 個功能按鈕，上面的按鈕代表 reset，左邊的按鈕代表投入 5 元，中間的按鈕代表投入 10 元，右邊的按鈕代表投入 50 元，下面的按鈕代表取消交易。鍵盤上有 4 個功能鍵，A 代表 75 元的咖啡，S 代表 50 元的可樂(坑錢!)，D 代表 30 元的烏龍茶，F 代表 25 元的水。

我設計的 vending machine 共有兩個 state 一個是 INSERT state，處理投錢進販賣機。另一個 state 是 MAKE_CHANGE state，處理如果有找零錢每一秒找 5 元。

- Code Detail

1. Finite State Machine (Sequential Part)

```
// finite state machine
always@(posedge clk)begin
    if(rst)begin
        money <= 8'd0;
        state <= INSERT;
    end
    else begin
        state <= next_state;
        if(state == MAKE_CHANGE)begin
            if(one_sec_dclk)money <= next_money;
            else money <= money;
        end
        else money <= next_money;
    end
end
```

state 代表目前的 state 是哪一個 state，並在 positive edge clock 更新 state。money 代表投入販賣機的錢，但在 state 是

MAKE_CHANGE 的時候比較特別，如果販賣機要找零錢，他會每過 1 秒找 5 元。因此我設計了一個 one second clock divider 來處理這種狀況，只有 one second clock divider 等於 1’b1 時會更新 money 的值。

2. One Second Clock Divider

```
// create one second clock divider
always@(posedge clk)begin
    if(state != MAKE_CHANGE)begin
        one_sec_count <= 27'd0;
        one_sec_dclk <= 1'b0;
    end
    else begin
        one_sec_count <= next_one_sec_count;
        one_sec_dclk <= next_one_sec_dclk;
    end
end

always@(*)begin
    if(one_sec_count == 27'd99999999)begin
        next_one_sec_count  = 27'd0;
        next_one_sec_dclk = 1'b1;
    end
    else begin
        next_one_sec_count = one_sec_count + 27'd1;
        next_one_sec_dclk = 1'b0;
    end
end
```

One Second Clock Divider 是由一個 27 bit counter 所組成，並由這個 counter 來控制 one_sec_dclk 的訊號。因為我們希望在進入 MAKE_CHANGE state 的時候才會開始 1 秒 1 秒找錢，所以這個 One Second Clock Divider 我設計在進入 MAKE_CHANGE state 才會有作用，不是在 MAKE_CHANGE state one_sec_dclk 和 計算 counter 的變數 one_sec_count 設為 0。

3. Finite State Machine (Combinational Part)

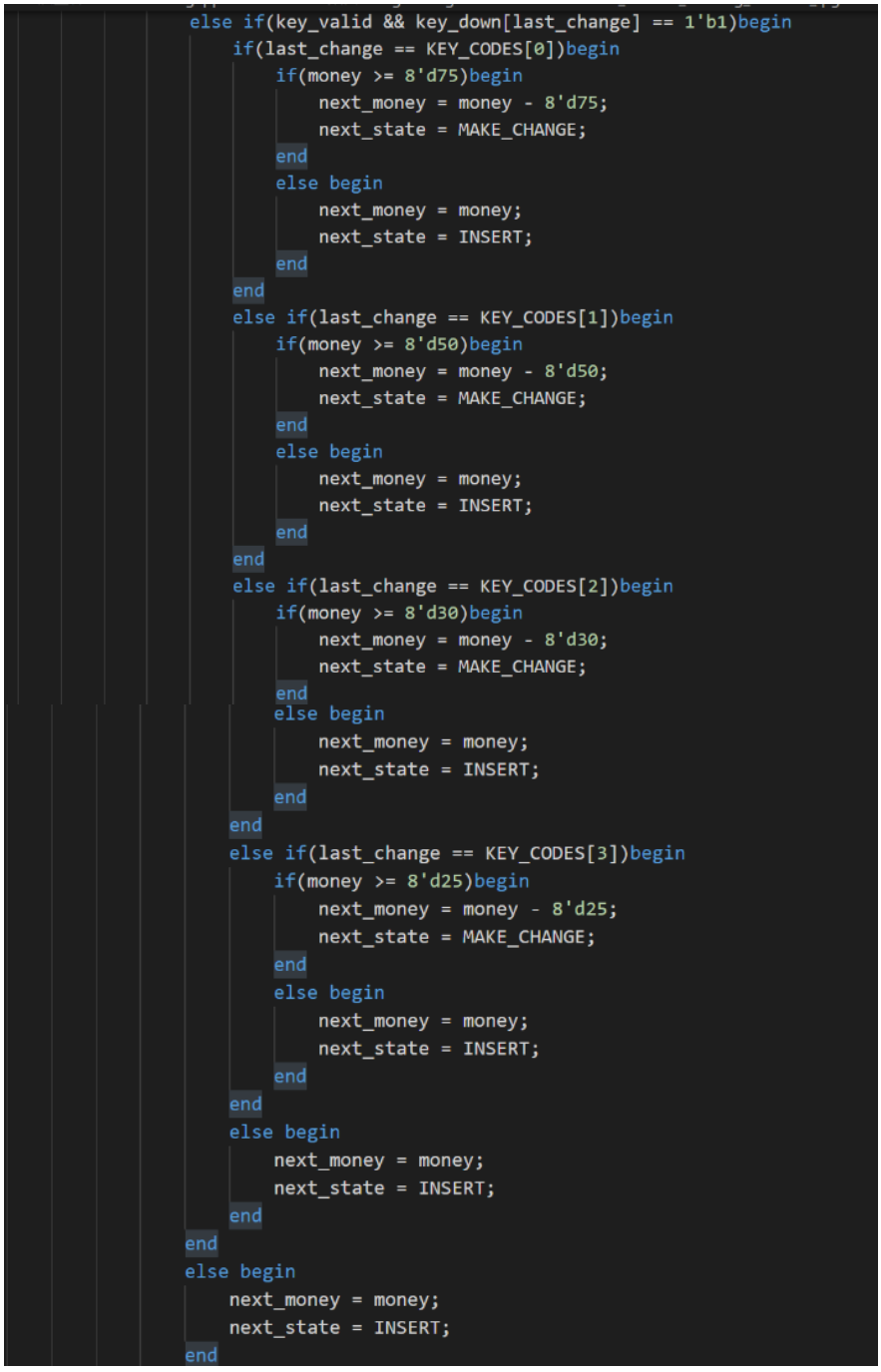
在 Finite state machine combinational block，我們只需要著重於 money 和 state。因此我便把會把 money 和 state 這 2 個變數更新的情況列舉出來。

INSERT state:

```
always@(*)begin
    case(state)
        INSERT:begin
            if(insert_five)begin
                if(money + 8'd5 <= 8'd100)next_money = money + 8'd5;
                else next_money = money;
                next_state = INSERT;
            end
            else if(insert_ten)begin
                if(money + 8'd10 <= 8'd100)next_money = money + 8'd10;
                else next_money = money;
                next_state = INSERT;
            end
            else if(insert_fifty)begin
                if(money + 8'd50 <= 8'd100)next_money = money + 8'd50;
                else next_money = money;
                next_state = INSERT;
            end
            else if(cancel)begin
                next_money = money;
                if(money == 8'd0)next_state = INSERT;
                else next_state = MAKE_CHANGE;
            end
```

上面的 code 列舉出在 INSERT state 5 個按鈕按下的個別的情況。insert_five 表示投入 5 元的按鈕經過處理的訊號、insert_ten 表示投入 10 元的按鈕經過處理的訊號、insert_fifty 表示投入 50 元的按鈕經過處理的訊號、cancel 表示取消交易的按鈕經過處理的訊號。因為根據 spec，我們最多只能投 100 元。因此我們需要處理如果投錢的額度超過 100 元的情況。這裡需要注意的是，money 應該設為幾個 bit。如果目前的錢是 100 元再按下投入 50 元的按鈕，判

斷式中會跟 100 元比較。因此我們應該設 money 為 8 個 bit，100+50 才不會 overflow。



```
else if(key_valid && key_down[last_change] == 1'b1)begin
    if(last_change == KEY_CODES[0])begin
        if(money >= 8'd75)begin
            next_money = money - 8'd75;
            next_state = MAKE_CHANGE;
        end
        else begin
            next_money = money;
            next_state = INSERT;
        end
    end
    else if(last_change == KEY_CODES[1])begin
        if(money >= 8'd50)begin
            next_money = money - 8'd50;
            next_state = MAKE_CHANGE;
        end
        else begin
            next_money = money;
            next_state = INSERT;
        end
    end
    else if(last_change == KEY_CODES[2])begin
        if(money >= 8'd30)begin
            next_money = money - 8'd30;
            next_state = MAKE_CHANGE;
        end
        else begin
            next_money = money;
            next_state = INSERT;
        end
    end
    else if(last_change == KEY_CODES[3])begin
        if(money >= 8'd25)begin
            next_money = money - 8'd25;
            next_state = MAKE_CHANGE;
        end
        else begin
            next_money = money;
            next_state = INSERT;
        end
    end
    else begin
        next_money = money;
        next_state = INSERT;
    end
end
else begin
    next_money = money;
    next_state = INSERT;
end
```

上面的 code 處理鍵盤按下 A、S、D、F 四種情形，如果符合其中一一種情形，會進入 state INSERT。並且根據所按下的鍵扣掉所對應

的錢。

MAKE_CHANGE state:

```
MAKE_CHANGE:begin
    if(money > 0)begin
        next_money = money - 8'd5;
        next_state = MAKE_CHANGE;
    end
    else begin
        next_money = 8'd0;
        next_state = INSERT;
    end
end
endcase
```

上面的 code 處理在 MAKE_CHANGE state，如果目前的錢大於 0，就要找錢，next_state 也還是 MAKE_CHANGE。如果不是的話，next_state 要回到 INSERT state。

4.LED:

```
always@(*)begin
    if(state == INSERT)begin
        if(money >= 8'd75)LED = 4'b1111;
        else if(money >= 8'd50)LED = 4'b0111;
        else if(money >= 8'd30)LED = 4'b0011;
        else if(money >= 8'd25)LED = 4'b0001;
        else LED = 4'b0000;
    end
    else LED = 4'b0000;
end
```

上面處理 LED 燈何時亮。我設計的是在 INSERT state LED 才會根據現有的錢來兩那些燈。LED 亮代表可以買那一種飲料，從左到右代表咖啡、可樂、烏龍茶和水。在 INSERT state，如果目前的錢大於 75 元，可以買任何一種飲料，如果目前的錢大於 50 元，可以買除

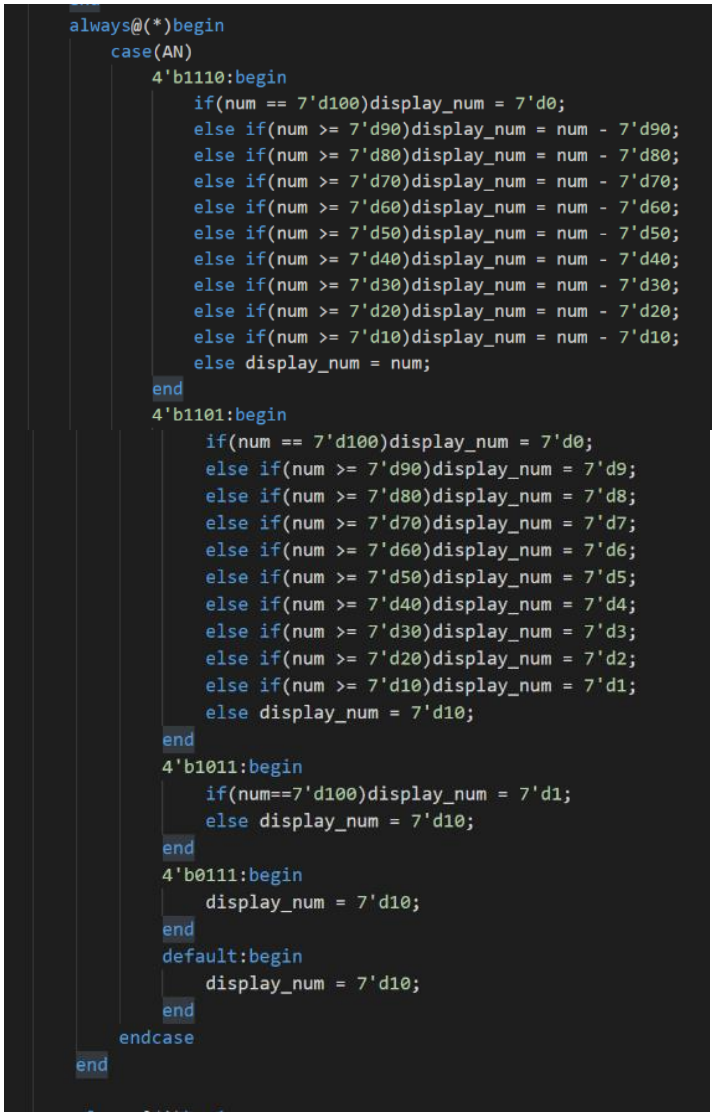
了咖啡之外任何一種飲料，依飲料價格類推。如果不再 INSERT state 的話，LED 不會亮，代表此狀態不能買飲料。

5. Seven Segment:

SevenSegment 這個 module 要顯示投入販賣機的錢有多少。因此我們需要 input 之前在 top module 寫的 money 當作 SevenSegment 的 input(變數 num)。

```
module SevenSegment(
    input clk,
    input rst,
    input [6:0] num,
    output reg [6:0] Seven_Segment,
    output reg [3:0] AN
    );
    reg [18:0] clock_divider;
    reg [3:0] display_num;
    wire [1:0] activating_LED;

    always@(posedge clk)begin
        if(rst)begin
            clock_divider <= 19'd0;
        end
        else begin
            clock_divider <= clock_divider + 19'd1;
        end
    end
    assign activating_LED = clock_divider[18:17];
    always@(*)begin
        case(activating_LED)
            2'b00:AN = 4'b1110;
            2'b01:AN = 4'b1101;
            2'b10:AN = 4'b1011;
            2'b11:AN = 4'b0111;
```

SevenSegment 是主要是由 counter 所構成，並利用 counter 所形成的訊號來控制 4 個 7 段顯示器顯示的順序。我設計的 counter 是 19 bit 最左邊兩個 bit 是來控制四個 7 段顯示器(從左到右 4 個 7 段顯示器代表 11、10、01、00)。Counter 是隨著 fpga 的 clock 來累加，而 17 個 bit 最多可以累加到 2 個 17 次方，剛好是接近 1 毫秒，也就是人類視覺暫留的時間。

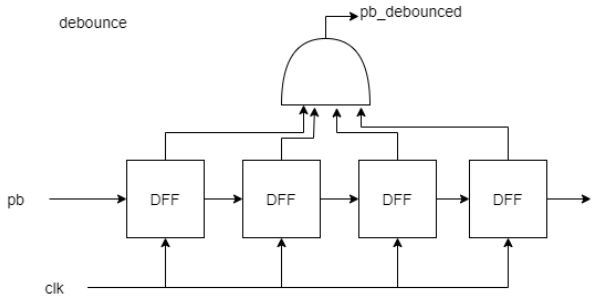


```
always@(*)begin
    case(AN)
        4'b1110:begin
            if(num == 7'd100)display_num = 7'd0;
            else if(num >= 7'd90)display_num = num - 7'd90;
            else if(num >= 7'd80)display_num = num - 7'd80;
            else if(num >= 7'd70)display_num = num - 7'd70;
            else if(num >= 7'd60)display_num = num - 7'd60;
            else if(num >= 7'd50)display_num = num - 7'd50;
            else if(num >= 7'd40)display_num = num - 7'd40;
            else if(num >= 7'd30)display_num = num - 7'd30;
            else if(num >= 7'd20)display_num = num - 7'd20;
            else if(num >= 7'd10)display_num = num - 7'd10;
            else display_num = num;
        end
        4'b1101:begin
            if(num == 7'd100)display_num = 7'd0;
            else if(num >= 7'd90)display_num = 7'd9;
            else if(num >= 7'd80)display_num = 7'd8;
            else if(num >= 7'd70)display_num = 7'd7;
            else if(num >= 7'd60)display_num = 7'd6;
            else if(num >= 7'd50)display_num = 7'd5;
            else if(num >= 7'd40)display_num = 7'd4;
            else if(num >= 7'd30)display_num = 7'd3;
            else if(num >= 7'd20)display_num = 7'd2;
            else if(num >= 7'd10)display_num = 7'd1;
            else display_num = 7'd10;
        end
        4'b1011:begin
            if(num==7'd100)display_num = 7'd1;
            else display_num = 7'd10;
        end
        4'b0111:begin
            display_num = 7'd10;
        end
        default:begin
            display_num = 7'd10;
        end
    endcase
end
```

接下來，根據是哪一個 7 段顯示器是可以亮的，我們依據 num 來判斷要顯示的數字為何。例如，如果是最右邊的 7 段顯示器是可以亮的，我們要依據 num 的個位數是甚麼數字來決定要顯示什麼。接下來依此類推十位數及百位數，也要注意如果 num 沒有十位數是不用顯示十位數的，在把條件列入進去。最後，display_num 代表要顯示的是什麼數字，我們再把對應的數字寫入七段顯示器的 7bit 就完成了。

Debounce circuit:

Block diagram:



Debounce circuit 由 1 個 combinational circuit 和 4 個 sequential circuit (4 個 D flip flop)所組成。

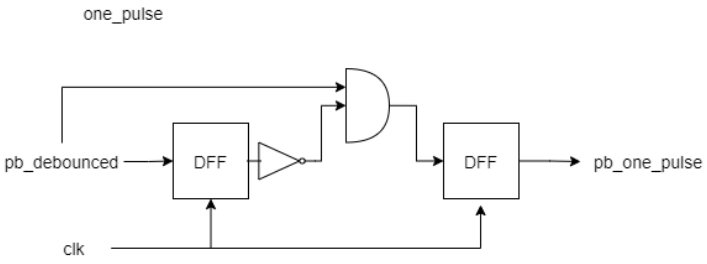
Combinational circuit 來判斷 4 個 d flip flop 是否都是 1，如果是 output pb_debounced 等於 1 反之為 0。

Sequential circuit 來儲存更新 d flip flop 的值，每一個 clock cycle 的 positive edge，input pb 餵給第一個 d flip flop，第一個 d flip flop 的值餵給第二個 d flip flop，依此類推。

```
module Debounce(
    input clk,
    input pb,
    output wire debounced
);
    reg [3:0]DFF;
    always@(posedge clk)begin
        DFF[3:1] <= DFF[2:0];
        DFF[0] <= pb;
    end
    assign debounced = (DFF == 4'b1111) ? 1'b1 : 1'b0
endmodule
```

One pulse circuit:

Block diagram:

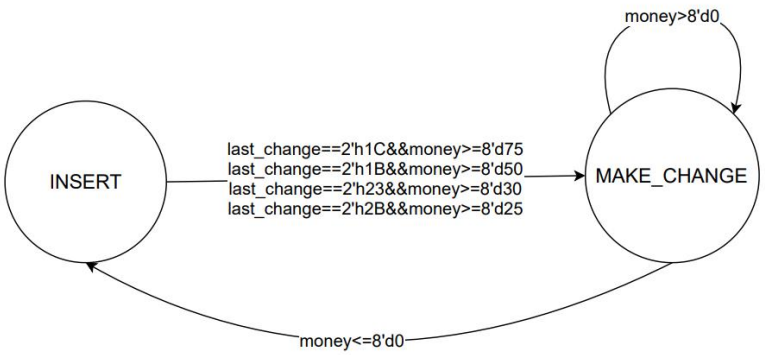


One pulse circuit 的作用是要 output 一個 clock cycle 的波。

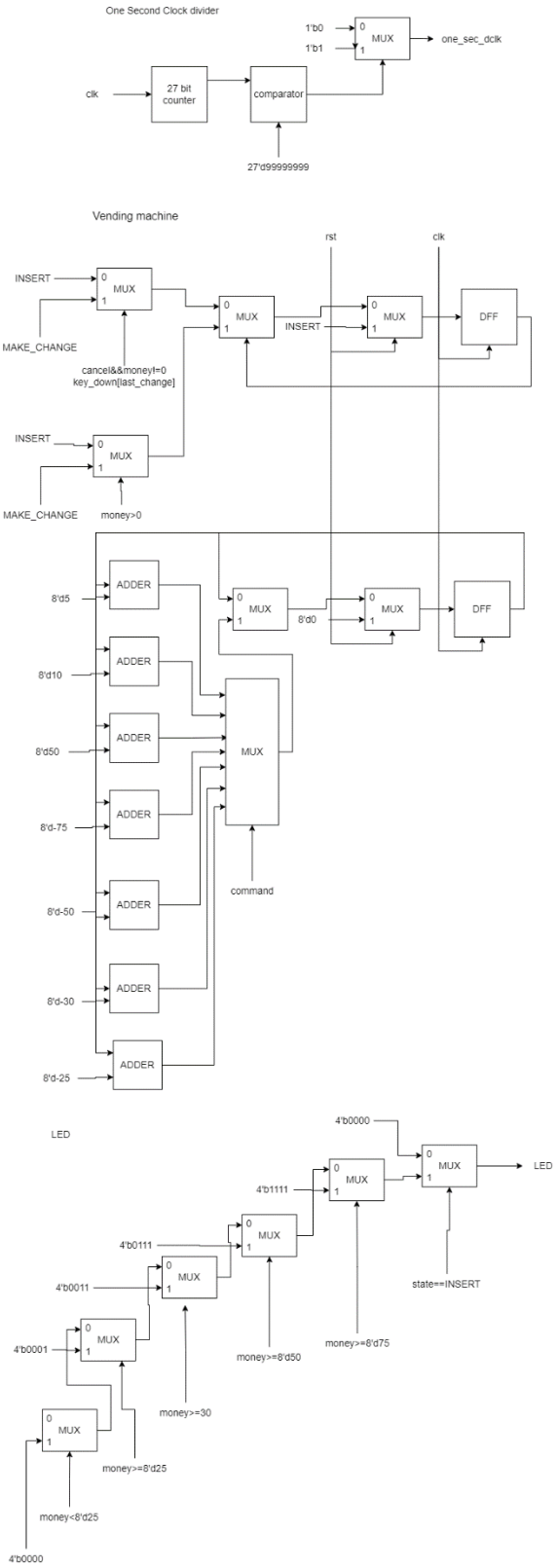
Input 是 debounced 過後的波和 clock。

```
module One_Pulse(
    input clk,
    input debounced,
    output reg one_pulse
);
    reg pb_delay;
    always@(posedge clk)begin
        one_pulse <= debounced & (!pb_delay);
        pb_delay <= debounced;
    end
endmodule
```

- State diagram



- Block diagram



7. 心得

洪聖祥: 這次 lab 雖然沒有很難但畫 block diagram 花了最多時間，詳細畫出設計細節就比較困難了。我自己畫完都不知道自己畫的對不對，我非常非常非常擔心 final project report 要畫 block diagram 要怎麼畫，如果一旦把 module 設計得很複雜而且很多功能，block diagram 會變得又大又難看。經過 5 次 lab，只有前兩次的 block diagram 最好畫，越到後面的 lab 我對畫圖感到恐懼。Keyboard 的部分我覺得蠻簡單的，只要知道一些訊號是有什麼作用的就好了。不用知道底層的細節。

劉奇泓:這次的 lab 第一次運用到 keyboard 和 audio，上星期二課堂在教 keyboard and audio 時我因為身體不適在家休息，結果到了做 lab 的時候我對 keyboard 的運作和 submodule 的細節都不清楚，因此在做 fpga 第一題的時候花了很多時間釐清各個 submodule 的功能，幸好最終有完成 fpga 的 demo，也讓我更熟悉了 keyboard 和 audio 的運作。希望這些經驗都能累積起來，往最後一個 lab 邁進，期末考和 project 也都能順順利利。

8. 分工

洪聖祥: advanced question 3、4 ，fpga 2。

劉奇泓: advanced question 1、2 ，fpga 1。