Lab 5 report

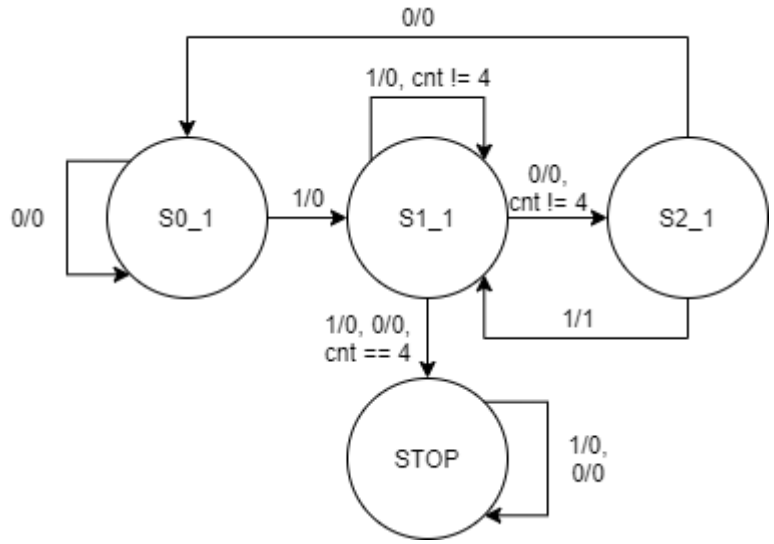
2020/11/26

108062125 高敦晉

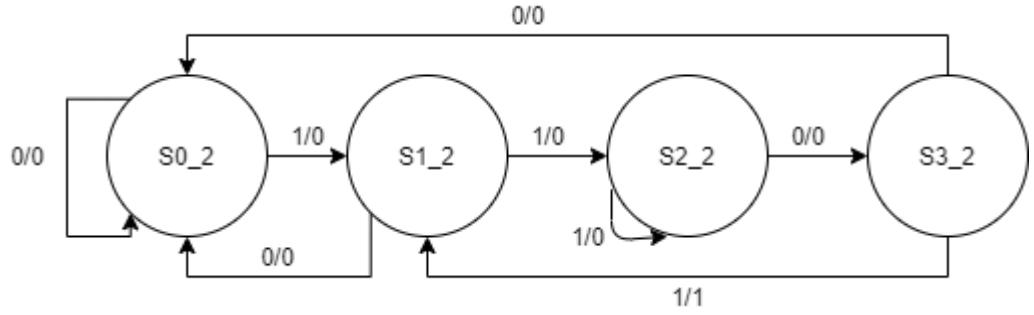
108062229 陳皇佑

Advance Question 1

State-transition Diagram



dec1 state

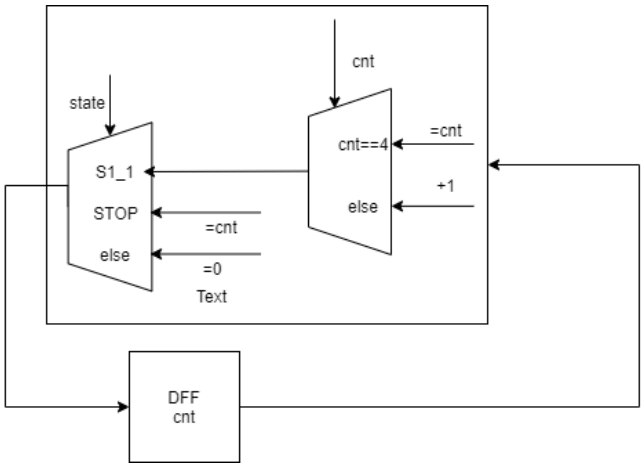


dec2 state

設計構想：

先寫出 dec1 和 dec2 的 state transition diagram，因為 dec1 → 101，dec2→1101 的最後一個和數列第一個皆為 1，因此可重複使用 state S1_1 和 S1_2。再另外寫一個 counter 紀錄 state1 在 S1_1 待了幾次，如果已經 4 次了，state1 就轉到 STOP state，並在那裏自轉。

```
S1_1:
begin
    if(cnt == 3'd4) begin
        nxt_state1 = STOP;
        nxt_cnt = cnt;
        dec1 = 1'b0;
    end
    else begin
        nxt_cnt = cnt + 1'b1;
        if(in == 1'b1) begin
            nxt_state1 = S1_1;
            dec1 = 1'b0;
        end
        else begin
            nxt_state1 = S2_1;
            dec1 = 1'b0;
        end
    end
end
```



counter

Testbench 設計構想：

從 0000~1111 跑過各種可能，並在 1111 後加入幾個 101 測試 dec1。

開發過程中的問題/學習：

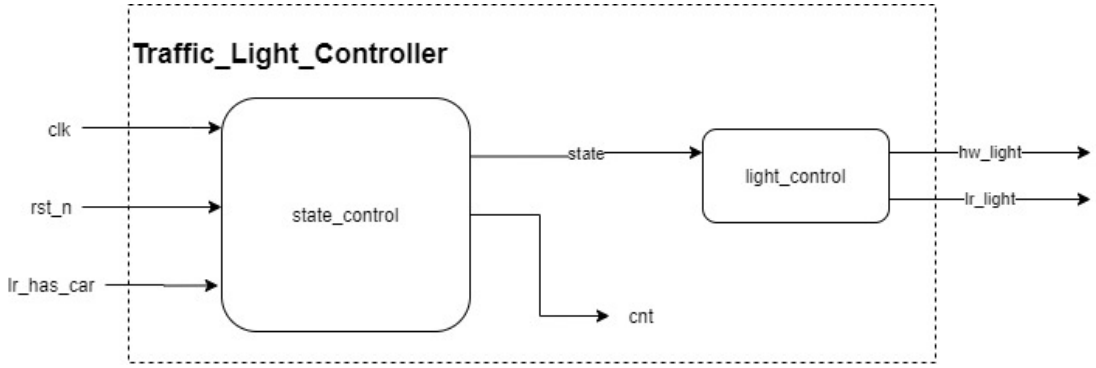
測試連續數列，可以重複利用前面的 state 節省空間，而如果 state 轉換有額外的條件，只需要在轉換 state 的 combinational circuit 寫好條件判斷即可。

Advance Question 2

設計構想：

整體來說分成 light_control unit 跟 state_control unit，一邊純處理 state，一邊純根據 state 處理 output。

整體的 hierarchy 如下。




```
7   wire [3-1:0] state;
8
9   light_control lc(.state(state), .hw_light(hw_light), .lr_light(lr_light));
10  state_control sc(.clk(clk), .rst_n(rst_n), .lr_has_car(lr_has_car), .state(state)
11                  ,.cnt(cnt));
```

Light Control Unit:

```
117   input [3-1:0] state;
118   output [3-1:0] hw_light, lr_light;
119   parameter HW_go = 3'b000;
120   parameter HW_wait = 3'b001;
121   parameter Going_toLR = 3'b010;
122   parameter LR_go = 3'b011;
123   parameter LR_wait = 3'b100;
124   parameter Going_toHW = 3'b101;
125   parameter Green = 3'b100;
126   parameter Yellow = 3'b010;
127   parameter Red = 3'b001;
128   wire [3-1:0] state;
129   reg [3-1:0] hw_light, lr_light;
```

將 state 用 parameter 定義出來

HW_go/LR_go: 東西/南北向通車。

HW_wait/LR_wait : 東西/南北向轉黃燈，南北/東西向維持紅燈

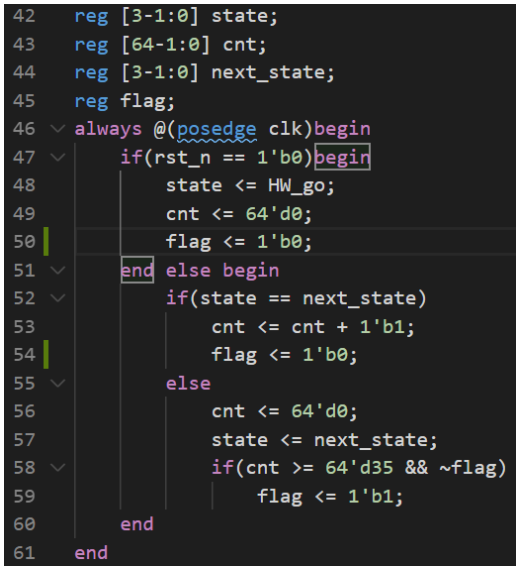
Going_toLR/Going_toHW:全部紅燈，下一個要讓南北/東西向通車

Green/Yellow/Red 分別代表哪個燈亮。

```
133   case (state)
134       HW_go : begin
135           hw_light = Green;
136           lr_light = Red;
137       end
138       HW_wait:begin
139           hw_light = Yellow;
140           lr_light = Red;
141       end
142       Going_toLR:begin
143           hw_light = Red;
144           lr_light = Red;
145       end
146       LR_go:begin
147           hw_light = Red;
148           lr_light = Green;
149       end
150       LR_wait:begin
151           hw_light = Red;
152           lr_light = Yellow;
153       end
154       Going_toHW:begin
155           hw_light = Red;
156           lr_light = Red;
157       end
158       default:begin
159           hw_light = Green;
160           lr_light = Green;
161       end
```

根據目前的 state 定義每個燈號。

State Control Unit :



```
reg [3-1:0] state;
reg [64-1:0] cnt;
reg [3-1:0] next_state;
reg flag;
always @(posedge clk)begin
    if(rst_n == 1'b0)begin
        state <= HW_go;
        cnt <= 64'd0;
        flag <= 1'b0;
    end else begin
        if(state == next_state)
            cnt <= cnt + 1'b1;
            flag <= 1'b0;
        else
            cnt <= 64'd0;
            state <= next_state;
            if(cnt >= 64'd35 && ~flag)
                flag <= 1'b1;
    end
end
```

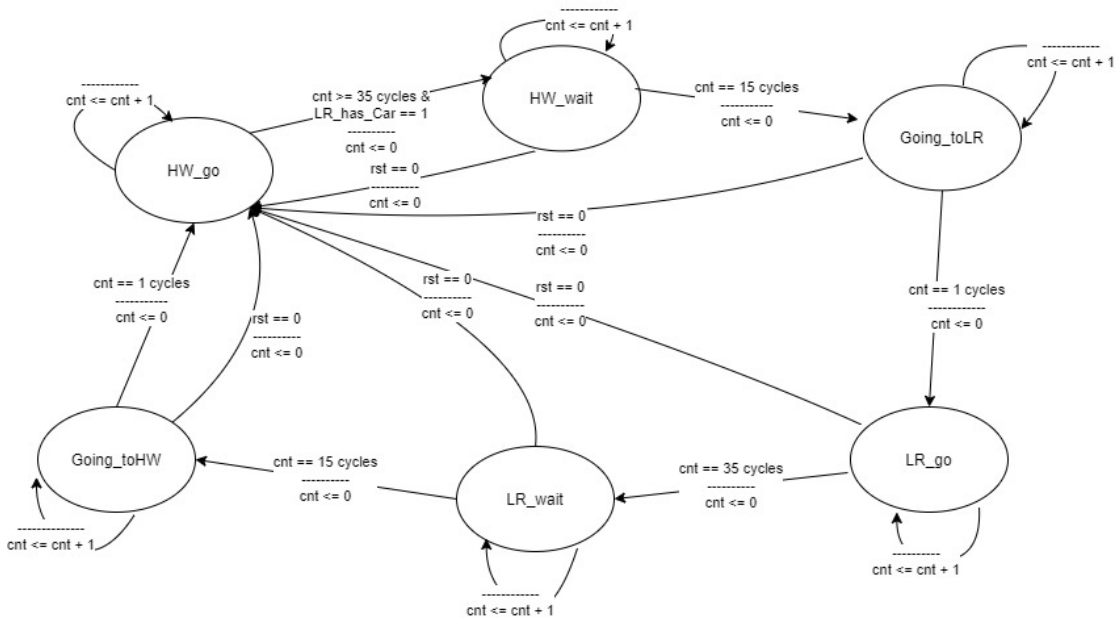
用 cnt 來數有幾個 cycle，如果下一個 state 換與現在的 state 不同，就重置 cnt。因為怕 cnt 有天會溢位然後以為沒超過 35，因此多設一個 flag 來表示已經數超過一次 35 了。

```
64  always @(*)begin
65  case (state)
66      HW_go :
67          if((cnt >= 64'd34 || flag == 1'b1) && lr_has_car == 1'b1)begin
68              next_state = HW_wait;
69          end else begin
70              next_state = HW_go;
71          end
72      HW_wait:
73          if(cnt >= 64'd14)begin
74              next_state = Going_toLR;
75          end else begin
76              next_state = HW_wait;
77          end
78      Going_toLR:
79          if(cnt >= 64'd0)begin
80              next_state = LR_go;
81          end else begin
82              next_state = Going_toLR;
83          end
```

```
84      LR_go:
85          if(cnt >= 64'd34)begin
86              next_state = LR_wait;
87          end else begin
88              next_state = LR_go;
89          end
90      LR_wait:
91          if(cnt >= 64'd14)begin
92              next_state = Going_toHW;
93          end else begin
94              next_state = LR_wait;
95          end
96      Going_toHW:
97          if(cnt >= 64'd0)begin
98              next_state = HW_go;
99          end else begin
00              next_state = Going_toHW;
01          end
02      default:
03          next_state = state;
04  endcase
```

將每個 state 根據當前的 input 以及 cnt，指定下一個轉換的 state。

下圖附上 state transition diagram



“-------"的上面代表條件，如果沒有寫就是 default 往這裡走。Rst 的條件優先度最高。下面代表會作的操作，主要是處理 cnt 的計算。

開發過程中的問題/學習：

這題主要學到的部分是在估算 cnt 時的條件式，要寫成 cnt >= (期望的 cycle 次數 – 1)由於 cnt 是從 0 開始數的。

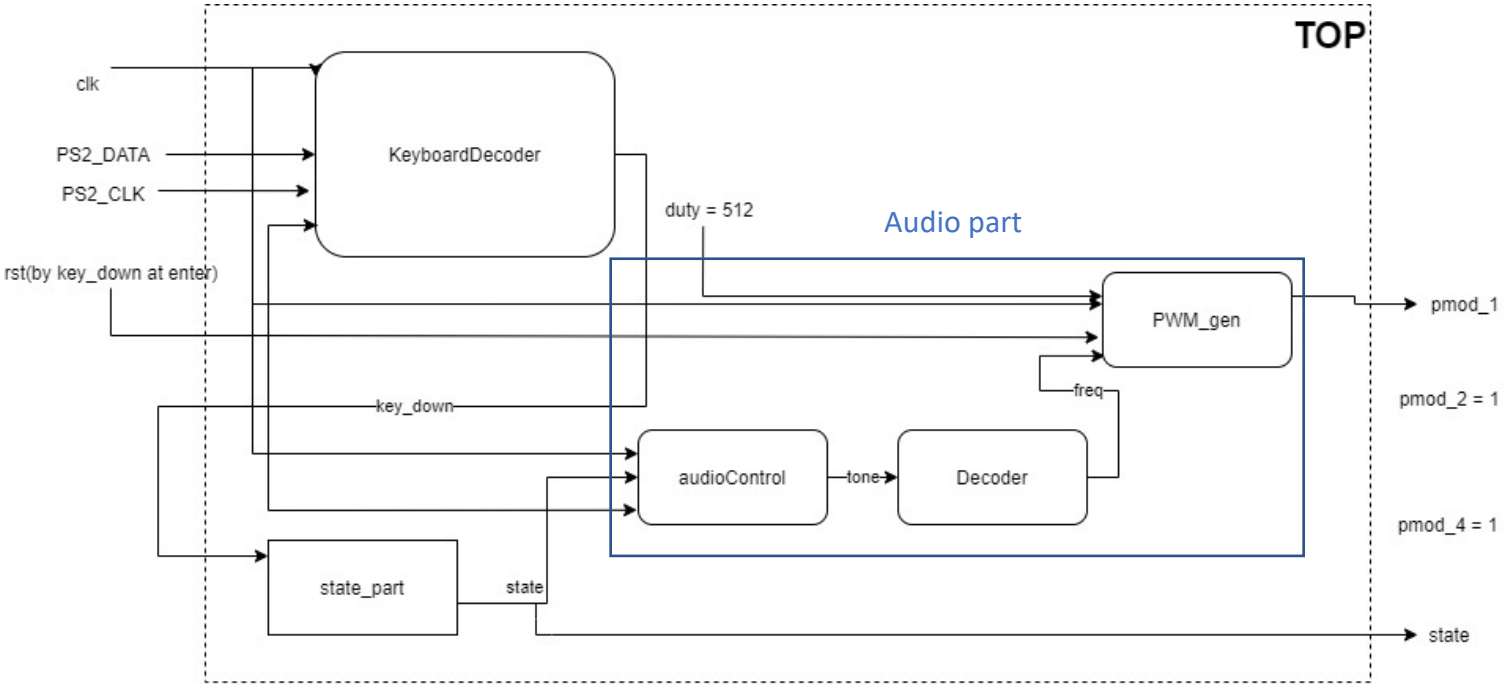
此外就是練習 FSM 的規劃以及實作，在這次的 Top module 只簡單的放入了兩個 submodule，在做測試時，可以很輕鬆的更改來接出 wire，可謂體會 FSM 的強大。

FPGA-1

設計構想：

分成 audio part, keyboard part, state part 三個部分

整體的 hierarchy



Audio Part :

```
38  wire [15:0] tone;
39  wire [31:0] freq;
40  assign pmod_2 = 1'd1;   //no gain(6dB)
41  assign pmod_4 = 1'd1;   //turn-on
42
43  audioControl AC(
44      .clk(clk),
45      .rst(rst),
46      .state(state),
47      .tone(tone)
48  );
49
50  Decoder decoder00 (
51      .tone(tone),
52      .freq(freq)
53  );
54
55  PWM_gen pwm_0 (
56      .clk(clk),
57      .reset(reset),
58      .freq(freq),
59      .duty(10'd512),
60      .PWM(pmod_1)
61  );
```

大致上跟 demo 時用的 code 差不多，新增了一個 audioControl 來做根據 state 輸出 tone。

另外也更改了 Decoder.v 裏頭的 freq 音高。

```
16'd1: freq = 32'd262;  //Do-m
16'd2: freq = 32'd294;  //Re-m
16'd3: freq = 32'd330;  //Mi-m
16'd4: freq = 32'd349;  //Fa-m
16'd5: freq = 32'd392;  //Sol-m
16'd6: freq = 32'd440;  //La-m
16'd7: freq = 32'd494;  //Si-m
16'd8: freq = 32'd262 << 1; //Do-h
16'd9: freq = 32'd294 << 1;
16'd10: freq = 32'd330 << 1;
16'd11: freq = 32'd349 << 1;
16'd12: freq = 32'd392 << 1;
16'd13: freq = 32'd440 << 1;
16'd14: freq = 32'd494 << 1;
16'd15: freq = 32'd262 << 2; // Do C6
16'd16: freq = 32'd294 << 2;
16'd17: freq = 32'd330 << 2;    //Mi-m
16'd18: freq = 32'd349 << 2;    //Fa-m
16'd19: freq = 32'd392 << 2;    //Sol-m
16'd20: freq = 32'd440 << 2;    //La-m
16'd21: freq = 32'd494 << 2;    //Si-m
16'd22: freq = 32'd262 << 3; // Do C7
16'd23: freq = 32'd294 << 3;
16'd24: freq = 32'd330 << 3;    //Mi-m
16'd25: freq = 32'd349 << 3;    //Fa-m
16'd26: freq = 32'd392 << 3;    //Sol-m
16'd27: freq = 32'd440 << 3;    //La-m
16'd28: freq = 32'd494 << 3;    //Si-m
16'd29: freq = 32'd262 << 4; // Do C8
default : freq = 32'd20000; //Do-dummy
```

audioControl :

```
214   parameter upOne = 3'b000;
215   parameter upHf = 3'b001;
216   parameter downOne = 3'b010;
217   parameter downHf = 3'b011;
218
219   reg [31:0] cnt;
220   parameter SEC = 32'd100_000_000;
221   parameter hfSEC = 32'd50_000_000;
222   always @(posedge clk, posedge rst)begin
223       if(rst)begin
224           cnt <= 32'd0;
225           tone <= 16'd1;
226       end else begin
227           if(cnt >= SEC)
228               cnt <= 32'd0;
229           else
230               cnt <= cnt + 1'b1;
231           case(state)
232           upOne:begin
233               if(cnt == SEC && tone != 16'd29)begin
234                   tone <= tone + 16'd1;
235               end else begin
236                   tone <= tone;
237               end
238           end
```

```
240           upHf:begin
241               if((cnt == SEC || cnt == hfSEC) && tone != 16'd29)begin
242                   tone <= tone + 16'd1;
243               end else begin
244                   tone <= tone;
245               end
246           end
247           downOne:begin
248               if(cnt == SEC && tone != 16'd1)begin
249                   tone <= tone - 16'd1;
250               end else begin
251                   tone <= tone;
252               end
253           end
254           downHf:begin
255               if((cnt == SEC || cnt == hfSEC) && tone != 16'd1)begin
256                   tone <= tone - 16'd1;
257               end else begin
258                   tone <= tone;
259               end
260           end
261           default :tone <= tone;
262           endcase
263       end
264   end
```

state 的定義：
downOne： 往下 1Hz 數音高，downHf： 往下 2Hz 數音高。
upOne： 往上 1Hz 數音高，upHf： 往上 2Hz 數音高。

rst 時，將 tone 直接設成 1 用 cnt 一直數到 100,000,000，每數到

100,000,000 就增加/減少 tone，製造出 2Hz 的方式是在 state 屬於 Hf 系列

時，在條件式裡多增加 if(cnt == hfSEC)

而 hfSEC == SEC/2=50,000,000，如此便能只用一個 cnt 製造出半秒增加/減

少的效果

Keyboard Part :

```
    parameter [8:0] LEFT_SHIFT_CODES  = 9'b0_0001_0010;
    parameter [8:0] RIGHT_SHIFT_CODES = 9'b0_0101_1001;
    parameter [8:0] KEY_CODES [0:7] = {
        9'b0_0100_0101, // 0 => 45
        9'b0_0001_0110, // 1 => 16
        9'b0_0001_1110, // 2 => 1E

        9'b0_0111_0000, // right_0 => 70
        9'b0_0110_1001, // right_1 => 69
        9'b0_0111_0010, // right_2 => 72

        9'b0_0101_1010, // enter => 5A
        9'b1_0101_1010  // right_enter => E05A
    };
    reg [9:0] last_key;

    wire shift_down;
    wire [511:0] key_down;
    wire [8:0] last_change;
    wire been_ready;

    // assign shift_down = (key_down[LEFT_SHIFT_CODES] == 1'b1 || key_down[RIGHT_SHIFT_CODES] == 1'b1) ? 1'b1 : 1'b0;
    assign enter_down = (key_down[KEY_CODES[6]] == 1'b1 || key_down[KEY_CODES[7]] == 1'b1) ? 1'b1 : 1'b0;
    assign rst = enter_down;
    KeyboardDecoder key_de (
        .key_down(key_down),
        .last_change(last_change),
        .key_valid(been_ready),
        .PS2_DATA(PS2_DATA),
        .PS2_CLK(PS2_CLK),
        .rst(rst),
        .clk(clk)
    );
```

大致上也跟 demo 的範例 keyboard code 一樣，差在按鍵只剩下需要的 0, 1, 2, enter code。

State Part :

```
    parameter upOne = 3'b000;
    parameter upHf = 3'b001;
    parameter downOne = 3'b010;
    parameter downHf = 3'b011;
        always @ (posedge clk, posedge rst) begin
            if(rst)begin
                state <= upHf;
            end else begin
                if (been_ready == 1'b1 && key_down[last_change] == 1'b1) begin
                    state <= next_state;
                end else begin
                    state <= state;
                end
            end
        end
```

處理 state，如果 rst == 1，state 直接 reset 到 upHf，如果有鍵被按下更新到 next_state，否則就保持當前的 state。next_state 的處理在下方。

```
always @ (*) begin
    case(last_change)
            KEY_CODES[0]: begin
                case(state)
                    upOne:
                        next_state = downOne;
                    upHf:
                        next_state = downHf;
                    default:
                        next_state = state;
                endcase
            end
            KEY_CODES[1]:begin
                case(state)
                    downOne:
                        next_state = upOne;
                    downHf:
                        next_state = upHf;
                    default:
                        next_state = state;
                endcase
            end
            KEY_CODES[2]:begin
                case(state)
                    downOne:
                        next_state = downHf;
                    upOne:
                        next_state = upHf;
                    downHf:
                        next_state = downOne;
                    upHf:
                        next_state = upOne;
```

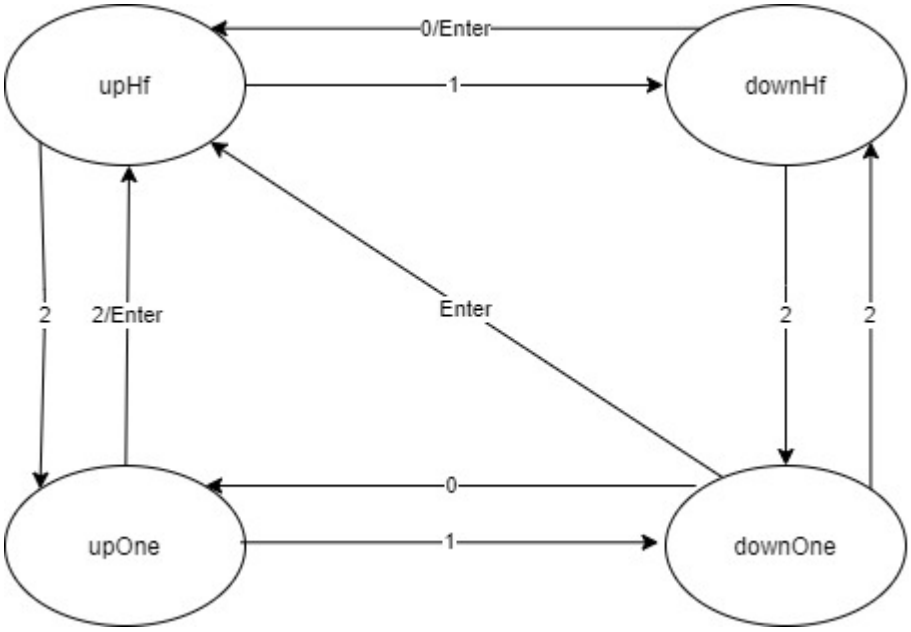
```
    KEY_CODES[3]: begin
        case(state)
            upOne:
                next_state = downOne;
            upHf:
                next_state = downHf;
            default:
                next_state = state;
        endcase
    end
    KEY_CODES[4]:begin
        case(state)
            downOne:
                next_state = upOne;
            downHf:
                next_state = upHf;
            default:
                next_state = state;
        endcase
    end
    KEY_CODES[5]:begin
        case(state)
            downOne:
                next_state = downHf;
            upOne:
                next_state = upHf;
            downHf:
                next_state = downOne;
            upHf:
                next_state = upOne;
            default:
                next_state = state;
        endcase
```

```
                KEY_CODES[6]:begin
                    next_state = upHf;
                end
                KEY_CODES[7]:begin
                    next_state = upHf;
                end
                default     : next_state = state;
        endcase

    end

endmodule
```

處理 next_state 的去處，根據 KEY_CODE 來做變換，default 是不變 state，state transition diagram 在下方。這邊要注意的是由於前面已經有判斷是否有按鍵按下，故不用再處理沒按鍵的 case。



開發過程中的問題/學習：

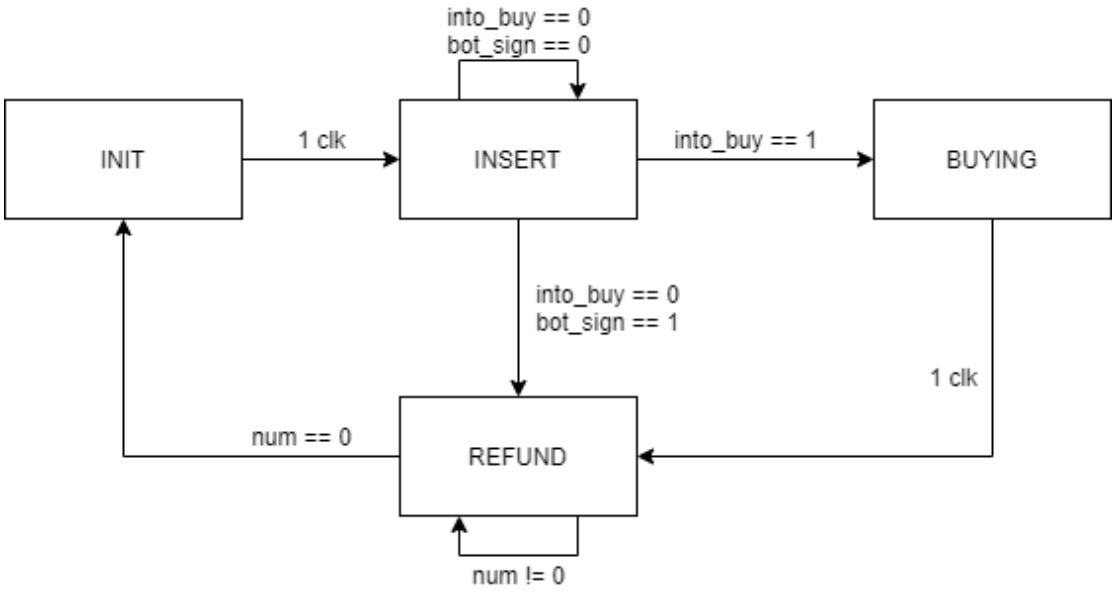
在寫這份題目之前，仔細的 trace demo 時的 keyboard 以及 piano code 一次，有練習到看懂整個架構的速度並擷取需要用的部分做使用、並能夠自行修改使其符合需要。

這個題目的設計結構中途大改了一次，原先我是希望能夠吧 keyboard 的訊號接成一個 botton 的感覺，然後再接進去一個 stateControl 的 module，但是後來一直弄不好 botton，所以就直接沿用 piano 的架構，拿 Keydown 以及 last_change 作為 state transition 的依據。

再一次體會到 FSM 的重要性，因為中途改架構的關係，多虧一開始就有先想好整體 state 的結構，更改 code 時也方便許多，很多部分都是小改後就能直接沿用的。

FPGA2

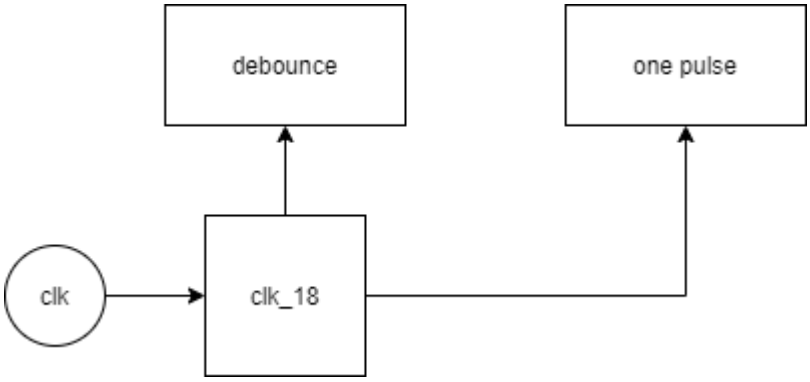
State-transition Diagram



Vending machine

設計構想：

先把所有按鈕接上 debounce 跟 one pulse，而 state 的 dff, Debounce, one_pulse，都接上 clockdevider(clk_18)，因為 debounce 要接上延長過的 clock 才算真正除顫。



```
always @ (posedge clk_18[17], posedge top_sign)
begin
  if(top_sign == 1'b1)
  begin
    state <= INIT;
    num <= 8'b0000_0000;
  end
  else
  begin
    state <= nxt_state;
    num <= nxt_num;
  end
end
```

在 INIT state 初始化 num、cnt、into_refund。

```
INIT:
begin
  nxt_state = INSERT;
  nxt_num = num;
  into_refund = 1'b0;
  nxt_cnt = 26'd1;
end
```

into_buy == 1 代表在 INSERT 狀態下 asdf 被按下，且金額足夠購買，下一個 state 為 BUYING。而因為 keyboard_decoder 的 clk 訊號是接原生 clk，故 into_buy 需要使用 delay 延長 clk_18 個長度。當 delay == 1 時代表現在已經有一個購買正在進行，此時再按其他商品並不會進行購買。把按哪個商品用 4bit drink_code 表達，哪些商品能夠購買用 4bit led 表示，只要 drink_code & led 不為 0000 代表可以購買。

```
always @ (posedge clk)
begin
  if(into_buy == 1'b0)
    delay <= 18'd0;
  else
    delay <= delay + 1'b1;
end
```

```
if(into_buy == 1'b1)
begin
  nxt_state = BUYING;
end
```

```
always @ (*)
begin
  if(num[7:4] < 4'd6)
  begin
    if(num[7:4] < 4'd3)
    begin
      if(num[7:4] == 4'd2)
      begin
        if(num[3:0] < 4'd5)
          led = 4'b0001;
        else
          led = 4'b0011;
      end
      else
        led = 4'b0000;
    end
    else
      led = 4'b0111;
  end
```

Led[3] 代表 coffee

Led[2] 代表 coke

Led[1] 代表 oolong

Led[0] 代表 water

```
always @ (*)
begin
  case (last_change)
    KEY_CODES[00] :
      drink_code = 4'b1000;
    KEY_CODES[01] :
      drink_code = 4'b0100;
    KEY_CODES[02] :
      drink_code = 4'b0010;
    KEY_CODES[03] :
      drink_code = 4'b0001;
    default     :
      drink_code = 4'b0000;
  endcase
end
```

drink_code[3] 代表 coffee

drink_code [2] 代表 coke

drink_code [1] 代表 oolong

drink_code [0] 代表 water

```
always @ (*)
begin
  if(state == INSERT)
  begin
    if(delay == 18'd0)
    begin
      if(been_ready == 1'b1 && key_down[last_change] == 1'b1)
      begin
        buy = drink_code & led;
        if(buy == 4'b0000)
          into_buy = 1'b0;
        else
          into_buy = 1'b1;
      end
      else
        into_buy = 1'b0;
    end
    else
      into_buy = 1'b1;
  end
  else
    into_buy = 1'b0;
```

用 coin 紀錄投入多少錢。

```
always @ (*)
begin
  if(state == INSERT)
  begin
    if(left_sign == 1'b1)
      coin = 6'd5;
    else
    begin
      if(ctr_sign == 1'b1)
        coin = 6'd10;
      else
      begin
        if(right_sign == 1'b1)
          coin = 6'd50;
        else
          coin = 6'd0;
      end
    end
  end
  else
    coin = 6'd0;
end
```

接著在 INSERT state 中依 coin 改變 num，

```
case(coin)
  6'd0:
    nxt_num = num;
  6'd5:
  begin
    if(num == 8'b1001_0101)
      nxt_num = 8'b1001_1001;
    else
    begin
      if(num[3:0] == 4'd5)
      begin
        nxt_num[3:0] = 4'd0;
        nxt_num[7:4] = num[7:4] + 4'd1;
      end
      else if(num[3:0] == 4'd0)
        nxt_num[3:0] = 4'd5;
      else
        nxt_num = 8'b1001_1001;
    end
  end
```

```
6'd10:
begin
  if(num[7:4] == 4'd9)
    nxt_num = 8'b1001_1001;
  else
    nxt_num[7:4] = num[7:4] + 4'd1;
end
6'd50:
begin
  if(num[7:4] < 4'd5)
    nxt_num[7:4] = num[7:4] + 4'd5;
  else
    nxt_num = 8'b1001_1001;
end
default:
  nxt_num = 8'b0001_0001;
```

在 INSERT 中，按下 bottom button 進入 REFUND state。

```
if(bot_sign == 1'b1)
  nxt_state = REFUND;
```

BUYING state 中依買入商品更改 num，並轉到 REFUND state。

```
nxt_state = REFUND;
```

```
case(buy)
  4'b1000:
    nxt_num[7:4] = num[7:4] - 4'd6;
  4'b0100:
    nxt_num[7:4] = num[7:4] - 4'd3;
  4'b0001:
    nxt_num[7:4] = num[7:4] - 4'd2;
  4'b0010:
  begin
    if(num[3:0] < 4'd5)
    begin
      nxt_num[7:4] = num[7:4] - 4'd3;
      nxt_num[3:0] = 4'd5;
    end
    else
    begin
      nxt_num[7:4] = num[7:4] - 4'd2;
      nxt_num[3:0] = num[3:0] - 4'd5;
    end
  end
```

第一次進入 REFUND，reset cnt，接著用 cnt 實現一秒變動一次的條件。接著看現在 num 為多少，將 num 減 5。當 num 為 0，下個 state 為 INIT，num keep 原值。

```
if(into_refund == 1'b0)
begin
  nxt_cnt = 26'd1;
  into_refund = 1'b1;
end
else
begin
  nxt_cnt = cnt + 1'b1;
end
```

```
always @ (posedge clk, posedge top_sign)
begin
  if (top_sign == 1'b1)
    cnt <= 26'd0;
  else
    cnt <= nxt_cnt;
end
```

```
if(num == 8'b0000_0000)
begin
  nxt_state = INIT;
  nxt_num = num;
end
```

```
else
begin
  nxt_state = REFUND;
  if(cnt == 26'd0)
  begin
    nxt_num[7:4] = num[7:4];
    if(num[3:0] == 4'd5)
      nxt_num[3:0] = 4'd0;
    else if(num[3:0] == 4'd9)
      nxt_num[3:0] = 4'd4;
    else if(num[3:0] == 4'd4)
    begin
      if(num[7:4] == 4'd0)
      begin
        nxt_num[7:4] = num[7:4];
        nxt_num[3:0] =  4'd0;
      end
      else
      begin
        nxt_num[7:4] = num[7:4] - 4'd1;
        nxt_num[3:0] =  4'd9;
      end
    end
          else
          begin
            nxt_num[7:4] = num[7:4] - 4'd1;
            nxt_num[3:0] =  4'd5;
          end
        end
        else
          nxt_num = num;
      end
    end
    default:
    begin
      nxt_state = INIT;
      nxt_num = 8'b0000_0000;
      into_refund = 1'b0;
    end
  endcase
end
```

開發過程中的問題/學習：

一開始按 reset 時 state 就會卡住，之後才發現 clock divider 不能接 reset，不然 reset 訊號會一直維持在 1。一開始也讀不到鍵盤的訊號，才發覺因為 keyboard_decoder 是接原生 clk，訊號維持時間太短，需要將期延長。而在做數字加減時原本等號兩邊都放同一個變數，結果會一直跳 warning，說會形成 self-loop，不能 generate bitstream，才知道好的 coding 習慣真的很重要。

分工：

AQ1、FPGA2 ,report: 陳皇佑

AQ2、FPGA1, report&report 合併 : 高敦晉