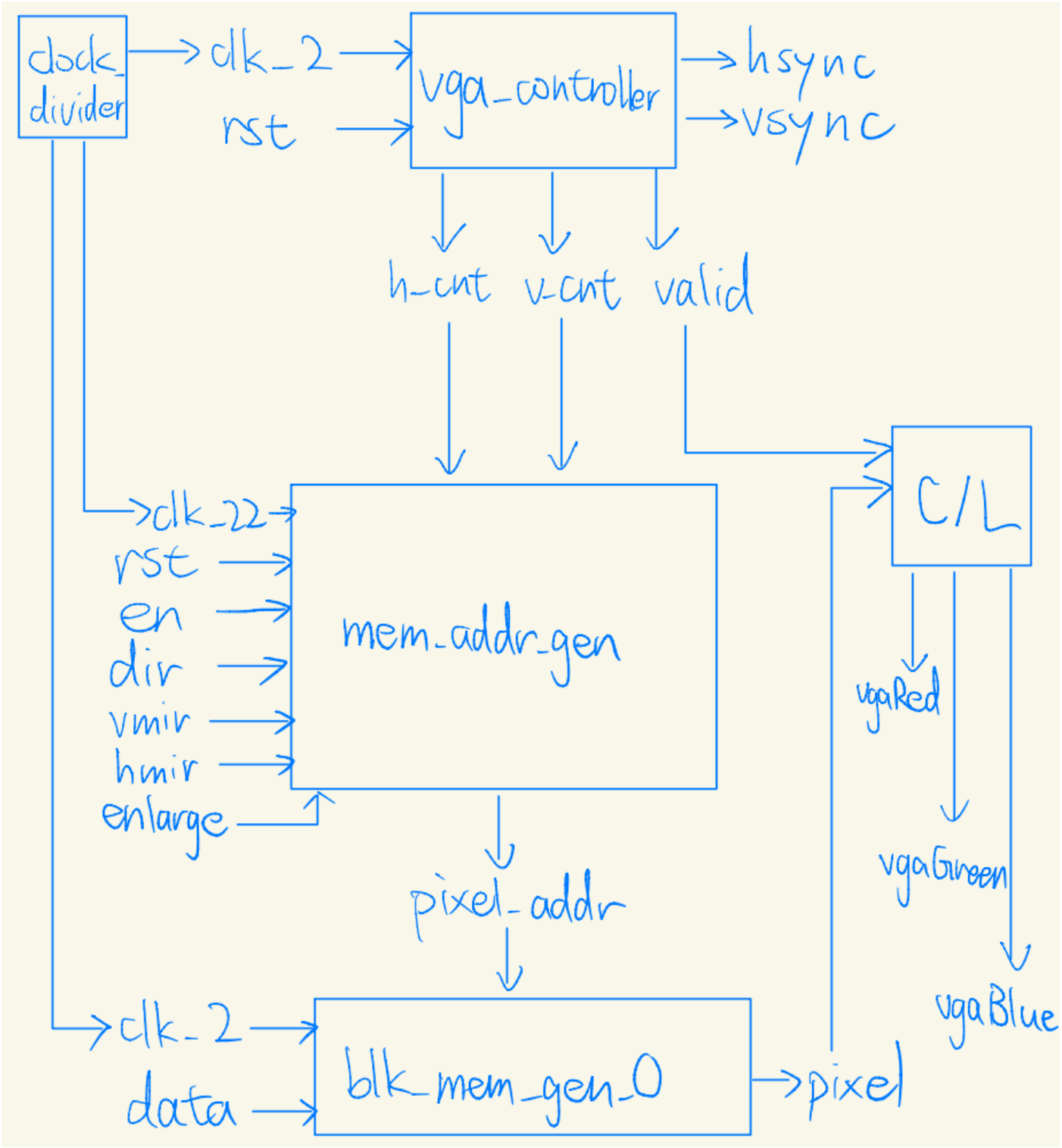
# Lab 6

學號: 109021115 姓名: 吳嘉濬

## A. Lab Implementation

以下是 lab6_1 的 block diagram：




我們主要實作的部分為 mem_addr_gen 的 block，基本運作原理是依照需求的功能在給定 h_cnt, v_cnt 的情況下決定你下個 cycle 的 pixel_addr，再利用 blk_mem_gen_0 的 frame buffer，去找到

對應 address 的 data，即 pixel 顏色的資料，進而顯現在螢幕上。

以下為 kernel code：

```
always@(*) begin
    next_position=position;
    if(en) begin
        if(dir) begin
            if(position==0) begin
                next_position=10'd319;
            end else begin
                next_position=position-1'b1;
            end
        end else begin
            if(position==319) begin
                next_position=10'd0;
            end else begin
                next_position=position+1'b1;
            end
        end
    end
end
```

每經過 clk_22 的 clock cycle，根據 en 和 dir 值，position 會有對應的增減，達到在螢幕上雙向水平移動的效果。

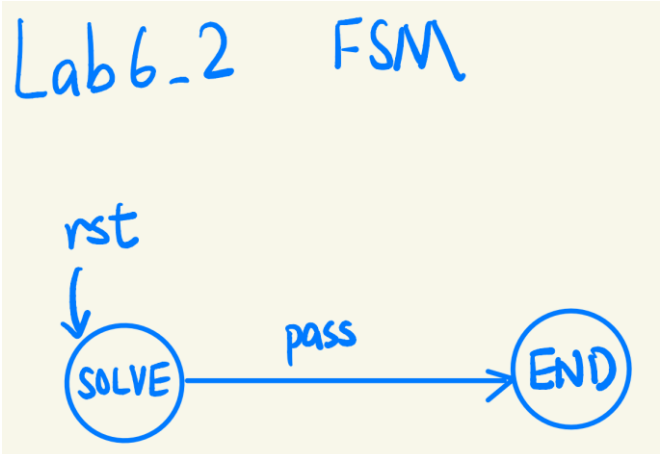
因為 lab6_1 形似 demo2，所以我的 coding style 仿造 demo2，直接利用 assign 去決定下個 cycle 的 pixel_addr：

```
assign pixel_addr = (hmir==1'b1) ? ((enlarge==1'b1) ? (((h_c>>2)+320*(v_c>>2)+(320*60+80)+(319-position))% 76800) : (((h_c>>1)+320*(v_c>>1)+(319-position))% 76800)) :
                    ((enlarge==1'b1) ? (((h_c>>2)+320*(v_c>>2)+(320*60+80)+position)% 76800) : (((h_c>>1)+320*(v_c>>1)+position)% 76800)) ;
                    //640*480 --> 320*240
assign h_c = (hmir==1'b1) ? (639-h_cnt) : h_cnt ;  //640-1
assign v_c = (vmir==1'b1) ? (479-v_cnt) : v_cnt ;  //480-1
```

先看第二個和第三個 assign statement，這是處理垂直和水平鏡像時做的變化。

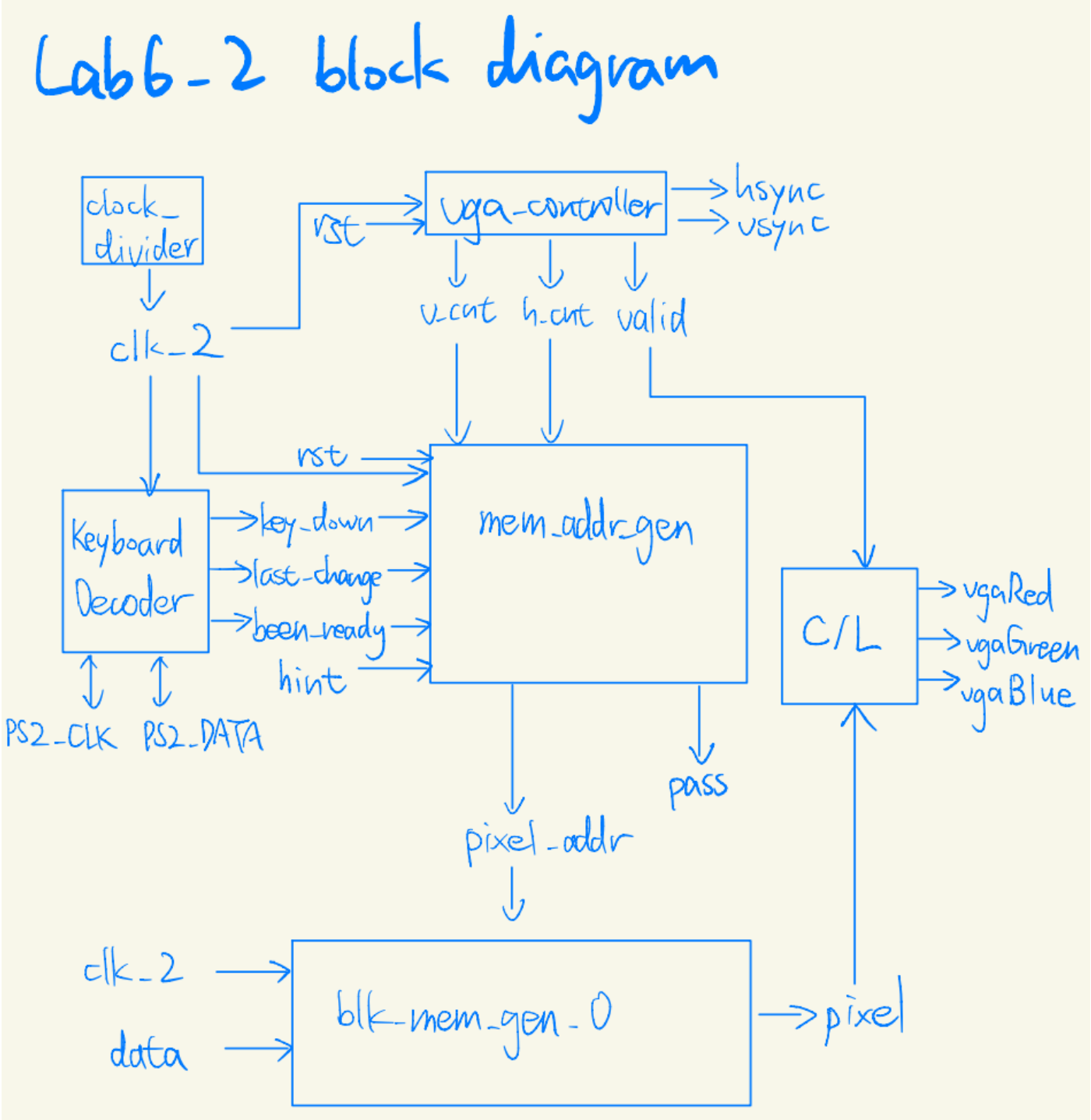
第一個 assign 考慮是否水平鏡像和是否要放大，所以總共有 4 種可能的計算。如果要水平鏡像，offset 不能是單純的+position，而是要+(319-position)(即 offset 從左到右變成從右到左)。enlarge 的部分，我希望圖案呈現是放大兩倍，而且要放大正中間，所以 offset 要另外加上(320*60+80)，如此畫面的左上角才會呈現往正中間放大兩倍後的圖案的左上方。另外，我讓 h_c<<1, v_c<<1 變成 h_c<<2, v_c<<2，如此，每連續 2*2 個 pixel 都會對應顯示放大前 1*1 個 pixel 的畫面，以達到放大 2 倍的效果。

以下為 lab6_2 的 FSM：



按下 rst 時，進入 SOLVE state，puzzle 會回到原本初始被打亂(非隨機)的狀態，當每個區塊對應初始圖片的位置正確且上下皆沒有顛倒時，pass 會被 assign 成 1'b1，進入 END state，從此不能再交換圖塊或是把某圖塊上下顛倒，直到再次被按下 rst。
自己實際上在實作時其實沒有分開成 2 個 state 設計，但都有確保機制的正常運作。

以下為 lab6_2 的 block diagram：




不同於 lab6_1，這次加上了鍵盤的控制，但基本上原理和 lab6_1 不會差太多，主要問題在於如何實作兩個區塊互換，以及讓某區塊旋轉 180 度，還有如何判定 puzzle 已經解開了。
首先，先說明基本的原理，我讓螢幕依照位置切成 4*4　16 塊，從左到右、從上到下，編號 0 到 15。以下為對應 code segment：

```
reg [4:0] state[15:0], next_state[15:0];
```

每一區塊存 5 個 bit 的資料，其中最左邊的 bit, i.e. state[x][4]，儲存此圖片是否 upside down，1'b0 表示沒有，1'b1 表示圖片是旋轉 180 度的，剩下 4 個 bit, i.e. state[x][3:0]，表示該區塊正在顯示「未被打亂圖片時，第"state[x][3:0]"個位置」對應的區塊。
因此我們可以判斷出何時 pass 可以被 set 成 1'b1：

```
assign pass = {state[0], state[1], state[2], state[3], state[4], state[5], state[6], state[7], state[8], state[9],
        state[10], state[11], state[12], state[13], state[14], state[15]}
        == 80'b00000_00001_00010_00011_00100_00101_00110_00111_01000_01001_01010_01011_01100_01101_01110_01111;
```

每個區塊正在顯示未被打亂前對應位置的區塊的畫面，並且皆沒有被 upside down。

以下為交換兩區塊以及旋轉某區塊 180 度的 kernel code：

```
always@(*) begin
    for(i=0;i<16;i=i+1) begin
        next_state[i]=state[i];
    end
    for(i=0;i<16;i=i+1) begin
        //next_state[i]=state[i];  //!!
        if(!pass && !hint && been_ready && key_down[last_change] && shift_down) begin
            if(last_change==KEY_CODES[i]) begin
                next_state[i]= {~state[i][4],state[i][3:0]};  //!!
            end
        end else begin
            for(j=0;j<16;j=j+1) begin
                //next_state[j]=state[j];
                if(i!=j) begin
                    if(!pass && !hint && been_ready && last_change==KEY_CODES[i] && key_down[KEY_CODES[i]] && key_down[KEY_CODES[j]]) begin
                        next_state[i]=state[j];
                        next_state[j]=state[i];
                    end
                end
            end
        end
    end
end
```

判斷出第 i 個區塊對應的 key 和 shift 鍵被同時按下時，我們讓 next_state[i]={~state[i][4], state[i][3:0]}，讓最左邊的 bit 從 0 變成 1 或從 1 變成 0，改變旋轉狀態。
判斷出第 i 個區塊對應的 key 和第 j 個區塊對應的 key 被同時按下時，我們讓 next_state[i]=state[j], next_state[j]=state[i]，第 i 個區塊和第 j 個區塊的資訊交換，達到交換區塊的效果。

最後則是如何利用以上資訊去決定每個 cycle 的 pixel_addr，以下為對應 kernel code：

```
always @(*) begin
    next_pa = 17'd0;
    if(hint) begin
        next_pa = (h_cnt>>1)+320*(v_cnt>>1);
    end else begin
        ho=160*(state[pos][3:0]%4);  //state[pos][3:0] indicates the index of pic fraction you want to display
        vo=120*(state[pos][3:0]/4);
        if(state[pos][4]==1'b0) begin
            next_pa = ((ho+(h_cnt%160))>>1)+320*((vo+(v_cnt%120))>>1);  //normal
        end else if(state[pos][4]==1'b1) begin
            next_pa = ((ho+(10'd160-(h_cnt%160)))>>1)+320*((vo+(10'd120-(v_cnt%120)))>>1);  //upside down
        end
    end
end
```

其中 ho 和 vo 儲存 16 個區塊的水平方向 offset(horizontal offset)和垂直方向 offset(vertical offset)，利用 ho 和 vo 的值，可以幫助我們在不同區塊時去計算下個 cycle 的 pixel_addr(i.e. next_pa)。

## B. Questions and Discussions

A. If we want to turn the image with a special effect of the negative film (which means each pixel is complemented in color, like the example shown in below, the word 'POPCAT' is original white, but

after complemented, it is shown in black), how would you modify your design of lab6_1?
把這一行

```
assign {vgaRed, vgaGreen, vgaBlue} = (valid==1'b1) ? pixel : 12'h0;
```

改成這一行

```
assign {vgaRed, vgaGreen, vgaBlue} = (valid==1'b1) ? {4'hf-pixel[11:8],4'hf-pixel[7:4],4'hf-pixel[3:0]} : 12'h0;
```

即可達到 negative film 的效果。
因為 pixel 記錄著在某個 cycle 時某一格 pixel 要顯示的顏色，其中每 4 個 bit 一組總共三組，分別代表著 R(red), G(green), B(blue)的 intensity。我們把 RGB 每個元素各自的 intensity 變成原本的互補，也又是從 4'hx 變成 4'hf-4'hx (i.e., 15-x)，即可達到負片的效果。(其中 x 是介於 0 到 f(15)之間的整數)

B. Suppose we want to create a VGA game with animations, how can you design the frame buffer and let the item on the monitor moves? (hint: you can use two frame buffers)
使用兩個 frame buffers，我們可以在其中一個 frame buffer(called A)在 display 到螢幕上時，讓另一個 frame buffer(called B)可以同時 update 下一個將要 display 的畫面，當進入下一幀時，A, B 兩者角色互換，換成 frame buffer B display 畫面，frame buffer A 去 update 下一個畫面，一直這樣反覆下去，如此能達到比較流暢逼真的連續畫面效果。

C. Our FPGA equips with the BRAM of only 1800 Kbits, which a 640×480 image cannot fit in. If we want to implement a video game, apart from storing a smaller image (e.g., 320×240) like we did in this lab, please give at least 2 possible methods to reduce the BRAM usage.
其中一個方法是 image tiling，將原始圖片切分成許多小區塊，稱為 tile，並把圖片中出現的小區塊組合儲存成一個 set，如此一來，顏色一樣但位置不同的色塊不會被重複儲存以達到節省 BRAM 使用的目的。
另一個方法是使用 RLE(run-length encoding)，把連續同個顏色的 pixel 以顏色及長度資訊來儲存，即可達到節省 BRAM 使用的目的。例如，有連續 100 個白色 pixel 時，我們與其使用 1200 個 bit 儲存{12'h0, 12'h0, …, 12'h0}，不如直接儲存{12'h0, 12'd100}(假設使用 12 bit 去儲存連續相同顏色 pixel 的長度)。

## C. Problem Encountered

```
always@(*) begin
    /*for(i=0;i<16;i=i+1) begin
        next_state[i]=state[i];
    end*/
    for(i=0;i<16;i=i+1) begin
        next_state[i]=state[i];  //!!
        if(!pass && !hint && been_ready && key_down[last_change] && shift_down) begin
            if(last_change==KEY_CODES[i]) begin
                next_state[i]= {~state[i][4],state[i][3:0]};  //!!
            end
        end else begin
            for(j=0;j<16;j=j+1) begin
                //next_state[j]=state[j];
                if(i!=j) begin
                    if(!pass && !hint && been_ready && last_change==KEY_CODES[i] && key_down[KEY_CODES[i]] && key_down[KEY_CODES[j]]) begin
                        next_state[i]=state[j];
                        next_state[j]=state[i];
                    end
                end
            end
        end
    end
end
```

在實作兩個色塊位置對調時，我一開始是使用以上的 code，其中 next_state[i]=state[i]; 這一行是為了給每一個色塊 default 的 assignment，但是實際在跑時，我發現欲交換的色塊假設為 A, B，理當在交換後會變成 B, A，但事實上卻是變成 A, A 或 B, B，於是我重新 trace 這一段 code，發現問題就是出現在 next_state[i]=state[i];這一行出現的位置不對，我應該把它擺在獨立的迴圈

內去執行，即上面被 comment 掉的第 2 到 4 行，否則如果在第 6 行執行，在第 5 行的迴圈當中，假設在 iteration i=x, j=y 時，發生了互換，next_state[x]=state[y]且 next_state[y]=state[x]，但因為沒有 break 掉迴圈，所以下一個 iteration 會繼續被執行，此時下一個 iteration 的 next_state[i]=state[i]這一行可能會讓剛剛交換完 state 的色塊又被 assign 成 default 的情況，導致可能其中一個色塊在交換完 state 後又被 default 回原本的 state，造成出現 A, A 或 B, B 的狀況。

## D. Suggestions

希望老師在上課時可以更頻繁講笑話，或許對出席率會有幫助。