Lab 3

學號: 110011138 姓名: 楊立慈

A. Lab Implementation

(a) Module: clock_divider

1. Fig.1 為此 module 的 block diagram，基本上就是一個會在 clk 的 positive edge 來時數到下一個數的 n-bit counter，並從$00\cdots0$數到$11\cdots1$，然後再回到$00\cdots0$重新數。Comb. Circuit 負責計算 counter + 1 並傳給 next_counter，同時把 counter 的第 n 個 bit 傳給 clk_div。

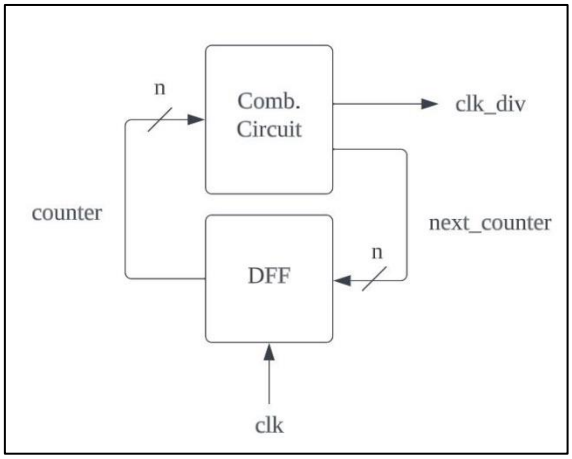

Fig. 1: clock_divider block diagram

2. Fig.2 為 clock_divider 的 code，最底下的兩個 assign statement 實作了 Fig.1 中的 Comb. Circuit。而在第 4 行我們用了 module parameter，這樣要對不同 n 實作 clock divider 時，就可以直接在 instantiate 這個 module 時順便給 n 的值就好。

```
3  module clock_divider
4      #(parameter n = 25)
5      (input clk, output clk_div);
6
7      reg [n - 1:0] counter = 0;
8      wire [n - 1:0] next_counter;
9
10     always @(posedge clk) begin
11         counter <= next_counter;
12     end
13
14     assign next_counter = counter + 1;
15     assign clk_div = counter[n - 1];
16 endmodule
```

Fig. 2: clock_divider code

(b) Lab3_1

1. Fig.3 為 lab3_1 的 block diagram。首先我們用了兩個 clock_divider module 並分別給不同的 n parameter 來得到不同 frequency 的 clock，再使用 2:1 MUX 根據不同 speed 選擇不同的 clock 給 local_clk 作為 DFF 的 clock source。主要的 FSM 以 led 訊號作為 state，而 Comb. Circuit 則會根據目前的 state 決定下一個 state 是什麼。

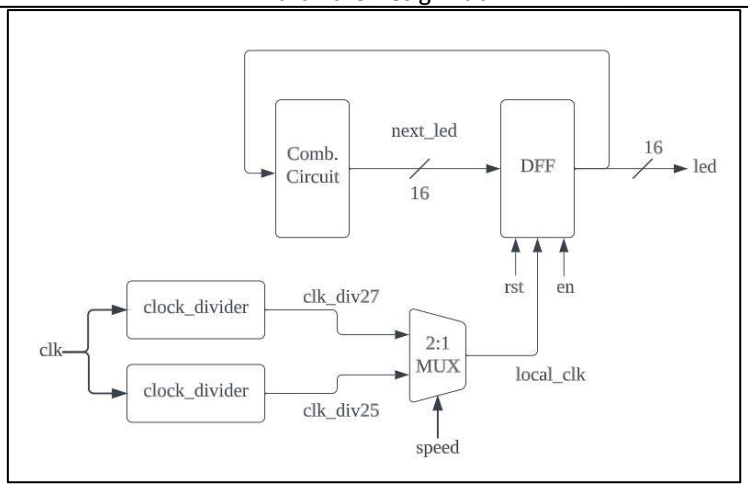

Fig. 3: lab3_1 block diagram

2. Fig.4 為 lab3_1 中 DFF 的 code。此單純參考講義上的，asynchronous positive reset 和 enable 的寫法，至於如何讓 rst 有 highest priority 會在問題與討論解釋。

```
always @(posedge local_clk or posedge rst) begin
    if (rst == 1'b1) begin
        led <= S0;
    end
    else if (en == 1'b1) begin
        led <= next_led;
    end
end
```

Fig. 4: lab3_1 DFF code

Fig.5 為 block diagram 中 Comb. Circuit 的實作，基本上先將 next_led default 成 S0 以防 latch inference，然後再依照目前的 led 值決定 next_led。其中 S0、S1 等等為根據 spec 所想要的 led 亮法所設定的 parameter。

```
always @(*) begin
    next_led = S0;
    case (led)
        S0: next_led = S1;
        S1: next_led = S2;
        S2: next_led = S3;
        S3: next_led = S4;
        S4: next_led = S0;
    endcase
end
```

Fig. 5: lab3_1 Combinational Circuit

3. Fig.6 為 lab3_1 的 state diagram，此處較為單純，當要 reset 時就到 S0 的 state，否則就照順序進到下一個 state，然後 S4 會循環回 S0 再繼續。

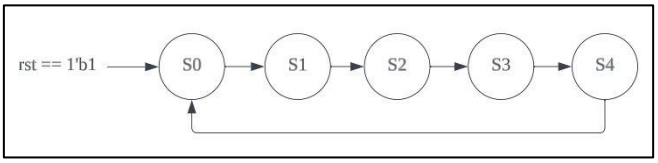

Fig. 6: lab3_1 state diagram

(c) Lab3_2

1. Fig.7 為 lab3_2 的 block diagram。下方根據不同 speed 給不同 clock source 的部分跟 lab3_1 一樣作法。主要不同處在這裡 DFF 的 output 有三個 signal，分別是 16 bit 的 led、2 bit 的 state、4 bit 的 counter，而同樣會由 Comb. Circuit 根據目前的 led、state、counter 值以及 input 的 dir 值決定 next state 的值。

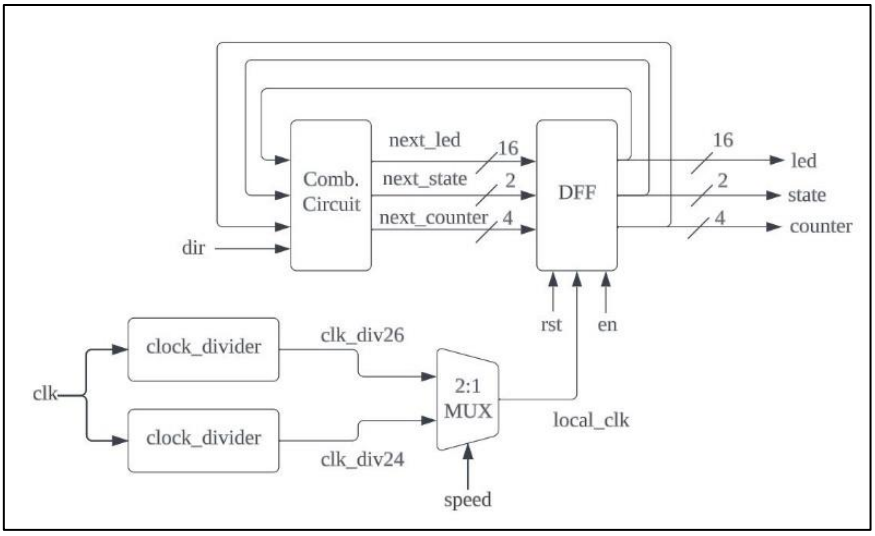

Fig. 7: lab3_2 block diagram

2. Fig.8 為 lab3_2 DFF 的實作 code，同樣為有 async positive reset 以及 enable 的 DFF。在 reset 時會把 state 設為 REGULAR、led 設為 S0、counter 設為 0000。

```
always @(posedge local_clk or posedge rst) begin
    if(rst == 1'b1) begin
        state <= REGULAR;
        led <= S0;
        counter <= 4'b0;
    end else if (en == 1'b1) begin
        state <= next_state;
        led <= next_led;
        counter <= next_counter;
    end
end
```

Fig. 8: lab3_2 DFF code

Fig.9 為 Comb. Circuit 裡面判斷 next_state、next_led、next_counter 的方法，這裡舉當 state == REGULAR 時為例子。由於進到 REGULAR state 時 counter 會是 0，因此我們可以利用 counter 從 0 數到 14 來實現 spec 上的等 3 個 round 才進到 ESCAPE state 的規定。因此當 counter == 4'd14 時，next_state 會是 ESCAPE、next_led 會維持目前的值(全亮)、next_counter 會歸零。否則就代表我們還在進行 3 個 round，因此要把 next_counter 增加一、next_state 一樣是 REGULAR、next_led 依照目前的 led 亮法決定下一個 led 的亮法。

```
case (state)
    REGULAR: begin
        // next_state
        if (counter == 4'd14) next_state = ESCAPE;
        else next_state = REGULAR;

        // next_counter
        if (counter == 4'd14) next_counter = 4'b0;
        else next_counter = counter + 1'b1;

        // next_led
        if (counter == 4'd14) next_led = led;
        else begin
            next_led = led;
            case (led)
                S0: next_led = S1;
                S1: next_led = S2;
                S2: next_led = S3;
                S3: next_led = S4;
                S4: next_led = S0;
            endcase
        end
    end
```

Fig. 9: lab3_2 combinational circuit partial code

3. Fig.10 為 lab3_2 的 state diagram。Reset 後會進入 REGULAR state，然後如前所述 counter 會開始從 0 數到 14 同時更新 led 的亮法，因此當 counter 為 14 時我們才到 ESCAPE state。

   在 ESCAPE state 時，當 dir 是 0 我們要往右方逐漸減少亮的 led，直到 led 全暗且 counter 是 1 時(因為如 spec 所寫要等一個 cycle，這裡我們利用 counter 數一個 cycle)，才進到 SHINING state；當 dir 是 1 我們則往左方逐漸增加亮的 led，直到 led 全亮且 counter 是 1 時(同樣因為要等一個 cycle)，才回到 REGULAR state。

   在 SHINING state 我們同樣用 counter 從 0 數到 9 來等 5 個 led 的 on-off cycles，然後就回到 REGULAR state。

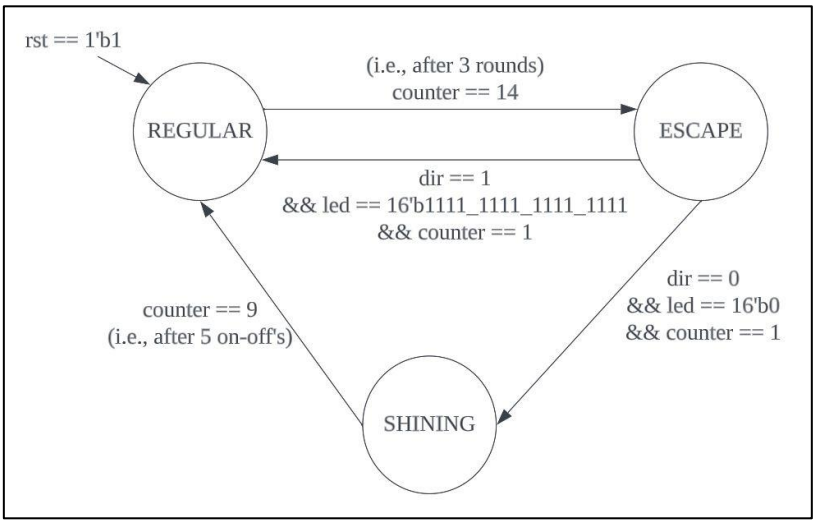

Fig. 10: lab3_2 state diagram

(d) Lab3_3

1. Fig.11 為 lab3_3 的 block diagram。由於每條蛇的 clk 不同，因此我們用了 3 個 DFF 給每條蛇，各自由不同的 clock source 驅動。Comb. Circuit 會算出 next state 的 snake1、snake2、snake3 各自的 position (pos)和 direction (dir)，同時會利用 snake1_pos、snake2_pos、snake3_pos 算出目前 led 的樣子並直接作為 output。

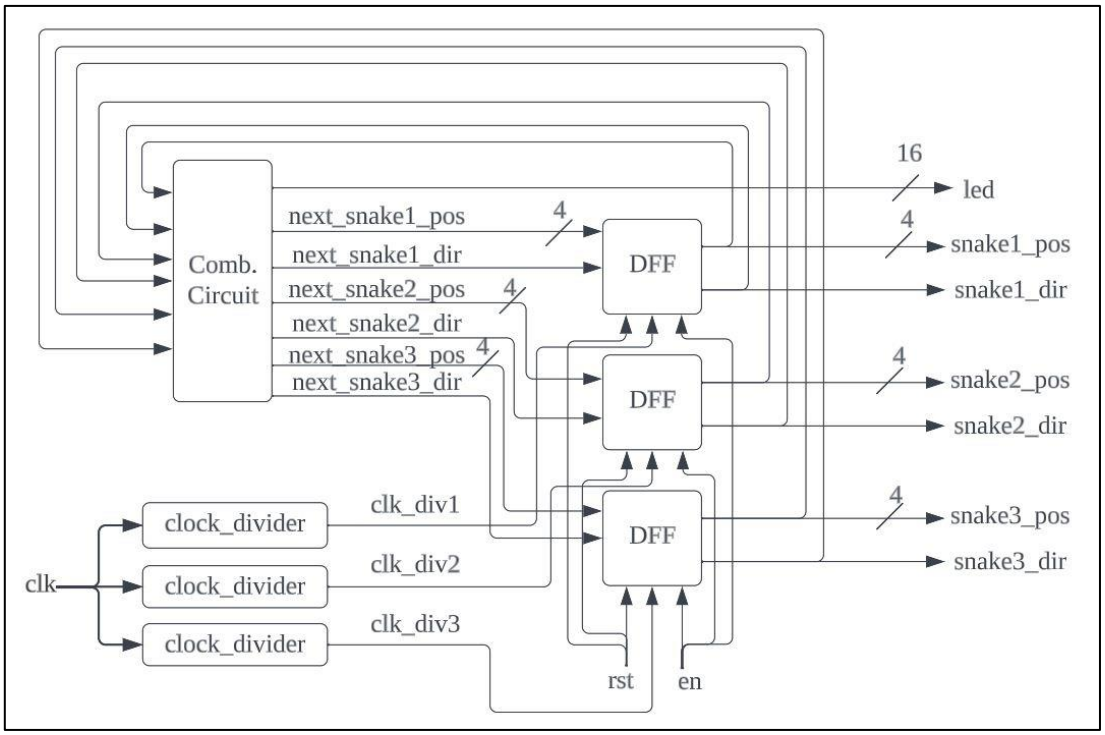

Fig. 11: lab3_3 block diagram

2. 以下以 snake1 為例子說明。Fig.12 為 snake1 的 DFF 實作 code。當 reset 時會把 snake1_pos 設為一開始的位置、snake1_dir 設為往右邊。

```
always @(posedge clk_div1 or posedge rst) begin
    if(rst == 1'b1) begin
        snake1_pos <= SNAKE1_INIT;
        snake1_dir <= RIGHT;
    end else if (en == 1'b1) begin
        snake1_pos <= next_snake1_pos;
        snake1_dir <= next_snake1_dir;
    end
end
```

Fig. 12: lab3_3 snake1 DFF code

Fig.13 為 snake1 的 combinational logic 的 code。我以目前要走的方向分不同 case。要往左走(snake1_dir == LEFT)先確認左邊能不能走(會不會撞牆)，若左邊不能走則確認右邊能不能走(會不會撞到 snake2)，若兩邊都不能走則維持原樣；要往右走(snake1_dir == RIGHT)也是類似的邏輯。然後當原來方向不能走時，要改變 next_snake1_dir 以及 next_snake1_pos 朝反方向走。

```
case (snake1_dir)
    LEFT: begin
        // check if left side is walkable
        if (snake1_pos < LEFT_WALL) begin
            next_snake1_pos = snake1_pos + 1'b1;
        end
        // else, check if right side is walkable
        else if (snake1_pos - 1 > snake2_pos) begin
            next_snake1_pos = snake1_pos - 1'b1;
            next_snake1_dir = RIGHT;
        end
        // else, both left & right sides are not walkable, so stay still, and remain direction
    end
    RIGHT: begin
        // check if right side is walkable
        if (snake1_pos - 1 > snake2_pos) begin
            next_snake1_pos = snake1_pos - 1'b1;
        end
        // else, check if left side is walkable
        else if (snake1_pos < LEFT_WALL) begin
            next_snake1_pos = snake1_pos + 1'b1;
            next_snake1_dir = LEFT;
        end
        // else, both right & left sides are not walkable, so stay still, and remain direction
    end
endcase
```

Fig. 13: lab3_3 snake1 Combinational circuit code

Fig.14 為 led 的 combinational logic code。由於我的 snake1_pos、snake2_pos、snake3_pos 是存各個 snake 的最左邊身體位置，因此舉例來說 snake2 對到 led 的配置就是 binary number 11 往左移 snake2_pos – 1。我們再把三個 snake 對到的 led 配置 OR 起來就是整體的 led 亮法了。

```
always @(*) begin
    led = (16'b1 << snake1_pos) | (16'b11 << (snake2_pos - 1)) | (16'b111 << (snake3_pos - 2));
end
```

Fig. 14: lab3_3 led combinational logic code

B. Questions and Discussions

(a) 實作如 Fig.15 所示，首先我們要確保 rst 訊號一拉起來變為 1 就要 reset 了，因此就算不是 clock edge 我們也要 reset，所以要在 sensitivity list 裡加上 posedge rst。此外，把 rst 放在第一個 if statement 確認，這樣就會優先確認需不需要 reset，一旦需要就會進到 if (rst == 1’b1)的 block 裡做 reset，而不會執行其他動作。

```
always @(posedge local_clk or posedge rst) begin
    if (rst == 1'b1) begin
        led <= S0;
    end
    else if (en == 1'b1) begin
        led <= next_led;
    end
end
```

Fig. 15: highest priority rst

(b) 我們原本的 code 的問題在於當兩條蛇碰到時，舉 snake1 跟 snake2 碰到為例，只有先反彈的那條蛇(e.g., snake1)有更新到 direction，比較慢走的那條蛇(e.g., snake2)等到牠的 clk edge 來時，送到 DFF 的 next_snake2_dir 已經是 snake1 走掉後才判斷的方向，因此會維持原方向。

這個問題的來源是 snake2_dir 的更新速度不夠快，我們應該要在碰到 snake1 後就改變 snake2_dir。因此我們可以把 snake1_dir、snake2_dir、snake3_dir 從原本各自的 always block 裡拿出來，改成用最快的 clk 訊號驅動，如 Fig.16 所示。這樣一來一旦兩條蛇碰到，在各自 clock divider 的 pododge 來之前就已經更新為撞到後的新方向，也因此走的位置也已經是由新的方向去 combinational circuit 裡判斷。

```
always @(posedge clk or posedge rst) begin
    if(rst == 1'b1) begin
        snake1_dir <= RIGHT;
        snake2_dir <= RIGHT;
        snake3_dir <= LEFT;
    end else if (en == 1'b1) begin
        snake1_dir <= next_snake1_dir;
        snake2_dir <= next_snake2_dir;
        snake3_dir <= next_snake3_dir;
    end
end
```

Fig. 16: question B code

C. Problem Encountered

在寫 lab3_3 時我原本的 led 是用 sequential 的寫法，同樣用一個 DFF 去存、並在 combinational circuit 裡面更新各個 snake 的 next state 時順便一起判斷 led 的 next state。但是這樣 led 的 state update 的 always block 的 sensitivity list 就會需要偵測多個不同速度的 clock 的 posedge，因此會有 multiple driver 的問題導致沒辦法合成，如 Fig.17。我一開始的解法是改成用最快的 clock，也就是 clk，去更新 led，但這樣有可能更新到不對的 next_led 值，因為 clk 的 posedge 不一定是三個 snake 的 clk divider 的 posedge。

後來我想到其實 led 當下的亮法只跟各個 snake 的當下位置有關時，我才發現原來只要用一個 combinational circuit，以 snake1_pos、snake2_pos、snake3_pos 作為 input 去判斷 led 的值就好，才修改成如 Fig.14 的寫法。

```
always @(posedge clk_div1 or posedge clk_div2 or posedge clk_div3 or posedge rst) begin
    if(rst == 1'b1) begin
        led <= LED_INIT;
    end else if (en == 1'b1) begin
        led <= next_led;
    end
end
```

Fig. 17: Multiple Driver error

D. **Suggestions**

希望能提供每次 lab 的解答，不然有點難確認自己的 coding style 好不好，或者會不會寫的太冗長，其實可以有更精簡的寫法等。