Lab 4

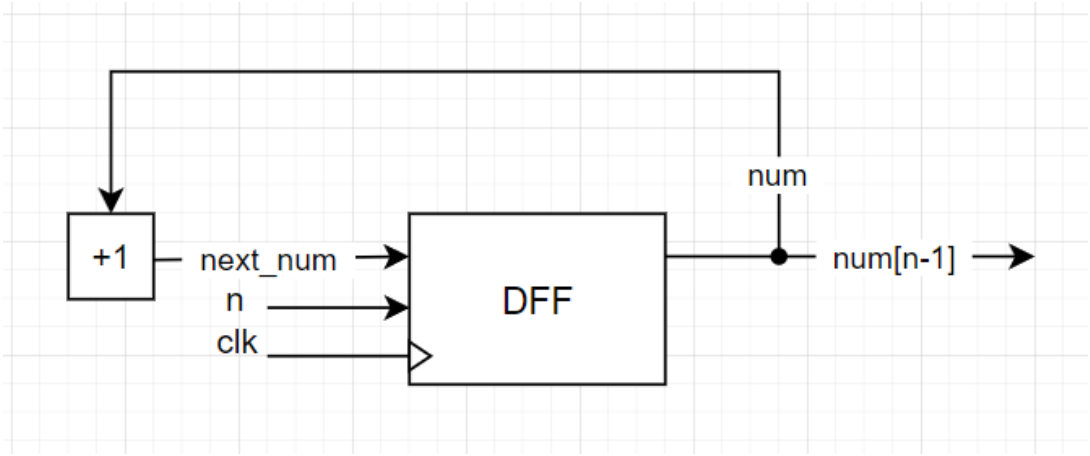
學號: 109062173 姓名: 葉昱揚

A. Lab Implementation

1. Block diagram of the design with an explanation.

Lab 的 block diagram 很大，我先把幾個小 module 獨立畫出來。

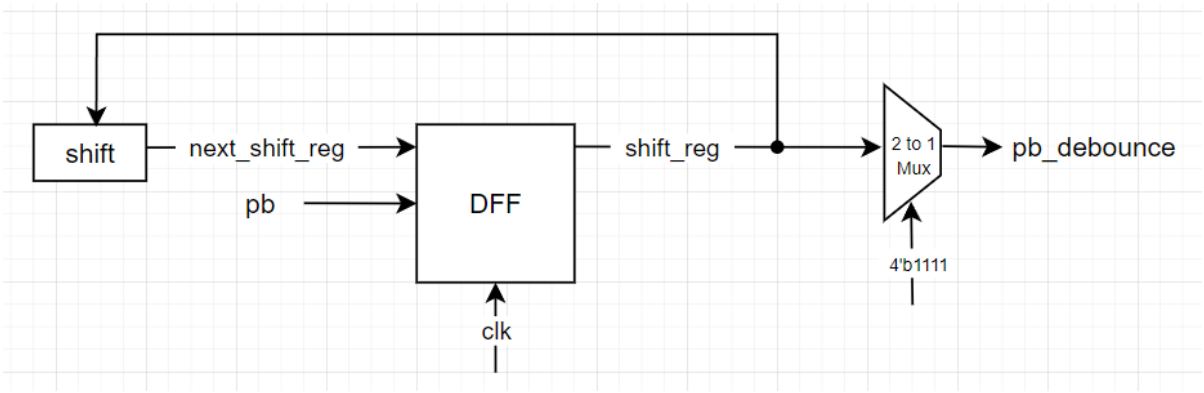
*Clock_divider block diagram:



Input: clk ,n / output: num[n-1]

原理是每次 clk 的 posedge 時，將 counter num 的值+1。且因為 num 是 n bit register，num[0]、num[1]、num[2]、……、num[n-1]分別代表不同切割頻率的 clk，num[n-1]代表 clk/(2^n)頻率的 clock。

*Debounce block diagram



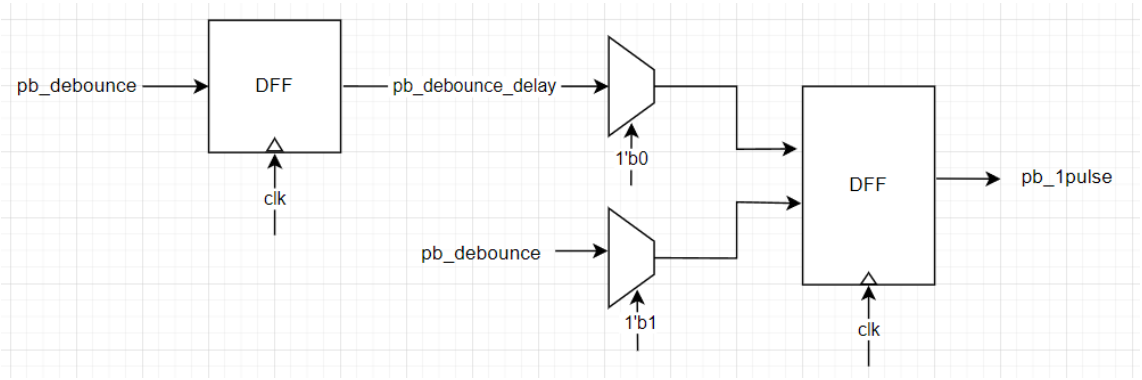
Input:clk,pb / output:pb_debounce

本次 lab 中，需要使用 fpga 版上的 button。因為 button 內部有彈簧，當我們按壓 / 放開 button 時它會回彈振動。舉個例子來說，下壓時理想狀況之下會得到訊號 1，但因為彈簧振動，button 實際上介於下壓與放開之間的狀態，也就是我們會得到不穩定的 0/1 訊號，而非理想情況下的穩定 0/1 訊號。

上圖的 block diagram 利用 register 儲存壓下 button 後的 pb 訊號，並將 register shift 一格已儲存下一次的 pb 訊號。而 pb_debounce 的處理相當簡單，我會判斷 register 裡面的值 ：

1) 若不全部為 1，代表 pb 給了一串不穩定的訊號，register 裡有 0 與 1，代表 button 還在彈簧振動的階段，此時 pb_debounce 為 0。
2) 若全部為 1，代表 pb 給了一串穩定的連續訊號 1，此時 pb_debounce 為 1。

*One pulse block diagram

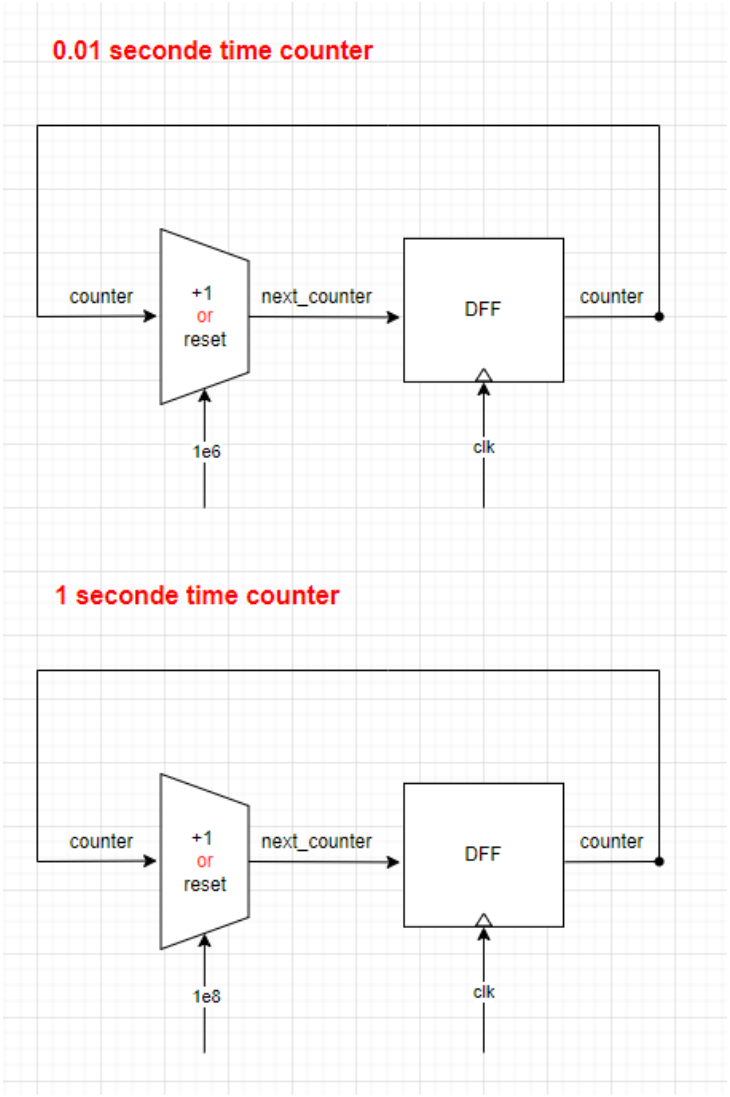


Input: clk,pb_debounce / output:pb_1_pulse

首先會將 pb_debounce 利用 DFF 延遲一個 cycle，稱為 pb_debounce_delay。再來判斷 pb_debounce_delay 是否為 0、pb_debounce 是否為 1:

1) 若是，pb_1pulse 為 1。
2) 若否，pb_1pulse 為 0。

*time counter block diagram

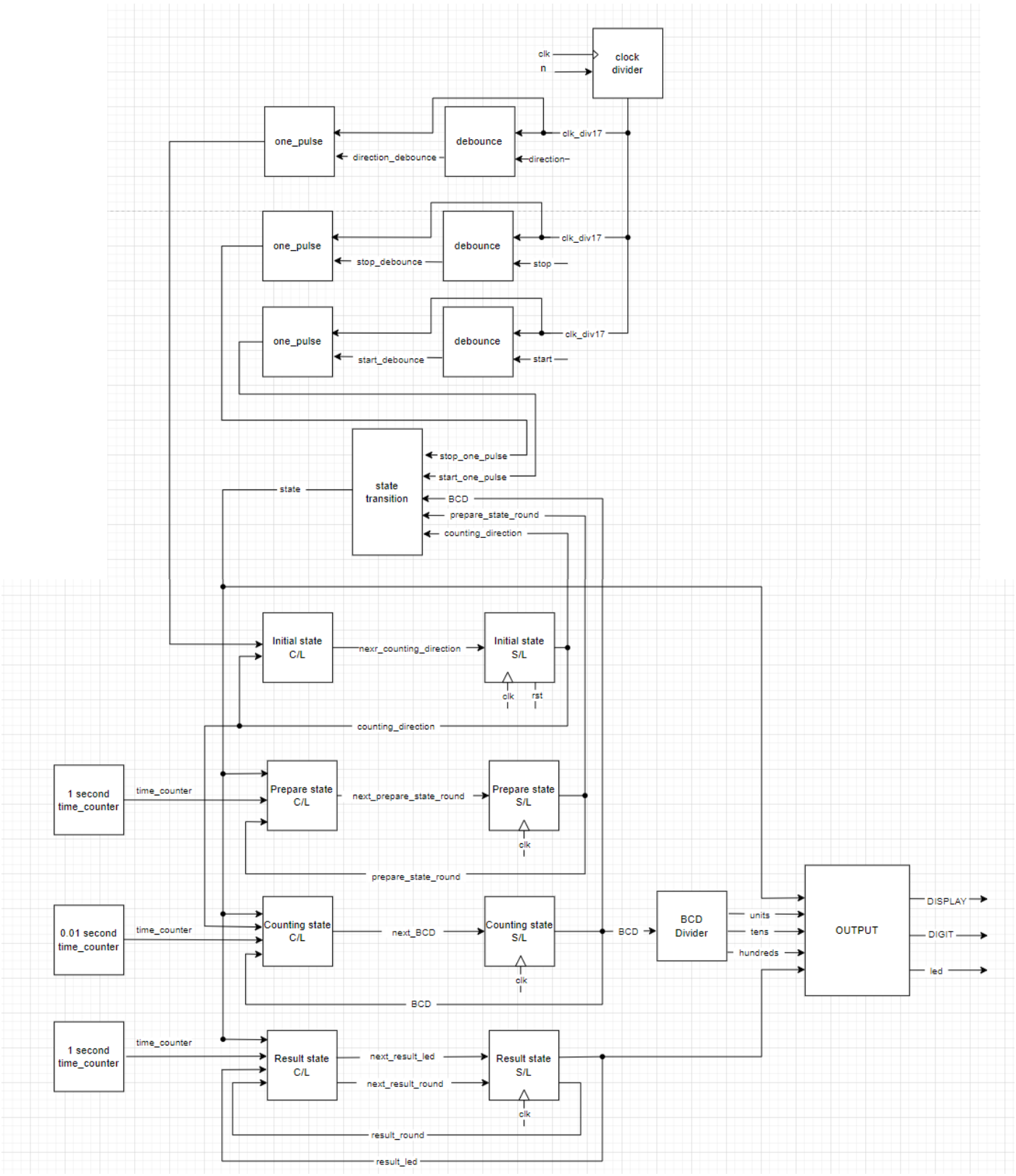


上圖是 lab 中用到的 time_counter block diagram，用途是計算準確的 0.01s、1s。

首先我們知道 fpga board 提供的 clk frequency 是 100Mhz，也就是每秒鐘 10^8 個 cycle，能夠反推出 0.01s 需要 10^6 個 cycle，1s 需要 10^8 個 cycle。

於是設計一個 counter 計數，並用 100Mhz 的 clk trigger。
當 counter 為 10^6 的時候代表經過 0.01s； 當 counter 為 10^8 的時候代表經過 1s。
如此設計出計算準確時間的 module。

*lab4_1



上圖是 lab4_1 的 block diagram。

在這個 lab 中有 4 個 state，分別為 : Initial、Prepare、Counting、Result

Initial:

這裡是 FSM 的起始點，rst 時會回到這個 state。

這裡會判斷 direction button 是否被 push，可以看到有一條處理好 direction button 訊號的 direction_one_pulse 接到這個 state:

若偵測到 direction_one_pulse，則改變 counting_direction。

若沒偵測到 direction_one_pulse，counting_direction 維持不變。

同時這代表只有在 Initial state 可以改變 counting_directoin，其他 state 壓下 direction button 都是無效操作。

以下情況發生時會進入 Prepare state:

1) 當按下 start button 時會進入 Prepare state。

Prepare:

這個 state 要等待 3 秒，3 秒過後自動進入 Counting state。

於是這邊使用 1 second time counter，time_counter 的 block diagram 已經在上面 report 解釋過了，這邊用 time_counter 來計算”準確”經過的時間，並用 prepare_state_round 來記錄總共經過幾秒。

以下情況發生時會進入 Counting state:

1) 當 prepare_state_round = 3 時代表已經等待了 3 秒，可以進入 Counting state。

Counting:

在這個 state，BCD 會根據 counting_direction 的值決定計算方向。

1) 若 counting_direction=DOWN，BCD 會從 999 往下數，直到 0 為止。
2) 若 counting_direction=UP，BCD 會從 0 往上數，直到 999 為止。

BCD 每經過 0.01 second 就會遞增/遞減，因此這邊使用設計好的 0.01 second time counter 計算準確的時間 ！

BCD 會經過簡單的除法和模運算得到百位、十位、個位數並在 Output 的地方顯示出來
(百位=BCD/100 ; 十位=(BCD/10)%10 ; 個位=BCD%10)

以下情況發生時會進入 Result state:

1) BCD 為 999 且 counting_direction 為 UP
2) BCD 為 0 且 counting_direction 為 DOWN
3) 按下 stop button 時

Result:

這個 state 的 7 segment 會顯示 counting state 的 BCD 值，且 led 燈會間隔一秒顯示”on off on off on”
因此這邊用了 1 second time counter 來控制 led 燈的輸出，result_round 紀錄目前經過幾秒。

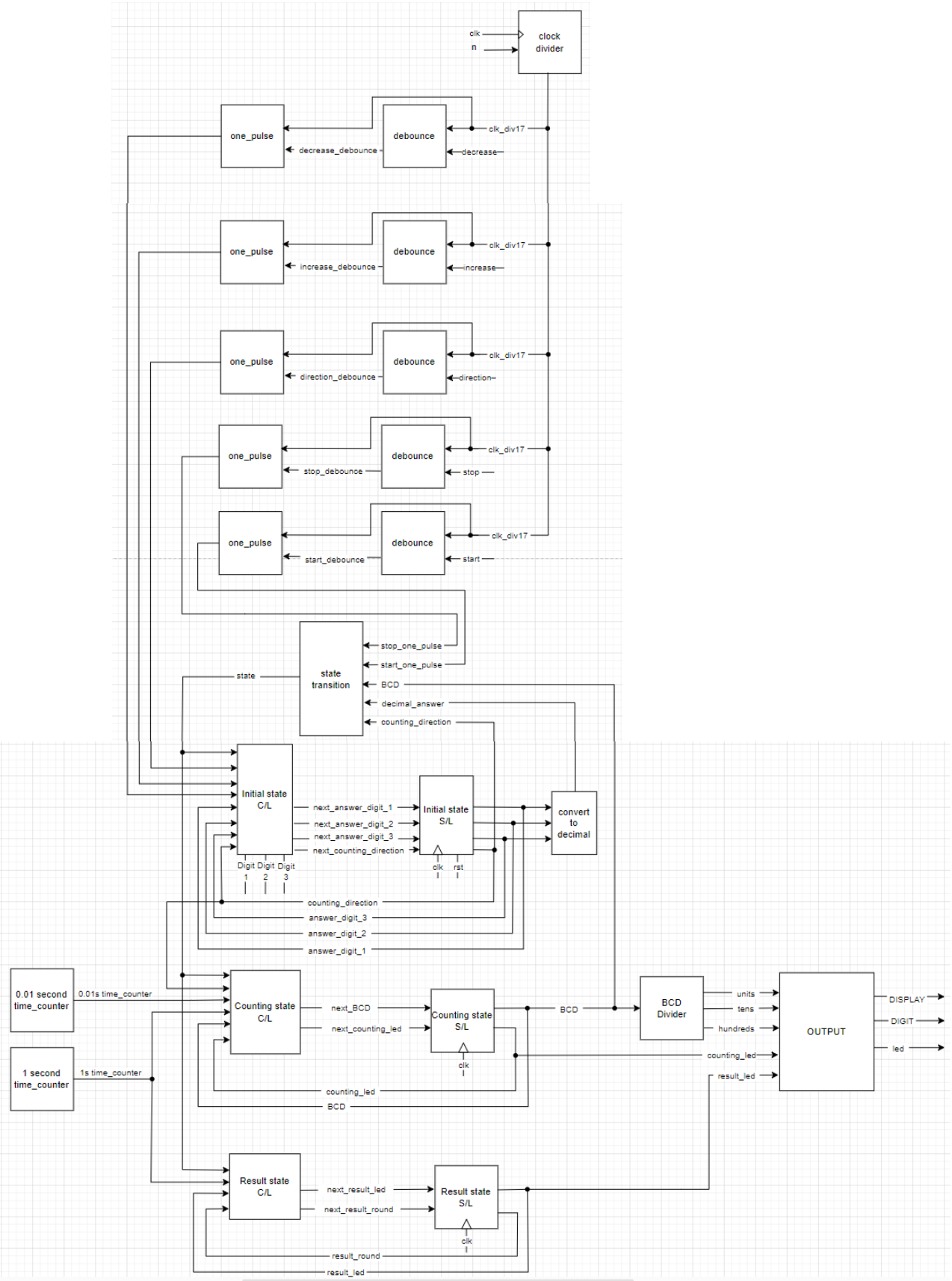
以下情況發生時會進入 Initial state:

1) 按下 start button 時

State_transition:

已整合在各 state 的 ”以下情況發生時會進入 xxxxx state” 區段。

*lab4_2



上圖是 lab4_2 的 block diagram。

在這個 lab 中有 4 個 state，分別為 : Initial、Counting、Success、Fail。

其中我把 Success、Fail 合併寫在上圖的 Result 裡面。

Initial:
這裡是 FSM 的起始點，rst 時會回到這個 state。
這裡會判斷 direction button 是否被 push，可以看到有一條處理好 direction button 訊號的 direction_one_pulse 接到這個 state:
若偵測到 direction_one_pulse，則改變 counting_direction。
若沒偵測到 direction_one_pulse，counting_direction 維持不變。
同時這代表只有在 Initial state 可以改變 counting_directoin，其他 state 壓下 direction button 都是無效操作。

除此之外，Digit_1、Digit_2、Digit_3 可以選擇要控制得位數，當與位數對應的 Digit switch 被拉起時，push increase / decrease button 就可以遞增/遞減該位數。

以下情況發生時會進入 Prepare state:
1) 當按下 start button 時會進入 Prepare state。

Counting:
在這個 state，BCD 會根據 counting_direction 的值決定計算方向。
1) 若 counting_direction=DOWN，BCD 會從 999 往下數，直到 0 為止。
2) 若 counting_direction=UP，BCD 會從 0 往上數，直到 999 為止。

BCD 每經過 0.01 second 就會遞增/遞減，因此這邊使用設計好的 0.01 second time counter 計算準確的時間 !
BCD 會經過簡單的除法和模運算得到百位、十位、個位數並在 Output 的地方顯示出來
(百位=BCD/100 ; 十位=(BCD/10)%10 ; 個位=BCD%10)

進入這個 state 0~3 秒間，7-segement-display 顯示 BCD 的，led 亮起。
進入這個 state 3 秒後，7-segement-display 顯示 Dash，led 熄滅。
於是用 1 second time counter 計算準確時間，以此控制輸出。

以下情況發生時會進入 Success state:
1) 按下 stop button，且 BCD 介於 the number we set in the INITIAL state” ± 100,

以下情況發生時會進入 Fail state:
1) 按下 stop button，且 BCD 不介於 the number we set in the INITIAL state” ± 100,
2) BCD 為 999 且 counting_direction 為 UP
3) BCD 為 0 且 counting_direction 為 DOWN

Success:

這個 state 的 7 segment 會顯示 S 和 counting state 的 BCD 值，且 led 燈會間隔一秒顯示”on off on off on”，因此這邊用了 1 second time counter 來控制 led 燈的輸出，result_round 紀錄目前經過幾秒。

以下情況發生時會進入 Initial state:
1) 按下 start button 時

Fail:

這個 state 的 7 segment 會顯示 F 和 counting state 的 BCD 值，且 led 燈會間隔一秒顯示”on off on off on off”，因此這邊用了 1 second time counter 來控制 led 燈的輸出，result_round 紀錄目前經過幾秒。

以下情況發生時會進入 Initial state:
1) 按下 start button 時

Success & Fail

合併寫在 result state 的原因是它們兩個只差在 led 燈多熄滅一次和 7-segment display 顯示 F/S，前者只需要靠 result_round 判斷現在亮起/熄滅 led 幾次就可以控制，後者靠 state 判斷就可以解決 !

State transition:
已整合在各 state 的 ”以下情況發生時會進入 xxxxx state” 區段。

2. Partial code screenshot with the explanation

*lab4_1

```
//////////////////////////////// (basay3 7-segment) ////////////////////////////////
reg [3:0] value, next_value, next_DIGIT;
always@(posedge clk_div17) begin
    value <= next_value;
    DIGIT <= next_DIGIT;
end
always@(*) begin
    case(state)
        COUNTING,RESULT:begin
            case (DIGIT)
                4'b1110: begin
                    next_value = tens;
                    next_DIGIT = 4'b1101;
                end
                4'b1101: begin
                    next_value = hundreds;
                    next_DIGIT = 4'b1011;
                end
                4'b1011: begin
                    next_value = counting_direction;
                    next_DIGIT = 4'b0111;
                end
                4'b0111: begin
                    next_value = units;
                    next_DIGIT = 4'b1110;
                end
                default: begin
                    next_value = units;
                    next_DIGIT = 4'b1110;
                end
            endcase
        end
    endcase
end
////////////////////////////////////////////////////////////////////////////////
```

上圖是 7-segment display 在 counting state 和 result state 的操作。
相較於其他的 state，這邊需要處理 BCD，擷取出百位、十位、個位，並分配在各個 7-segment。
且分頻 clk_div17 讓 7-segment 能夠在不同位數顯示不同的數字。

```
always @(posedge clk) begin
    counting_time_counter <= next_counting_time_counter ;
end

always @(*) begin
    if(state==INITIAL) begin
        next_counting_time_counter = 0;
    end
    else if(state==COUNTING) begin
        if (counting_time_counter == 10**6-1'b1) next_counting_time_counter = 0;
        else next_counting_time_counter = counting_time_counter + 1'b1;
    end
    else begin
        next_counting_time_counter = counting_time_counter;
    end
end
```

上圖是使用於 counting state 的 0.01 秒的 counter。
可以看到 couting_time_counter 這個計數器會逐漸遞增，且用 100Mhz 的 clk trigger，當數到 10^6-1 的時候代表準確的 0.01 秒經過，應用於 BCD 遞增/遞減。

1 second time counter 同理，counter 數到 10^8-1 的時候代表準確的 1 秒經過。

```
always@(posedge clk) begin
    BCD <= next_BCD;
end

reg [26:0] counting_time_counter, next_counting_time_counter;
always@(*) begin
    if(state==PREPARE) begin
        next_BCD = (counting_direction==UP) ? 10'd0 : 10'd999;
    end
    else if(state==COUNTING && (counting_time_counter == 10**6-1'b1)) begin
        if(counting_direction==DOWN && BCD==10'd0) next_BCD = 0;
        else if(counting_direction==UP && BCD==10'd999) next_BCD = 999;
        else next_BCD = (counting_direction==UP) ? BCD + 1 : BCD - 1;
    end
    else begin
        next_BCD = BCD;
    end
end
```

上圖是使用 0.01 秒 time counter 來遞增遞減 BCD 的 code segment。

重點就是 else if 裡面的 counting_time_counter==10**6-1，進去後再判斷 counting_direction 把 BCD +1/-1。

*lab4_2

```
always@(posedge clk,posedge rst) begin
    if(rst) begin
        answer_digit_1 <= 0;
        answer_digit_2 <= 0;
        answer_digit_3 <= 0;

    end
    else begin
        answer_digit_1 <= next_answer_digit_1;
        answer_digit_2 <= next_answer_digit_2;
        answer_digit_3 <= next_answer_digit_3;
    end
end
always@(*) begin
    if(state==INITIAL) begin
        /* answer digit 1*/
        if(Digit_1==1 && increase_one_pulse) next_answer_digit_1 = (answer_digit_1==4'd9) ? 4'd0 : answer_digit_1+1'b1;
        else if(Digit_1==1 && decrease_one_pulse) next_answer_digit_1 = (answer_digit_1==4'd0) ? 4'd9 : answer_digit_1-1'b1;
        else next_answer_digit_1 = answer_digit_1;

        /* answer digit 2*/
        if(Digit_2==1 && increase_one_pulse) next_answer_digit_2 = (answer_digit_2==4'd9) ? 4'd0 : answer_digit_2+1'b1;
        else if(Digit_2==1 && decrease_one_pulse) next_answer_digit_2 = (answer_digit_2==4'd0) ? 4'd9 : answer_digit_2-1'b1;
        else next_answer_digit_2 = answer_digit_2;

        /* answer digit 3*/
        if(Digit_3==1 && increase_one_pulse) next_answer_digit_3 = (answer_digit_3==4'd9) ? 4'd0 : answer_digit_3+1'b1;
        else if(Digit_3==1 && decrease_one_pulse) next_answer_digit_3 = (answer_digit_3==4'd0) ? 4'd9 : answer_digit_3-1'b1;
        else next_answer_digit_3 = answer_digit_3;
    end
    else begin
        next_answer_digit_1 = answer_digit_1;
        next_answer_digit_2 = answer_digit_2;
        next_answer_digit_3 = answer_digit_3;
    end
end
```

上圖是 initial state 對 3 個 digit 的操作。
可以看到 3 個 digit 的判斷完全一樣，都是判斷對應的 switch 是否有被拉起，再判斷 increase/decrease button 有沒有被 push。如此可以同時操作 0~3 個 digit 共同 +/- 值。

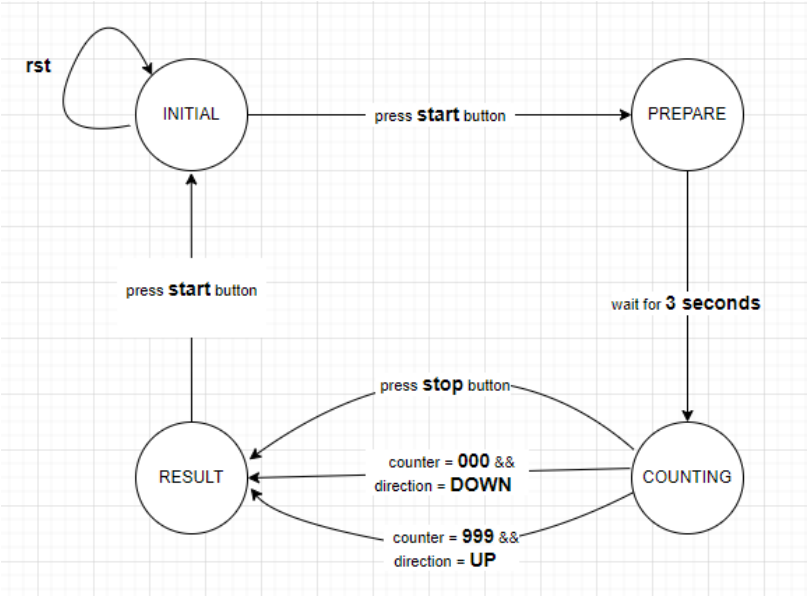
```
always @(*) begin
    if(state==INITIAL) begin
        next_result_time_counter = 0;
        next_result_led = ALL_LED_LIGHT;
        next_result_round = 0;
    end
    else if((state==SUCCESS) && result_round < 3'd4) begin
        if (result_time_counter == 10**8-1'b1) begin
            next_result_time_counter = 0;
            next_result_led = ~result_led;
            next_result_round = result_round + 1;
        end
        else begin
            next_result_time_counter = result_time_counter + 1'b1;
            next_result_led = result_led;
            next_result_round = result_round;
        end
    end
    else if((state==FAIL) && result_round < 3'd5) begin
        if (result_time_counter == 10**8-1'b1) begin
            next_result_time_counter = 0;
            next_result_led = ~result_led;
            next_result_round = result_round + 1;
        end
        else begin
            next_result_time_counter = result_time_counter + 1'b1;
            next_result_led = result_led;
            next_result_round = result_round;
        end
    end
    else begin
        next_result_time_counter = result_time_counter;
        next_result_led = result_led;
        next_result_round = result_round;
    end
end
```

上圖是 Success state(紅框)和 Fail state(黃框)的操作

可以看到兩個唯一的差別是判斷 result_round 的部分。
只是因為 fail state 需要多熄滅一次 led 燈，所以在 fail state 的時候 result_round 可以比在 success state 的時候大一個值，如此讓 led 燈再操作一個 cycle。

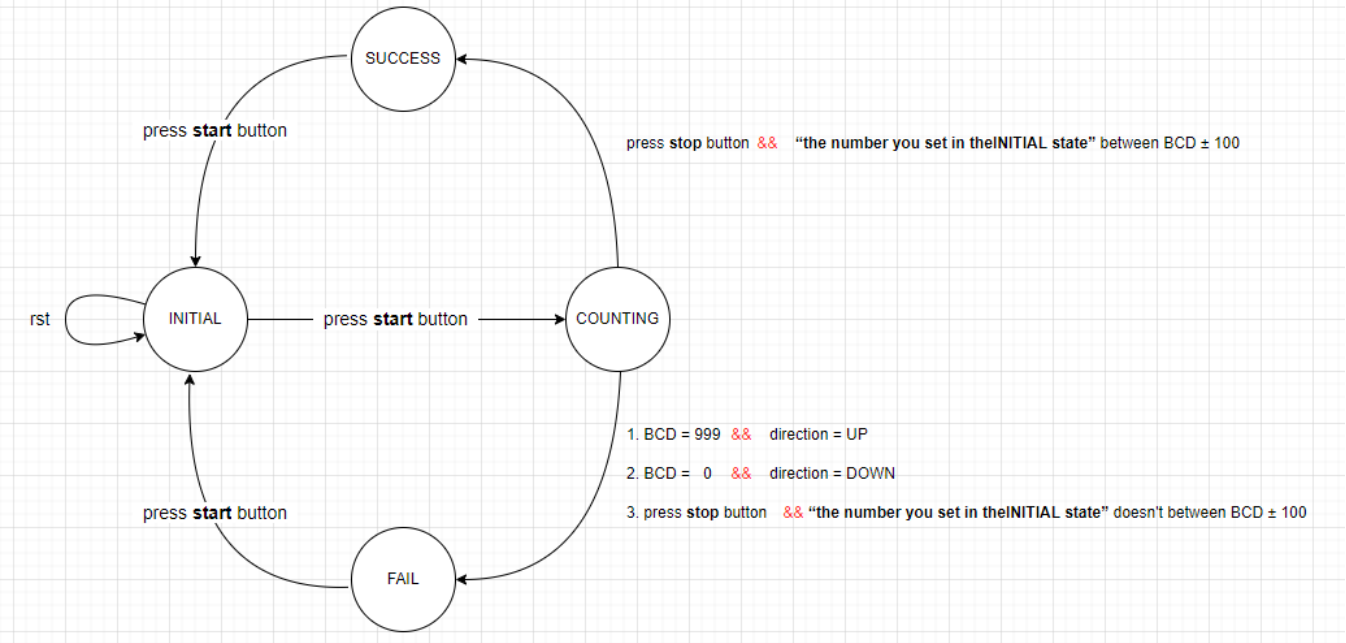
3. Finite state machine (FSM) with an explanation.

*lab4_1



Initial:
　　當按下 start button 時會進入 Prepare state。
Prepare:
　　當 prepare_state_round = 3 時進入 Counting state。
Counting:
　　BCD 為 999 且 counting_direction 為 UP 時進入 Result state。
　　BCD 為 0 且 counting_direction 為 DOWN 時進入 Result state。
　　按下 stop button 時進入 Result state。
Result:
　　按下 start button 時進入 Initial state。

*lab4_2



Initial:

按下 start button 時進入 Initial state.

Counting:

按下 stop button，且 BCD 介於 the number we set in the INITIAL state" ± 100,進入 Success state.

按下 stop button，且 BCD 不介於 the number we set in the INITIAL state" ± 100 進入 Fail state.

BCD 為 999 且 counting_direction 為 UP 進入 Fail state.

BCD 為 0 且 counting_direction 為 DOWN 進入 Fail state.

Success:

按下 start button 時進入 Initial state.

Fail:

按下 start button 時進入 Initial state.

B. Questions and Discussions

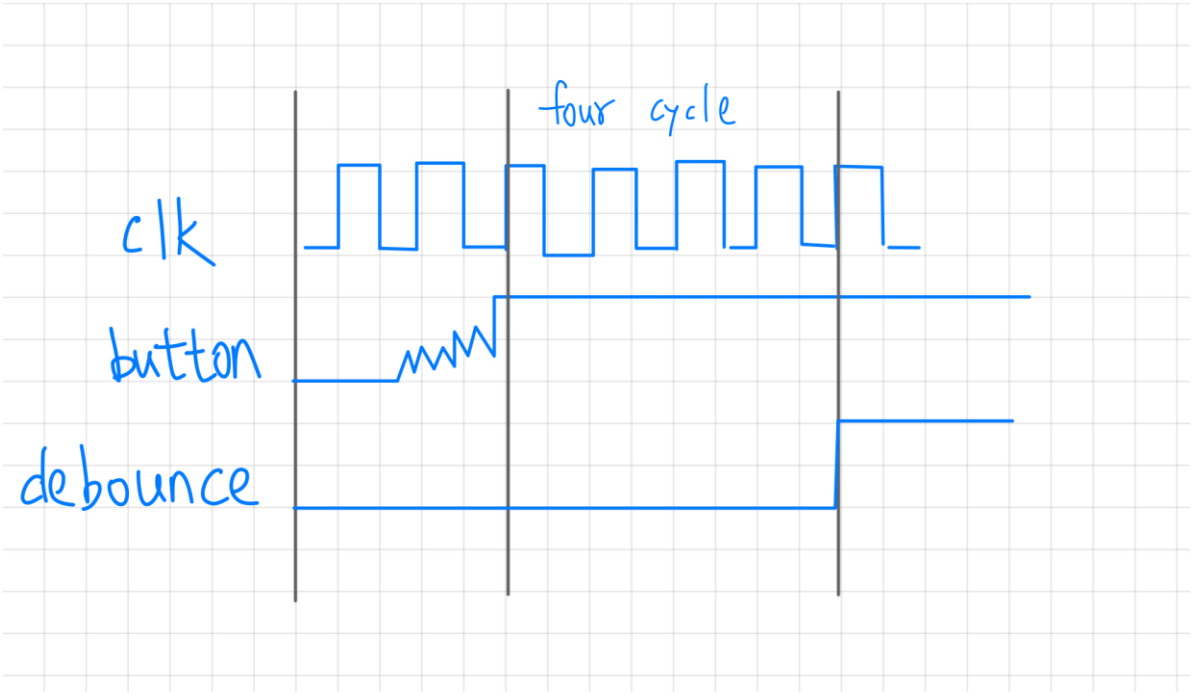
A. Why do we need the debounce and one-pulse modules? What is the relation between the FSM's clock rate and the one-pulse module's clock rate? Explain your reason. And what will happen when we implement with a wrong clock rate? (Please draw waveforms to explain)

因為 button 內部有彈簧，當我們按壓 / 放開 button 時它會回彈振動。舉個例子來說，下壓時理想狀況之下會得到訊號 1，但因為彈簧振動，button 實際上介於下壓與放開之間的狀態，也就是我們會得到不穩定的 0/1 訊號，而非理想情況下的穩定 0/1 訊號。

Debounce 就是要處理這些不穩定訊號，這他們變成連續的 0/1 訊號。

One pulse 則是因為如果 FSM 間的切換都是判斷某個 button 有沒有被壓下，可能會因為壓下一次 button 後訊號持續時間過長，造成連續跳躍多個 state。但實際上我們只想要 "壓一次 button，切換一個 state"這樣的結果，而非"壓一次 button，切換多個 state"，one pulse 就是產生 exactly one cycle，確保只會跳換一個 state，避免上述情況發生。

如果使用 wrong clock rate，在板子上 demo 時可能造成卡頓感。
比方說，若使用非常慢的 clk 做 debounce，按下 button 後板子可能不會立刻反應，可能要等個幾秒才有反應。假設 debounce 時 register 存了 4 個 value，因為 register 需要 shift 後存入 pb，它需要 4 個 cycle 才能清掉就有全部的 value，壓下 button 後就要等待至少 4 個 cycle，板子才會顯示對應的操作，如下圖 ：



B. Please propose two different design methods to manipulate two or more clocks in one module.

Method1:
用頻率更快的 clk trigger，但在內部使用 counter 來放慢我們要作的 operation。
比方說 always block 用 100Mhz 的 clk trigger，但我可以在內部設計一個 counter，當它從 0 數到某個值的時候再作 operation，利用 counter 來判斷是否作 operation，這代表 operation 執行的速度由 counter 決定，而非單一 clk 決定。

Method2:
用 n to 1 mux 選擇要使用的 clock 即可。
比方說現在有 clk_div1、clk_div2、……、clk_divn，它們都是一個 n to 1 mux 的 input，再拉一條線來當 selector，選擇要哪個 clk 當作 output 即可。

C. Problem Encountered

最初實作 exactly 0.01 / 1 seconds 的時候遇到問題。
起初是寫了一個每過 0.01s 就會 trigger 的 clk_0.01 出來，並拿它來 trigger counting state。

```
module exactly_clock_divider #(
    parameter target_frequency=1
)(
    input clk_in,
    output reg clk_out
);
reg [26:0] counter=0, next_counter;
reg [26:0] MAX=10**6/target_frequency/2;
always @(posedge clk_in) begin
    if (counter == MAX-1'b1) begin
        counter <= 0;
        clk_out <= ~clk_out;
    end else begin
        counter <= next_counter;
        clk_out <= clk_out;
    end
end
always @(*) begin
    next_counter = counter + 1'b1;
end
endmodule
```

上圖的 module 可以 output 出 0.01s / 1s trigger 的 clk。
但後來發現每個 state 都要用同頻率的 clk trigger 才對。後續才改成進入 counting state 後在裡面用 counter 計數，而不是直接用 0.01s trigger 1 次的 clk_0.01。

在 prepare state 一開始也是用 1s trigger 的 clk 來跑，發現從 initial 轉到 prepare 後不會等待 3 秒，有時會瞬間進入 counting state，才發現其他 state 的 clk 在跑的時候，prepare state 的 clk 也正在跑，因此很難準確地待滿 3 秒。後續才改成進入 prepare state 後在裡面用 counter 計數，而不是直接用 1s trigger 1 次的 clk 跑。

D. Suggestions

No suggestions. I have a good lab implementation experience. Thanks for TA.

And here is the meme this week.