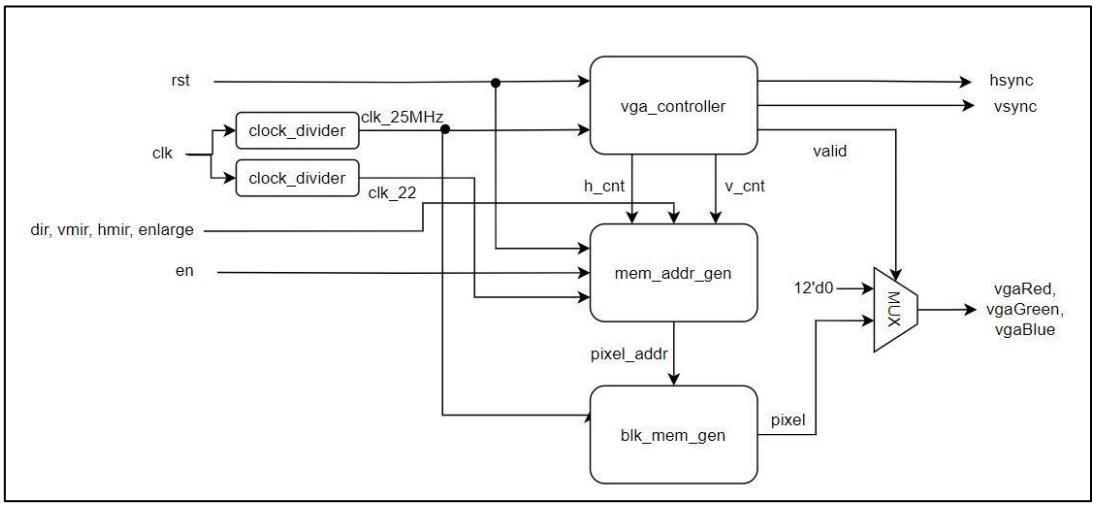
Lab 6

學號: 110011138 姓名: 楊立慈

A. Lab Implementation

(a) Lab6_1

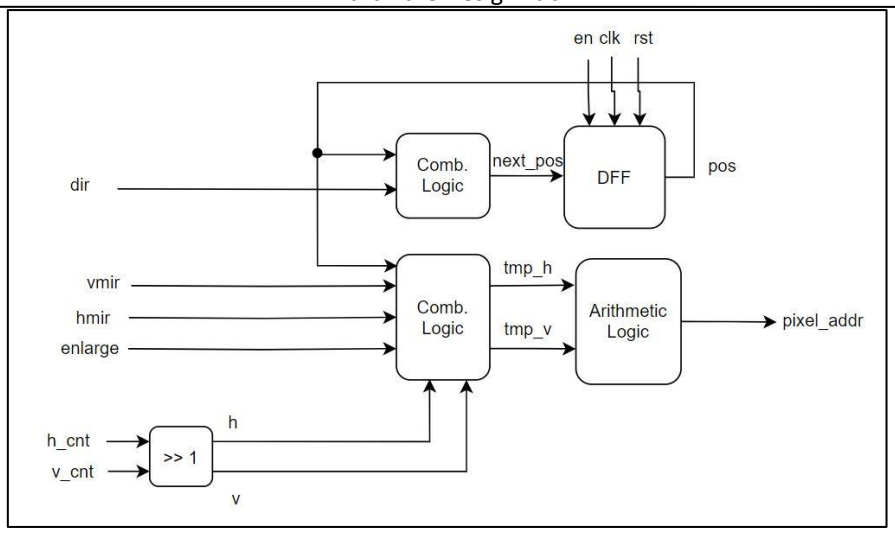
下圖為 lab6_1 module 的 block diagram，基本上與講義上 demo2 的 block diagram 很類似。首先，我們利用 vga_controller 來產生 vga display 要用的 hsync、vsync 訊號外，也產生 h_cnt、v_cnt 訊號，我們接著利用 h_cnt、v_cnt、dir、vmir、hmir、enlarge 訊號來在 mem_addr_gen module 裡計算目前 h_cnt、v_cnt 所對應到圖片的 pixel address。最後再利用 blk_mem_gen module，根據 pixel_addr 來計算目前 pixel 的 RGB 值，並利用 MUX 以 valid==1'b1 作為 selection line 來決定 vga RGB 的值是 pixel 或是 0。



其中 MUX 的實作 code 如下，主要是利用 concatenation 將 12-bit 的 pixel assign 給 output 的 vgaRed、vgaGreen、vgaBlue。

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

下圖為 mem_addr_gen module 的 block diagram。首先，由於我們用的圖片由 640x480 縮小為 320x240，因此 h_cnt、v_cnt 要先除以 2，也就是>>1 的運算。再來我們利用 DFF 存 pos，也就是目前圖片捲動的位置，而 next_pos 的 Comb. Logic 只要由 pos、dir 即可決定。再來我們利用 pos、vmir、hmir、enlarge、h、v 來計算 tmp_h、tmp_v。最後的 pixel_addr 就可以由 tmp_h、tmp_v 來計算。



下圖為 block diagram 中 pos 的 DFF 以及 Comb. Logic 實作。可看到 pos 在 reset 後會回到 0、並帶有 enable 功能來決定圖片是否要捲動。Next_pos 的 Comb. Logic 則是依 direction 是向左或向右有不同的數法，如果是向左則 next_pos 是 pos+1 (超過極限 H_LEN 則回到 0 重數)，如果向右則是 pos-1 (同樣超過極限 0 則回到 H_LEN – 1 開始數)。

```
always @ (posedge clk or posedge rst) begin
    if (rst == 1'b1) pos <= 0;
    else if (en == 1'b1) pos <= next_pos;
end

always @(*) begin
    next_pos = pos;
    if (dir == TO_LEFT) next_pos = (pos == H_LEN - 1) ? 0 : pos + 1;
    else next_pos = (pos == 0) ? H_LEN - 1 : pos - 1;
end
```

下圖是計算 tmp_h、tmp_v arrays 的 Comb. Logic 實作。其中 tmp_h[0]、tmp_v[0]代表考慮是否 enlarge 以及經過 pos 位移後的(h, v)座標，tmp_h[1]、tmp_v[1]代表考慮是否要 vertical mirror 後的(h, v)座標，tmp_h[2]、tmp_v[2]代表考慮是否要 horizontal mirror 後的(h, v)座標，且每一次計算都是由先前的(h, v)座標計算，因此結果的 tmp_h[2]、tmp_v[2]就是考慮所有因素後的最終座標。首先，由於我訂的 enlarge 是增加 2 倍，因此如果要 enlarge 則 tmp_h[0]會是 h 除以 2 之後再位移 pos 單位再加上 79 (i.e., H_LEN / 4 – 1, 因為我們增加 2 倍後顯示的圖片會變成原本捲動的圖片的中心的 160x120 的範圍，因此距離原本 320x240 的框框左邊會有 79 個單位的差異，所以這邊要加上此差異)；如果不用 enlarge，則單純 h 加上 pos 位移。最後要記得 %H_LEN 來確保再 H_LEN 範圍內，而 tmp_v[0]也是類似概念，不過不用 pos 的位移。至於 tmp_v[1]則是如果要 vmir 的話要 update 成 tmp_v[0]跟圖片下方的邊界，i.e.，V_LEN – 1，的差。Tmp_h[2]也是類似的判斷概念。

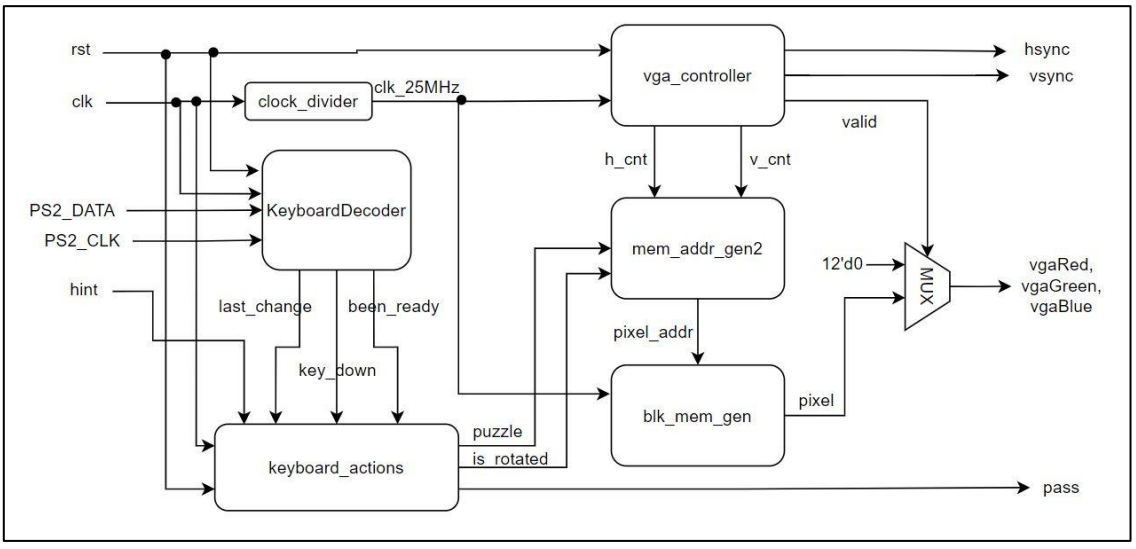
```
always @(*) begin
    // check enlarge (2 times)
    tmp_h[0] = (enlarge ? (h >> 1) + pos + 79 : h + pos) % H_LEN;
    tmp_v[0] = enlarge ? (v >> 1) + 59 : v;
    // check vertical mirror
    tmp_h[1] = tmp_h[0];
    tmp_v[1] = (vmir) ? (V_LEN - 1 - tmp_v[0]) : tmp_v[0];
    // check horizontal mirror
    tmp_h[2] = (hmir) ? (H_LEN - 1 - tmp_h[1]) : tmp_h[1];
    tmp_v[2] = tmp_v[1];
end
```

下圖為>>1 以及計算 pixel_addr 的 Arithmetic Logic 實作。基本上就是利用先前計算的 tmp_h[2]、tmp_v[2]，將 2D 的(h, v)座標轉為 1D 的座標，並記得要%SIZE 以免超過範圍。

```
assign h = h_cnt >> 1;
assign v = v_cnt >> 1;
assign pixel_addr = (tmp_h[2] + H_LEN * tmp_v[2]) % SIZE;
```

(b) Lab6_2

下圖為 lab6_2 module 的 block diagram。大致與 lab6_1 類似，不過多了 KeyboardDecoder 來處理 keyboard 的輸入，並將 last_change、been_ready、key_down 等訊號輸出傳到 keyboard_actions module，再加上 hint 訊號來計算出 puzzle、is_rotated、pass 訊號。其中 puzzle 為 size 16 的 array，第 i 個 element 存的是顯示出來的拼圖的第 i 個格子對應到原本圖片的格子號碼(我對 16 個格子的編號方法為由左到右、由上到下，從 0 編到 15)。而 is_rotated 也是 size 16 的 array，第 i 個 element 存的是拼圖的第 i 個格子是否要轉 180 度。利用 puzzle、is_rotated 資訊再加上 h_cnt、v_cnt 等就可以在 mem_addr_gen2 module 計算 pixel_addr。



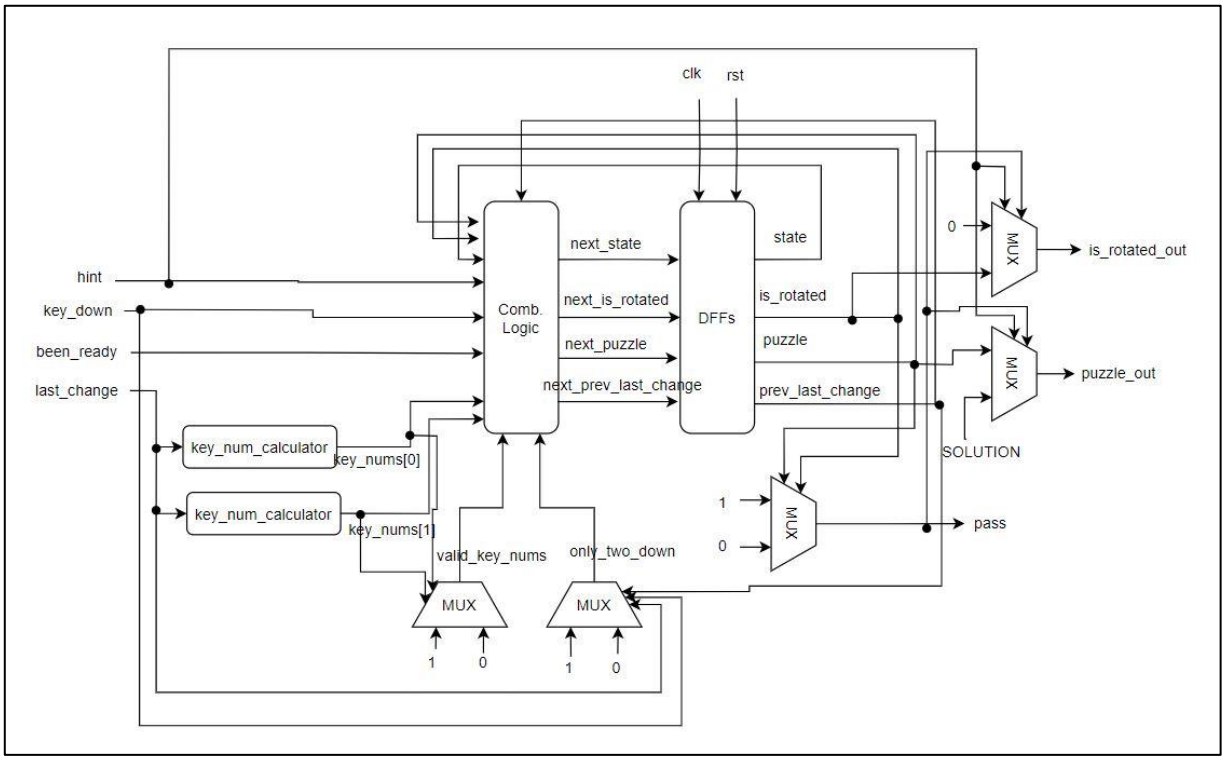
下圖為 puzzle、is_rotated 等訊號的 decalration 以及 MUX 的實作(與 lab6_1 一樣)。值得注意的是 puzzle、is_rotated 在 keyboard_actions、mem_addr_gen2 modules 裡都是 declare 成

size 16 的 array，然而由於 verilog 不能傳 array，因此我傳的是 flattened 過後的 2D array。此外，由於 is_rotated 每個 element 只要一個 bit 就夠了，因此我後來就把它當作 16-bit 的 variable 傳即可。

```
wire [0:63] puzzle; // 16 4-bit
wire [0:15] is_rotated;

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

下圖為 keyboard_actions module 的 block diagram。可看到我們用 4 個 DFF 來儲存 state、is_rotated、puzzle、prev_last_change 等資訊。在左下角，我們利用 key_num_calculator modules 來將 last_change、prev_last_change 轉成對應的格子號碼，儲存於 key_nums。再利用 key_nums[0]、key_nums[1]通過 MUX 計算 valid_key_nums 訊號，代表算出來的 key_num 是否是合法的格子號碼。以及利用 key_down、last_change、prev_last_change 訊號通過 MUX 計算 only_two_down 訊號，代表 key_down 裡是否只有 last_change 和 prev_last_change 兩個按鍵按下。在右方則利用 is_rotated、puzzle 訊號通過 MUX 計算 pass 輸出訊號，以及用 hint、pass 訊號通過 MUX 計算 is_rotated_out 和 puzzle_out 輸出訊號。



下圖為計算 pass 輸出訊號的 MUX 實作。基本上當拼圖某格的格子號碼跟對應到原本圖片的格子號碼不同或者某格有被轉 180 度，也就是說(puzzle[0] != 0 || is_rotated[0]) || (puzzle[1] != 0 || is_rotated[1]) || … || (puzzle[15] != 0 || is_rotated[15])，pass 就會是 0，代表拼圖還沒拚好，否則 pass 就是 1。這裡運用了 for-loop 來讓 code 簡潔一些。

```
always @(*) begin
    pass = 1'b1;
    for (pass_i = 0; pass_i < 16; pass_i = pass_i + 1'b1) begin
        if (puzzle[pass_i] != pass_i || is_rotated[pass_i]) begin
            pass = 1'b0;
        end
    end
end
```

下圖為 is_rotated_out、puzzle_out 的 MUX 的實作。利用 hint | pass 作為 selection line，如果要 hint 或者拼圖已經拚完了，則 is_rotated_out 會是 0、puzzle_out 則利用 concatenation assign SOLUTION 的值(要用 concatenation 是因為 SOLUTION 是 2D array 但 puzzle_out 是 flattened 過後的 1D array)；否則分別 assign 目前 is_rotated、puzzle 的值。

```
always @(*) begin
    if (hint | pass) begin
        is_rotated_out = 0;
        puzzle_out = {
            SOLUTION[0], SOLUTION[1], SOLUTION[2], SOLUTION[3],
            SOLUTION[4], SOLUTION[5], SOLUTION[6], SOLUTION[7],
            SOLUTION[8], SOLUTION[9], SOLUTION[10], SOLUTION[11],
            SOLUTION[12], SOLUTION[13], SOLUTION[14], SOLUTION[15]
        };
    end
    else begin
        is_rotated_out = is_rotated;
        puzzle_out = {
            puzzle[0], puzzle[1], puzzle[2], puzzle[3],
            puzzle[4], puzzle[5], puzzle[6], puzzle[7],
            puzzle[8], puzzle[9], puzzle[10], puzzle[11],
            puzzle[12], puzzle[13], puzzle[14], puzzle[15]
        };
    end
end
```

下圖為計算 valid_key_nums、only_two_down 的 MUX 實作。當 key_nums 其中一個是 UNKNOWN 時就代表 key_nums 不合法，因此 valid_key_nums 是 0，否則即為 1。而當 last_change 不等於 prev_last_change (i.e.，兩個按鍵不同)且 key_down 中只有 last_change 和 prev_last_change 按下時(判斷方式如圖所示，即代表 key_down 中只有第 last_change 個 bit 和第 prev_last_change 個 bit 是 1)，則代表只有 2 個按鍵按下。

```
assign valid_key_nums = (key_nums[0] != UNKNOWN && key_nums[1] != UNKNOWN) ? 1'b1 : 1'b0;
assign only_two_down = (last_change != prev_last_change
                    && key_down == ((512'b1 << last_change) | (512'b1 << prev_last_change)))
                    ? 1'b1 : 1'b0;
```

下兩張圖是最主要計算 next_state、next_is_rotated、next_puzzle、next_prev_last_change 的 Comb. Logic 的實作。首先我們主要有 2 個 state: WAIT_KEYS_DOWN 和 WAIT_KEYS_RELEASE，下圖為 WAIT_KEYS_DOWN state 中的 Comb. Logic。當 been_ready 過且只有 2 個按鍵按下且 按下的按鍵都是合法的，則進入 WAIT_KEYS_RELEASE state，同時更新 next_is_rotated、next_puzzle。當不用 hint、拼圖還沒 pass 時，我們再依據 key_nums 的

值來判斷要更新甚麼，如果其中一個是 SHIFT，則就將另一個按鍵對應的格子號碼的 next_is_rotated 更新為原本的 complement；如果兩個按鍵都不是 SHIFT，代表要交換兩個格子，因此就將兩個格子號碼對應的 puzzle、is_rotated 值對調。

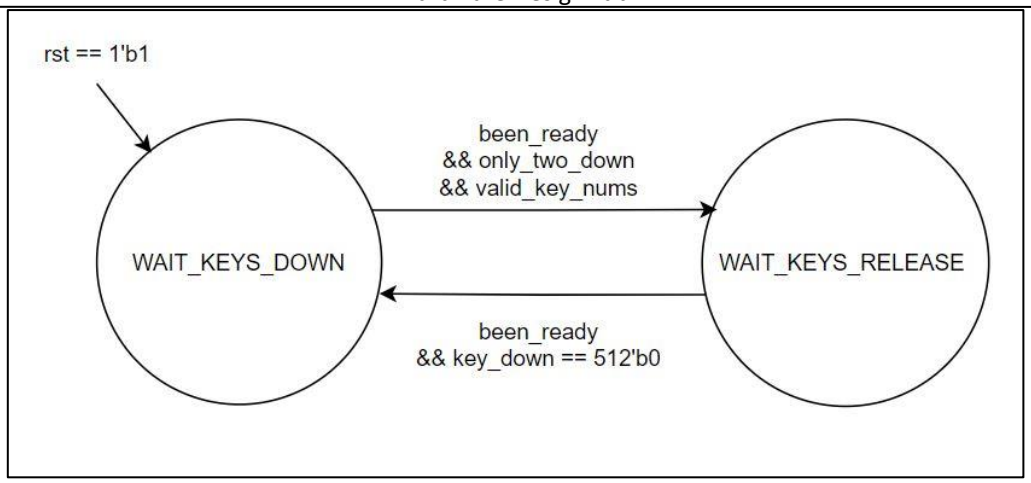
```
case (state)
    WAIT_KEYS_DOWN: begin
        if (been_ready && only_two_down && valid_key_nums) begin
            next_state = WAIT_KEYS_RELEASE;

            // process keyboard input if !hint and !pass
            if (hint != 1'b1 && pass != 1'b1) begin
                // rotate block key_nums[1]
                if (key_nums[0] == SHIFT_NUM) begin
                    next_is_rotated[key_nums[1]] = ~is_rotated[key_nums[1]];
                end
                // rotate block key_nums[0]
                else if (key_nums[1] == SHIFT_NUM) begin
                    next_is_rotated[key_nums[0]] = ~is_rotated[key_nums[0]];
                end
                // swap blocks
                else begin
                    next_puzzle[key_nums[0]] = puzzle[key_nums[1]];
                    next_is_rotated[key_nums[0]] = is_rotated[key_nums[1]];
                    next_puzzle[key_nums[1]] = puzzle[key_nums[0]];
                    next_is_rotated[key_nums[1]] = is_rotated[key_nums[0]];
                end
            end
        end
    end
```

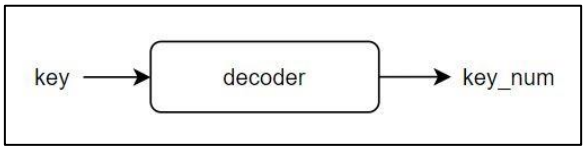
下圖為延續上述的 Comb. Logic，只是是在 WAIT_KEYS_RELEASE state。在這裡代表有 2 個按鍵已經按下，而且因為 spec 不希望這時候再按其他按鍵也會有反應，因此在這裡所有的 next_state、next_is_rotated、next_puzzle 值都不變，除了當 been_ready 過且所有按鍵都已放開(i.e.，key_down == 0)，則 next_state 就回到 WAIT_KEYS_DOWN。

```
    WAIT_KEYS_RELEASE: begin
        if (been_ready && key_down == 512'b0) begin
            next_state = WAIT_KEYS_DOWN;
        end
    end
endcase
```

下圖為此 module 的 state diagram。當 reset 後會回到 WAIT_KEYS_DOWN 等待按鍵按下。在 WAIT_KEYS_DOWN 時，如果只有 exactly 兩個按鍵按下(判斷方法如前所述)，則會進到 WAIT_KEYS_RELEASE。在 WAIT_KEYS_RELEASE 時，如果所有按鍵都放開了(判斷方法如前所述)，則回到 WAIT_KEYS_DOWN。



下圖為 key_num_calculator module 的 block diagram。可看到單純為將傳入的 key 轉為對應的格子號碼並輸出。

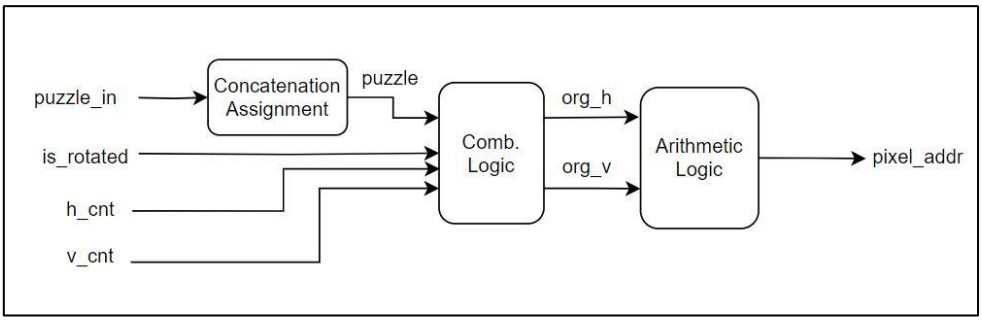


下圖為 decoder 的實作，可看到如果 key 有對應的 KEY_CODE 則設 key_num 為該對應號碼，若為 SHIFT 則對應到 SHIFT_NUM，若沒有對應的 KEY_CODE 則代表是非法的按鍵，因此 key_num 設為 UNKNOWN。

```
always @ (*) begin
    case (key)
        KEY_CODES[00] : key_num = 5'd0;
        KEY_CODES[01] : key_num = 5'd1;
        KEY_CODES[02] : key_num = 5'd2;
        KEY_CODES[03] : key_num = 5'd3;
        KEY_CODES[04] : key_num = 5'd4;
        KEY_CODES[05] : key_num = 5'd5;
        KEY_CODES[06] : key_num = 5'd6;
        KEY_CODES[07] : key_num = 5'd7;
        KEY_CODES[08] : key_num = 5'd8;
        KEY_CODES[09] : key_num = 5'd9;
        KEY_CODES[10] : key_num = 5'd10;
        KEY_CODES[11] : key_num = 5'd11;
        KEY_CODES[12] : key_num = 5'd12;
        KEY_CODES[13] : key_num = 5'd13;
        KEY_CODES[14] : key_num = 5'd14;
        KEY_CODES[15] : key_num = 5'd15;
        KEY_CODES[SHIFT_NUM] : key_num = SHIFT_NUM;
        default : key_num = UNKNOWN;
    endcase
end
```

下圖為 mem_addr_gen2 的 block diagram。首先，我們將輸入的 flattened array 轉為 2D 的

array puzzle 後，再加上 is_rotated、h_cnt、v_cnt 訊號經過 Comb. Logic 算出 org_h、org_v，這兩個訊號代表目前的 h_cnt、v_cnt 對應到原本圖片的(h, v)座標。接著用跟 lab6_1 一樣的方式算出 pixel_addr 輸出。



下圖為 Concatenation Assignment 的實作，基本上就是利用 concatenation 來將 1D 的 puzzle_in assign 給對應到 puzzle array 中的 4-bit 的 element。

```
always @(*) begin
    {puzzle[0], puzzle[1], puzzle[2], puzzle[3],
     puzzle[4], puzzle[5], puzzle[6], puzzle[7],
     puzzle[8], puzzle[9], puzzle[10], puzzle[11],
     puzzle[12], puzzle[13], puzzle[14], puzzle[15]} = {
        puzzle_in[0:3], puzzle_in[4:7], puzzle_in[8:11], puzzle_in[12:15],
        puzzle_in[16:19], puzzle_in[20:23], puzzle_in[24:27], puzzle_in[28:31],
        puzzle_in[32:35], puzzle_in[36:39], puzzle_in[40:43], puzzle_in[44:47],
        puzzle_in[48:51], puzzle_in[52:55], puzzle_in[56:59], puzzle_in[60:63]
    };
end
```

下圖為 Comb. Logic 的實作。其中 h、v 跟 lab6_1 一樣代表縮小到 320x240 圖片的(h, v)座標。blk_num 代表目前(h, v)對應到拼圖中的第幾格。org_blk_num 代表拼圖中的這一格對回原本圖片的第幾格，可以直接用 puzzle[blk_num]得到。rel_h、rel_v 代表目前(h, v)相對目前拼圖中所在格子的最左上角的相對位置，在判斷 rel_v 時會依照目前所在格子是否有旋轉(用 is_rotated[blk_num]判斷)來決定 rel_v 要不要做 vertical mirror (做法與 lab6_1 vmir 一樣，如果有旋轉 180 度則要進行 vertical mirror)。org_blk_h、org_blk_v 代表對應回原本圖片的格子的左上角的(h, v)座標(這裡用了一些 bit operation 來取代%、/運算)。org_h、org_v 代表最終我們要顯示的 pixel 在原本圖片的(h, v)座標，也因此用 org_blk 座標加上 rel 相對座標即可。

```
always @(*) begin
    h = h_cnt >> 1;
    v = v_cnt >> 1;
    blk_num = 4 * (v / V_BLOCK_SIZE) + (h / H_BLOCK_SIZE);
    org_blk_num = puzzle[blk_num];
    rel_h = h % H_BLOCK_SIZE;
    rel_v = is_rotated[blk_num] ? V_BLOCK_SIZE - v % V_BLOCK_SIZE - 1 : v % V_BLOCK_SIZE;
    org_blk_h = H_BLOCK_SIZE * (org_blk_num & 4'b0011); // org_blk_num % 4
    org_blk_v = V_BLOCK_SIZE * (org_blk_num >> 2); // org_blk_num / 4
    org_h = org_blk_h + rel_h;
    org_v = org_blk_v + rel_v;
end
```

下圖是 Arithmetic Logic 的實作。基本上跟 lab6_1 一樣的概念，只有變數名稱要將 tmp_h[2]、tmp_v[2]改為 org_h、org_v。

```
assign pixel_addr = (org_h + H_LEN * org_v) % SIZE;
```

B. Questions and Discussions

(a) 由於我們每個 pixel 的每個顏色是用 4 個 bit 表示，每個顏色的值會在 0 到 15 之間，因此要作出該顏色的對立色只要用 15 減掉該顏色的值即可，相當於對該顏色的每個 bit 做 complement。所以要將每個 pixel 的顏色都 complement 只需要在 lab6_1 中 assign 每個 pixel 三個顏色的值給 vgaRed、vgaGreen、vgaBlue 時，改成 assign ~pixel 即可，如下圖。

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

(b) 我們可以跟著某個 clk source，然後在每個 clk cycle 都 display 一個靜態的影像，並且讓每個 cycle 的影像中要動的物品位置相較上一個 cycle 中該物品的位置，只改變一點點。如此一來，當 clk 的頻率夠快時，由於視覺暫留的影響人眼看起來會好像物品的動作是連續的，形成動畫的效果。我們可以利用 2 個 frame buffer，當其中一個 frame buffer 被用來 display 當前 clk cycle 的畫面時，我們可以將下一個 cycle 要 display 的畫面先載在另一個 frame buffer 裡，這樣下一個 cycle 一來就只要交換 2 個 frame buffer 就好，先前載好畫面的 frame buffer 用來 display，先前拿來 display 的 buffer 變成用來載再下一個 cycle 的畫面。

(c) Method 1: 將 RGB 每個顏色使用的 bit 數減少，雖然這樣能顯示的顏色組合會更少，但確實可以減少圖片需要的 bit 數。舉例來說，如果將每個顏色的 bit 數由目前的 4-bit 降為 3-bit，1800kbits 的 BRAM 就可以容納下 516x387 的圖片 。

Method 2: 利用 image compression algorithm，並另外寫一個可以根據 compress 過後的資料來計算原先該 pixel 的 RGB 值的 module。舉例來說，一個 naïve 的壓縮方法可以是將圖片的所有 pixel 由左到右、由上到下的存在一個 1D array 裡，然後 array 中如果有相鄰的 pixel 是一樣的顏色，我們可以改成用該顏色和重複該顏色的 pixel 數量來記錄這群相鄰同色的 pixel。然後在 verilog code 裡面要讀取 pixel 的 RGB 值時就可以利用這些資料來回推它的顏色。

Method 3: 我們目前利用 12 個 bit 總共可以表現2^{12}個顏色，但有時候一張圖片不一定會用到這麼多顏色，或者有些顏色彼此的差異其實一般人眼分不出來。因此我們可以考慮用 hardcode 的方式建一個 array 來存所有該圖片可能用到的 RGB 顏色，然後用 array index 當作代表該顏色的值，比如說 index 0 代表 RGB = (5, 3, 8)、index 1 代表 RGB = (12, 15, 1)等等，接著在存圖片每個 pixel 的顏色時只要存該顏色的 array index 即可。舉例來說，如果一張圖片只有 100 個顏色，那每個 pixel 只需要 7 個 bit 來存該顏色的 array index 就夠了。

C. Problem Encountered

在實作 lab6_1 時有出現當圖片移動時到某個位置後會跳回圖片最左邊對到螢幕左側的位置，而不是順順的持續移動。一開始我以為是板子或是螢幕的問題，不過再跟同學借來測試後來是有同樣的問題，後來我利用 7-segment 將像是 pixel_addr、tmp_h[2]、tmp_v[2]、pos 等等值都顯

示出來後才發現問題出在由於我的 code 有些是直接複製 demo 的 code 過來的，因此 pos 只有 8 個 bit，也導致每次數到 255 之後，pos 就會回到 0 而不是繼續數到 319再回到 0。

雖然這個問題讓我 debug 了很久，但卻也讓我發現其實可以利用 LED、7-segment 等等顯示器來幫助 debug。尤其是到了後面這些 lab 很難用 simulation 看波型，我原本 debug 都只能一直重複確認 code 的邏輯直到找到問題。

D. Suggestions

由於建議之前都寫完了，所以附一個療癒狗狗影片 : youtube