Lab 4 report

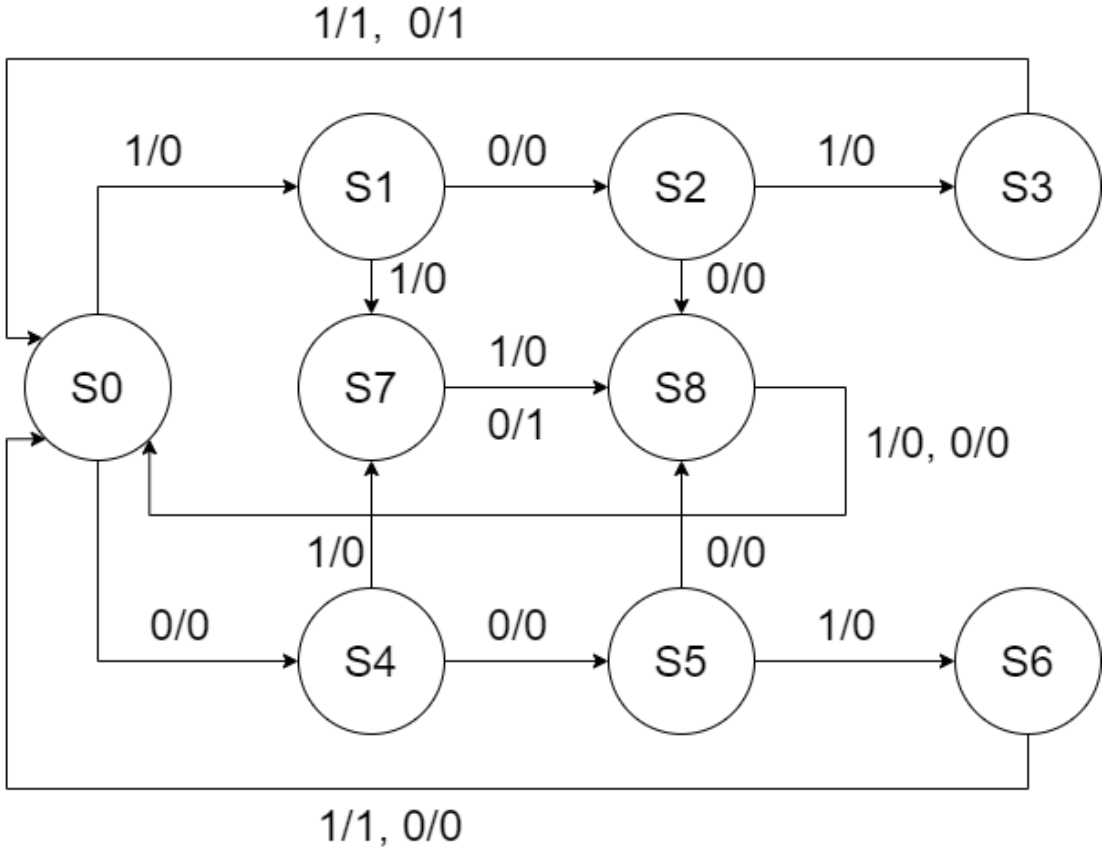
2020/11/12

108062125 高敦晉

108062229 陳皇佑

AQ1

State-transition Diagram



設計構想：

仔細觀察發現，偵測數列具重複性—1010, 1011 前三個 bit 相同，故可以共用同樣的 state(S0, S1, S2, S3)。而因為每 4 個 clk 偵測一次，因此當受偵測數列確認不符合目標數列時，就轉到另外的 state(S7, S8)，讓每種可能輸入都在經過 4 個 state 後回到原 state。照上列敘述畫出 state-transition diagram，依照 mealy machine 的方式將其轉成程式碼即可(output 及 state 變化要放在 input 的 if-else 裡，如下圖)。

```
S0:
begin
    if(in == 1'b1) begin
        nxt_state = S1;
        dec = 1'b0;
    end
    else begin
        nxt_state = S4;
        dec = 1'b0;
    end
end
```

Testbench 設計構想：

從 0000 到 1111 列出所有可能，然後一個 clk 一個 bit 從最右邊的 bit 讀到最左邊的 bit。

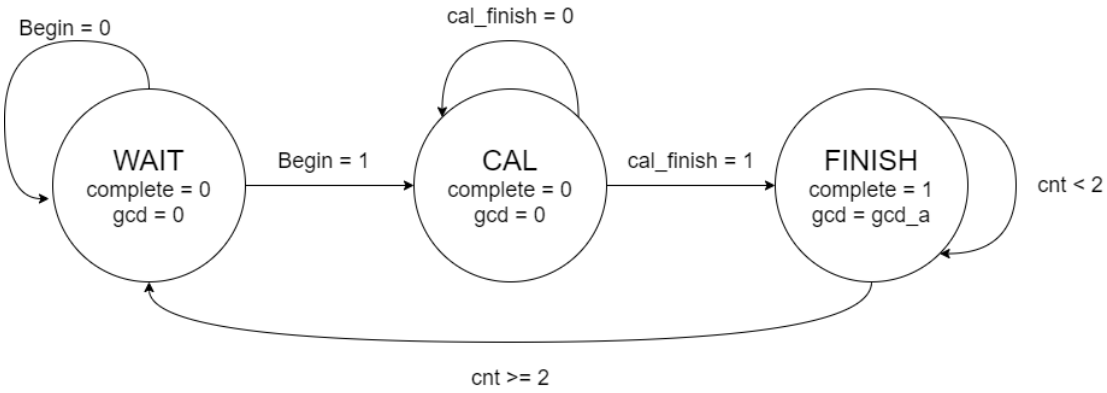
```
@ (negedge clk) rst_n = 1'b0;
@ (posedge clk)
@ (negedge clk) rst_n = 1'b1;
#(`CYC * 3);
repeat(2**4 - 1) begin
    for(i = 0; i < 4; i = i + 1) begin
        #(`CYC) in = cnt[i];
    end
    cnt = cnt + 1'b1;
end
#(`CYC) $finish;
```

開發過程中的問題/學習：

學到如何將一個現實應用的問題轉化成 mealy machine 的圖，將他轉成程式碼變得容易許多。而且如果好好安排 state，可以節省 state 的個數，讓 code 看起來更簡潔。

AQ2

State-transition Diagram



設計構想：

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

State reset 成 WAIT。

```
always @ (posedge clk) begin
    gcd_a <= nxt_gcd_a;
    gcd_b <= nxt_gcd_b;
end
```

為了讓輾轉相除法能一個 clk 做一次操作，用 dff 實現。

```
WAIT:
begin
    Complete = 1'b0;
    gcd = 16'd0;
    if(Begin == 1'b1) nxt_state = CAL;
    else nxt_state = WAIT;
end
```

```
if(state == WAIT) begin
    if(Begin == 1'b1) begin
        nxt_gcd_a = a;
        nxt_gcd_b = b;
        cal_finish = 1'b0;
    end
    else begin
        nxt_gcd_a = nxt_gcd_a;
        nxt_gcd_b = nxt_gcd_b;
        cal_finish = cal_finish;
    end
end
```

當 WAIT 狀態時，如果 Begin 為 1，state 就變 CAL，並且讀取 input，讓 nxt_gcd_a = a 和 nxt_gcd_b = b，重製 cal_finish = 0。Begin = 0，state 就變 WAIT，nxt_gcd_a = a 和 nxt_gcd_b = b，和 cal_finish 維持原值。

```
CAL:
begin
    Complete = 1'b0;
    gcd = 16'd0;
    if(cal_finish == 1'b0) nxt_state = CAL;
    else nxt_state = FINISH;
end
```

```
else if (state == CAL) begin
    if(gcd_b == 16'd0) begin
        nxt_gcd_a = gcd_a;
        nxt_gcd_b = gcd_b;
        cal_finish = 1'b1;
    end
    else begin
        cal_finish = 1'b0;
        if(gcd_a > gcd_b) begin
            nxt_gcd_a = gcd_a - gcd_b;
            nxt_gcd_b = gcd_b;
        end
        else begin
            nxt_gcd_a = gcd_a;
            nxt_gcd_b = gcd_b - gcd_a;
        end
    end
end
```

cal_finish 代表輾轉相除法完成，當 state = CAL 時，如果還沒完成(cal_finish = 0)，state hold on CAL，完成(cal_finish = 1) state 變成 FINISH。

當 gcd_b = 0 時，輾轉相除法完成，nxt_gcd_a = gcd_a，nxt_gcd_b = gcd_b，cal_finish = 1。若 gcd_b != 0，cal_finish = 0 持續做輾轉相除法，gcd_a > gcd_b 則 nxt_gcd_a = gcd_a - gcd_b，nxt_gcd_b = gcd_b，否則 nxt_gcd_a = gcd_a，nxt_gcd_b = gcd_b - gcd_a。

```
FINISH:
begin
    if(cnt < 2'b10) nxt_state = FINISH;
    else nxt_state = WAIT;
    Complete = 1'b1;
    gcd = gcd_a;
end
```

```
else if(state == FINISH) begin
    nxt_gcd_a = nxt_gcd_a;
    nxt_gcd_b = nxt_gcd_b;
    cal_finish = cal_finish;
end
```

```
always @ (negedge clk) begin
    if(state == FINISH) cnt <= nxt_cnt;
    else cnt <= 2'b00;
end
```

當 state = FINISH 時，如果 cnt < 2 維持 FINISH，否則轉成 WAIT。Counter 只在 state = FINISH 時運轉，讓 FINISH 可以維持 2clk。Counter 用 negedge trigger，因為要讓變數不在 posedge clk 時改變。State = FINISH，nxt_gcd_a、nxt_gcd_b 、cal_finish hold。

Testbench 設計構想：

```
reg [15:0] a = 16'd128;
reg [15:0] b = 16'd96;
```

```
@(negedge clk) rst_n = 1'b0;
@(negedge clk) rst_n = 1'b1;
@(negedge clk) Begin = 1'b1;
#(`CYC) Begin = 1'b0;
#(`CYC * 7);
a = 16'd85;
b = 16'd135;
#(`CYC * 2);
@(negedge clk) Begin = 1'b1;
@(negedge clk) Begin = 1'b0;
#(`CYC * 11);
a = 16'd247;
b = 16'd247;
Begin = 1'd1;
#(`CYC) Begin = 1'b0;
#(`CYC * 4);

$finish;
```

測 a > b，a < b，a == b，中間 begin = 0 時，觀察 state 是否維持在 0。

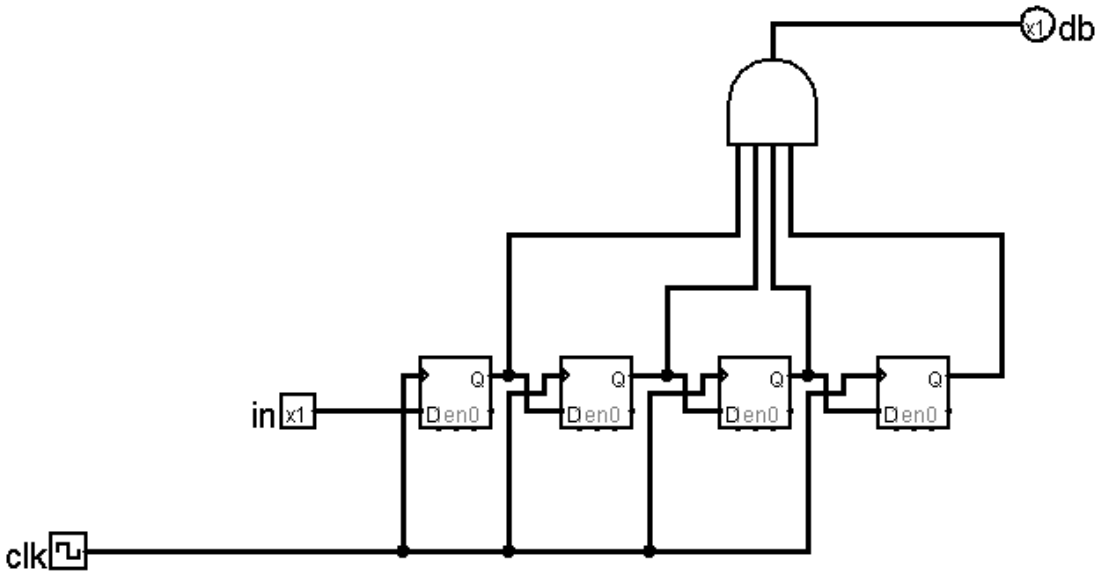
開發過程中的問題/學習：

一開始 counter 一直達不到想要的效果，後來發現是因為 dff 和 counter 都是 posedge，造成訊號不準，後來改成 posedge 就好了。也學到可以用減法取代 mod 來使用輾轉相除法。

FPGA

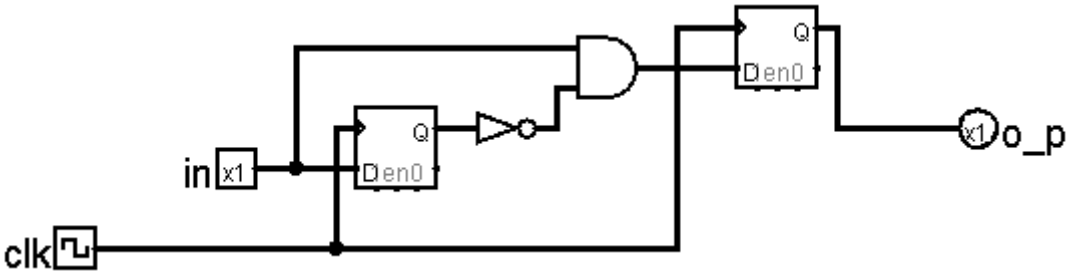
設計構想：

Onepluse、debounce 以及 for 7-segment 顯示頻率的 clock divider 基本都沿用上次的 code




```
275  module debounce(out, bottom, clk);
276  input bottom, clk;
277  output out;
278  reg [3:0] cnt;
279
280  always @(posedge clk)begin
281      cnt[3:1] <= cnt[2:0];
282      cnt[0] <= bottom;
283  end
284
285  assign out = (cnt[3:0] == 4'b1111 ? 1'b1 : 1'b0);
286
287  endmodule
```

Debounce 是讓 input 經過 4 個 dff，確保要四個 clk cycle 連續為 1。




```
289   module one_pulse(in, o_p, clk);
290   input in, clk;
291   output o_p;
292
293   reg in_delay;
294   reg o_p;
295   always @(posedge clk)begin
296       o_p <= in & (!in_delay);
297       in_delay <= in;
298   end
299
300   endmodule
```

One_pulse 是 in 跟 in_delay 的相反做 and，結果用 dff 輸出，這樣確保訊號 1 只會維持一個 clk cycle。

```
341   parameter conNum = 26'd10000000;
342   always @(negedge clk)begin
343       cnt <= next_cnt;
344       if(cnt == conNum - 1'b1)
345           // clk_10 <= !clk_10;
346           clk_10 <= 1'b1;
347       else
348           clk_10 <= 1'b0;
349   end
350
351   always @(*)begin
352       if(cnt >= conNum - 1'b1)begin
353           next_cnt  = 26'd0;
354       end
355       else begin
356           next_cnt = cnt + 1'b1;
357       end
358   end
```

```
always @ (posedge clk)
    begin
        cnt <= nxt_cnt;
        if (cnt === 16'b1111_1111_1111_1111) clk1_2_16 <= 1'b1;
        else clk1_2_16 <= 1'b0;
    end

always @ (*) begin
    if(cnt === 16'b1111_1111_1111_1111) nxt_cnt = 16'd0;
    else nxt_cnt = cnt + 1'b1;
end
```

Clock divider，用一個 dff 做計數器，clksecdiv_10 是數到10^7用來模擬現實中的0.1秒用，注意一下這裡是用 negedge trigger，後面會提到為何如此做。Clock divider 216 是數到2^{17}，之後 counter 歸 0，output 轉 1。用來做顯示頻率。Clock divider 2 是測試用的 module 就不列進來。

Extend module：

```
302  module Extend(clk, in, out);
303  input clk, in;
304  output out;
305  reg [25:0] cnt, next_cnt;
306
307  parameter conNum = 26'd10000000;
308
309  always @(posedge clk)begin
310      if(in == 1'b1 && cnt == 26'd0)begin
311          cnt <= cnt + 1'b1;
312      end
313      else if(cnt != 26'd0)begin
314          cnt <= next_cnt;
315      end
316      else begin
317          cnt <= 26'd0;
318      end
319  end
320
321  always @(*)begin
322      if(cnt == conNum - 1)
323          next_cnt = 26'd0;
324      else
325          next_cnt = cnt + 1'b1;
326  end
327
328  assign out = (cnt == 26'd0 ? 1'b0 : 1'b1);
329
330  endmodule
```

用來延長 one_pluse 的訊號直到能夠被 0.1 秒的 clk 吃到，解決的上次的 FPGA 按鈕 timing 要按在奇特時間點的問題。

按鈕處理單元：

```
19  wire p_bottom_db, r_bottom_db;
20  debounce db_push_bottom(.bottom(push_bottom), .out(p_bottom_db), .clk(clk));
21  debounce db_reset_bottom(.bottom(reset_bottom), .out(r_bottom_db), .clk(clk));
22
23  wire resetsignal, sp_signal; // sp -> start and pause
24  one_pulse gene_sp_signal(.in(p_bottom_db), .o_p(sp_signal), .clk(clk));
25  one_pulse gene_re_signal(.in(r_bottom_db), .o_p(resetsignal), .clk(clk));
26
27  wire r_signal_ex, sp_signal_ex;
28
29  Extend extend_r_signal(.clk(clk), .in(resetsignal), .out(r_signal_ex));
30  Extend extend_sp_signal(.clk(clk), .in(sp_signal), .out(sp_signal_ex));
```

對 push_bottom、reset_bottom 兩個按鈕做 debounce 和 one_pulse，debounce 傳入 clk，這樣只要一按 db_output 會馬上跳一，one_pulse 傳入，為了讓 one_pulse 長度達到 0.1sec clk 長

度，再將 signal 接入 Extend module。

```
wire n_resetsignal;
not gene_reset(n_resetsignal, r_signal_ex);
```

因為 reset 是等於 0 初始化，所以把處理完的 reset 取反。

```
assign push_bottom_out = push_bottom;
assign reset_bottom_out = reset_bottom;
```

因為按鈕有點不靈敏，用來觀察按鈕到底由沒有按下去的訊號燈。

```
46    counterup countit(.clk(clk), .r_signal(n_resetsignal), .sp_signal(sp_signal_ex)
47            ,.min(min), .sec1(sec1), .sec2(sec2), .secdiv10(secdiv10), .state(state));
```

把參數放進 counterup 內。

Counterup Module：

```
148 state_transition setup( .start_pause(sp_signal),
149                         .reset(r_signal),
150                         .clk(clk), .state(state),
151                       .min(min), .sec1(sec1), .sec2(sec2), .secdiv10(secdiv10)
152                       );
```

先將跟 state 有關的信號接近去 state_transition module 裏頭。

```
154    clksecdiv_10 gene_onetensec(.clk(clk), .clk_10(sec_div10_clk));
```

這次為了遵守上課聽教授講的 ”所有的 reg 都應該用同一個 clk trigger”，因此在有需要用到 10Hz clk 的地方都用 clksecdiv_10 這個 negedge trigger 的 module，用一個 10Hz 升起的信號來跟 posedge 的 clk 做 if。

```
158   always @(posedge clk)begin
159       if(sec_div10_clk == 1'b1) begin
160           case(state)
161           COUNT: begin
162               min <= next_min;
163               sec1 <= next_sec1;
164               sec2 <= next_sec2;
165               secdiv10 <= next_secdiv10;
166           end
167           WAIT: begin
168               min <= min;
169               sec1 <= sec1;
170               sec2 <= sec2;
171               secdiv10 <= secdiv10;
172           end
173           RESET:begin
174               min <= 4'd0;
175               sec1 <= 4'd0;
176               sec2 <= 4'd0;
177               secdiv10 <= 4'd0;
178           end
179           endcase
180       end else begin
181           min <= min;
182           sec1 <= sec1;
183           sec2 <= sec2;
184           secdiv10 <= secdiv10;
185       end
186   end
```

內部就看三個 state (COUNT, WAIT, RESET) 來依據要做什麼動作

如果是 COUNT，那就往下數 0.1 秒。

WAIT，維持現在的值不動。

RESET 重整回去 0:00.0

Else 這邊我是放不動。

```
189   always @(*)begin
190       if(sec1 == 4'd5 && sec2 == 4'd9 && secdiv10 == 4'd9)
191           if(min == 4'd9)
192               next_min = 4'd0;
193           else
194               next_min = min + 1'd1;
195       else
196           next_min = min;
197       if(secdiv10 == 4'd9 && sec2 == 4'd9)
198           if(sec1 == 4'd5)
199               next_sec1 = 4'd0;
200           else
201               next_sec1 = sec1 + 1'd1;
202       else
203           next_sec1 = sec1;
204       if(secdiv10 == 4'd9)begin
205           if(sec2 == 4'd9)
206               next_sec2 = 4'd0;
207           else
208               next_sec2 = sec2 + 1'd1;
209       end else begin
210           next_sec2 = sec2;
211       end
212       if(secdiv10 == 4'd9)
213           next_secdiv10 = 4'd0;
214       else
215           next_secdiv10 = secdiv10 + 1'd1;
216   end
```

用來算下一個秒數的 combinational 電路。順帶一提，為了顯示方便我在 second 的 2 位數是用兩個 4bit 的 reg 去做的。

基本上就是預先判別下一秒會不會進位，決定 next 值。

State Transition module：

```
module state_transition(start_pause, reset, clk, state,
                        min, sec1, sec2, secdiv10);
input start_pause, reset, clk;
input min, sec1, sec2, secdiv10;
output [1:0]state;
wire [3:0] min, sec1, sec2, secdiv10;
parameter COUNT = 2'b01;
parameter WAIT = 2'b10;
parameter RESET = 2'b11;

reg [1:0] state, next_state;

wire sec_div10_clk;

clksecdiv_10 gene_onetensec(.clk(clk), .clk_10(sec_div10_clk));
```

定義三個 state 的 2bit 值，不用 00 是因為我希望可以接出去信號燈觀察 state 的變換有沒有正確。下面一樣生一個 10Hz 的 clk。

```
238   always @(posedge clk)begin
239       if(sec_div10_clk == 1'b1)begin
240           if(reset == 1'b0)begin
241               state <= RESET;
242           end else
243               state <= next_state;
244       end
245       else begin
246           state <= state;
247       end
248   end
```

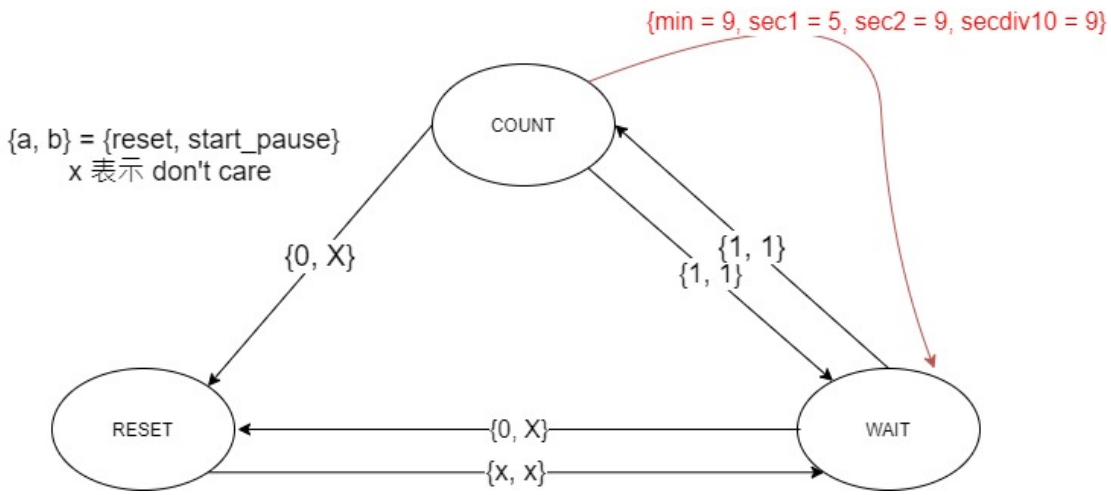
Sequential 電路，就 reset==1’b0 的話直接跳到 RESET state。不然就去 next_state。

```
250   always @(*)begin
251       case (state)
252           COUNT:
253               if(min == 4'd9 && sec1 == 4'd5 && sec2 == 4'd9 && secdiv10 == 4'd9)
254                   next_state = WAIT;
255               else begin
256                   if(start_pause == 1'b1)
257                       next_state = WAIT;
258                   else
259                       next_state = COUNT;
260               end
261           WAIT:
262               if(start_pause == 1'b1)
263                   next_state = COUNT;
264               else
265                   next_state = WAIT;
266           RESET:
267               next_state = WAIT;
268       default:
269           next_state = RESET;
270       endcase
271   end
272   endmodule
```

決定 next_state 的 combinational 電路，根據現在 state 的值以及 start_pause 這個按鈕 input，決定下一個 state 是什麼。

意外的是，在 COUNT state，是有多判斷現在的時間來決定，因為 overflow 之後要停在 0:00.0 的緣故，所以其實 input 看了 4 個東西。只是時間這個 input 在其他 state 被省略進 else 裏頭了。

下方是 state transition diagram。



基本上如果寫的{a,b}的部分，就是

```
always @ (posedge clk_216) begin
    if(not_rst_n_dp_op == 1'b0) begin
        an <= 4'b1110;
        case (an)
        4'b1110:
        begin
            an <= 4'b0111;
            case (num)
            4'b0000: out <= 7'b000_0001;
            4'b0001: out <= 7'b000_0001;
            4'b0010: out <= 7'b000_0001;
            4'b0011: out <= 7'b000_0001;
            4'b0100: out <= 7'b000_0001;
            4'b0101: out <= 7'b000_0001;
            4'b0110: out <= 7'b000_0001;
            4'b0111: out <= 7'b000_0001;
            4'b1000: out <= 7'b000_0001;
            4'b1001: out <= 7'b000_0001;
            4'b1010: out <= 7'b100_1111;
            4'b1011: out <= 7'b100_1111;
            4'b1100: out <= 7'b100_1111;
            4'b1101: out <= 7'b100_1111;
            4'b1110: out <= 7'b100_1111;
            4'b1100: out <= 7'b100_1111;
            4'b1101: out <= 7'b100_1111;
            4'b1110: out <= 7'b100_1111;
            4'b1111: out <= 7'b100_1111;
            default : out <= 7'b101_1111;
            endcase
        end
        4'b0111:
        begin
            an <= 4'b1011;
            case (num)
            4'b0000: out <= 7'b000_0001;
            4'b1010: out <= 7'b000_0001;
            4'b0001: out <= 7'b100_1111;
            4'b1011: out <= 7'b100_1111;
            4'b0010: out <= 7'b001_0010;
            4'b1100: out <= 7'b001_0010;
            4'b0011: out <= 7'b000_0110;
            4'b1101: out <= 7'b000_0110;
            4'b0100: out <= 7'b100_1100;
            4'b1110: out <= 7'b100_1100;
            4'b0101: out <= 7'b010_0100;
            4'b1111: out <= 7'b010_0100;
            4'b0110: out <= 7'b010_0000;
            4'b0111: out <= 7'b000_1111;
            4'b1000: out <= 7'b000_0000;
            4'b1001: out <= 7'b000_0100;
            default : out <= 7'b011_1111;
            endcase
        end
        4'b1011:
        begin
            an <= 4'b1101;
            out <= 7'b001_1101;
        end
        4'b1101:
        begin
            an <= 4'b1110;
            out <= 7'b001_1101;
        end
        default : out <= 7'b111_0111;
        endcase
```

用 2 的 16 次方 clk，讓肉眼觀察不到各個 digit 跳動。

若 reset，an 從 3 跑到 0，輸出各個 digit 的 num and direction

output。

```
4'b0111:
begin
    an <= 4'b1011;
    case (num)
    4'b0000: out <= 7'b000_0001;
    4'b1010: out <= 7'b000_0001;
    4'b0001: out <= 7'b100_1111;
    4'b1011: out <= 7'b100_1111;
    4'b0010: out <= 7'b001_0010;
    4'b1100: out <= 7'b001_0010;
    4'b0011: out <= 7'b000_0110;
    4'b1101: out <= 7'b000_0110;
    4'b0100: out <= 7'b100_1100;
    4'b1110: out <= 7'b100_1100;
    4'b0101: out <= 7'b010_0100;
    4'b1111: out <= 7'b010_0100;
    4'b0110: out <= 7'b010_0000;
    4'b0111: out <= 7'b000_1111;
    4'b1000: out <= 7'b000_0000;
    4'b1001: out <= 7'b000_0100;
    default : out <= 7'b011_1111;
    endcase
```

```
else begin
    case (an)
    4'b1110:
    begin
        an <= 4'b0111;
        case (num)
        4'b0000: out <= 7'b000_0001;
        4'b0001: out <= 7'b000_0001;
        4'b0010: out <= 7'b000_0001;
        4'b0011: out <= 7'b000_0001;
        4'b0100: out <= 7'b000_0001;
        4'b0101: out <= 7'b000_0001;
        4'b0110: out <= 7'b000_0001;
        4'b0111: out <= 7'b000_0001;
        4'b1000: out <= 7'b000_0001;
        4'b1001: out <= 7'b000_0001;
        4'b1010: out <= 7'b100_1111;
        4'b1011: out <= 7'b100_1111;
        4'b1100: out <= 7'b100_1111;
        4'b1101: out <= 7'b100_1111;
        4'b1110: out <= 7'b100_1111;
        4'b1111: out <= 7'b100_1111;
        default : out <= 7'b101_1111;
```

```
4'b1011:
begin
    an <= 4'b1101;
    if(dir == 1'b1) out <= 7'b001_1101;
    else out <= 7'b110_0011;
end
4'b1101:
begin
    an <= 4'b1110;
    if(dir == 1'b1) out <= 7'b001_1101;
    else out <= 7'b110_0011;
end
default : an <= 4'b1110;
endcase
```

Reset 不等於 0，an 從 3 到 0，依 counter 輸出的 out 和 direction 決定各 digit 的 output。

開發過程中的問題/學習：

因為想直觀的做 seven segment 輸出，就用很冗的 case 寫，在燒進板子時，因為 reset 是 0 的時候觸發，抓了很久的 bug 才發現。Debounce 和 one_pulse 一開始都用最慢的 clk 當參數，導致要按很久，才有反應。最後應該兩個都傳入原始 clk，使得一按下去就觸發，再把 output 訊號做延長，但一直想不到怎麼正確的延長訊號。最後才知道用一個新的 counter 算固定時間，以那個 counter 做延長。我們有用 led 燈確認 counter 有無正確運行及按鈕的訊號值，讓 debug 輕鬆不少。

分工

高敦晉 : FPGA

陳皇佑 : AQ1, AQ2