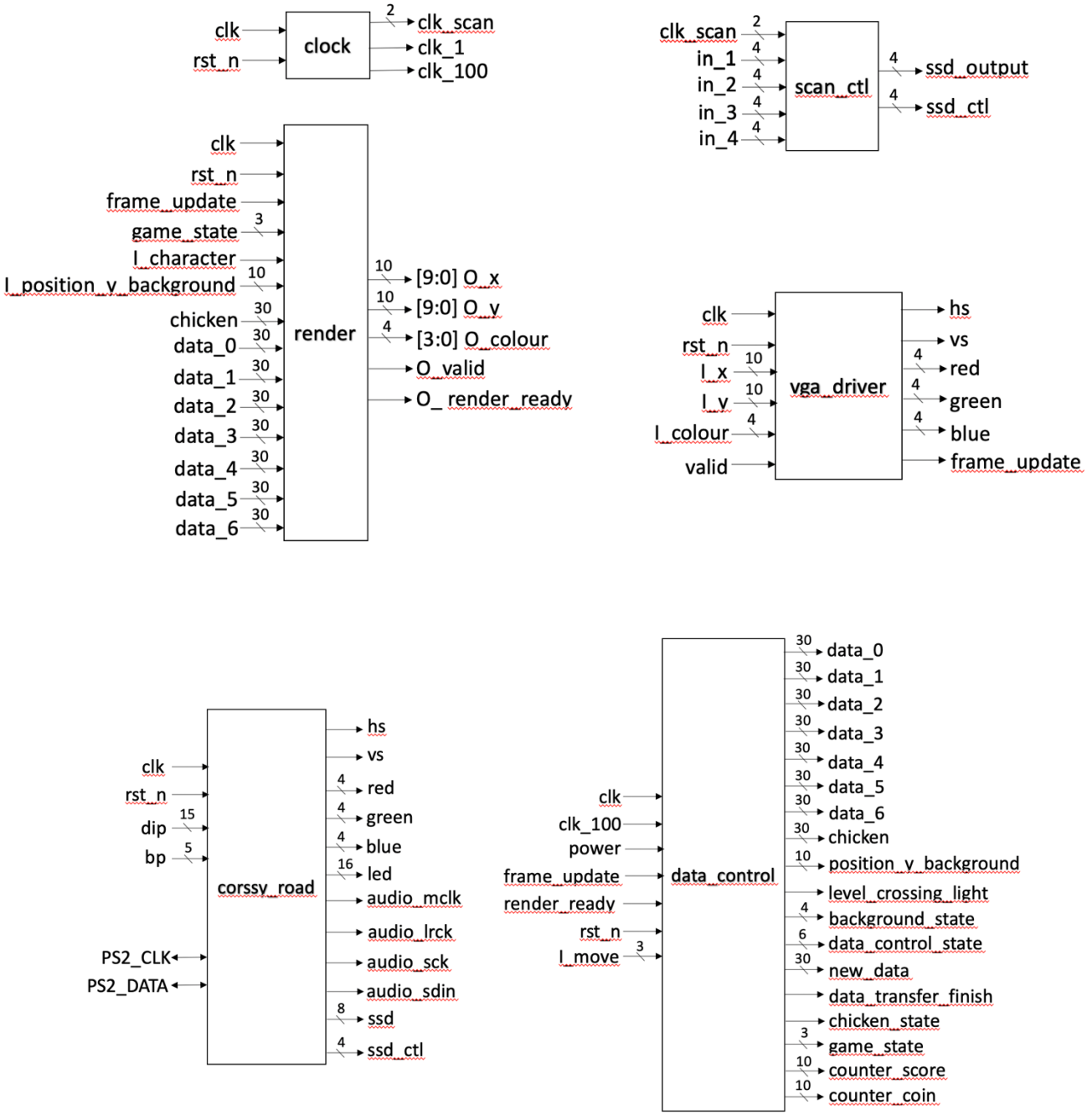
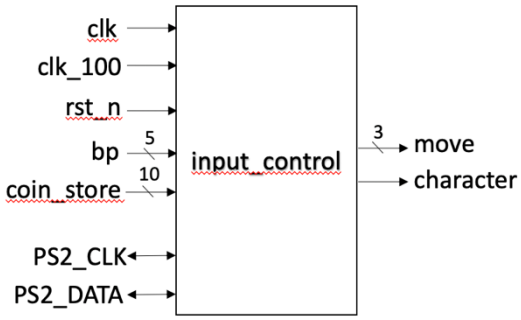
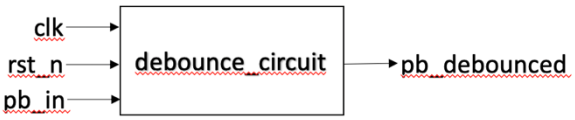
Final Project

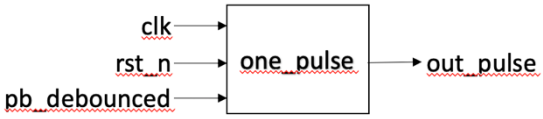
Design Specification

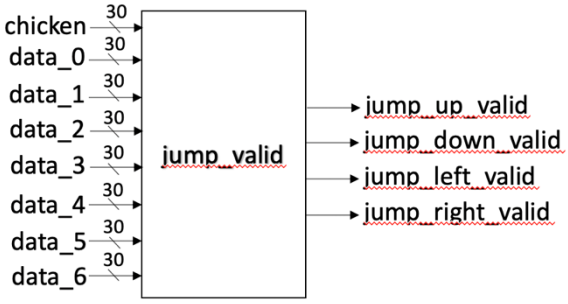
For a crossy:
Input: clk, rst_n, dip[14:0], bp[4;0].
Output: hs, vs, red[3:0], green[3:0], blue[3:0], led[15:0], audio_mclk, audio_sck, audio_lrck, audio_sdin, ssd[7:0], ssd_ctl[3:0].
Inout: PS2_CLK, PS2_DATA;

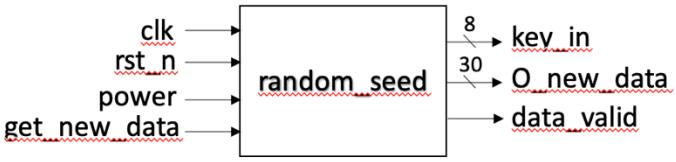


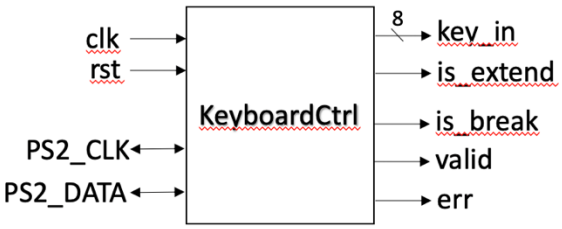


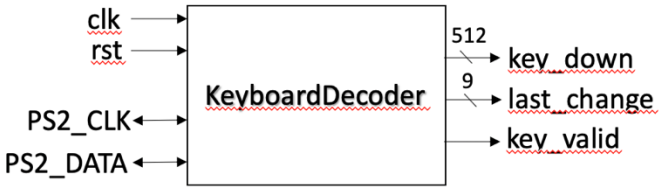


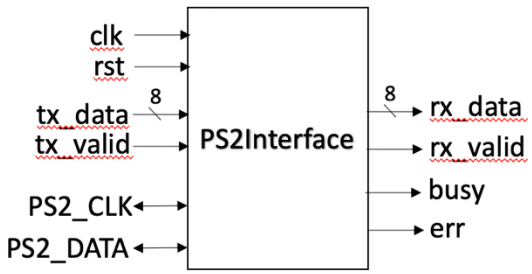


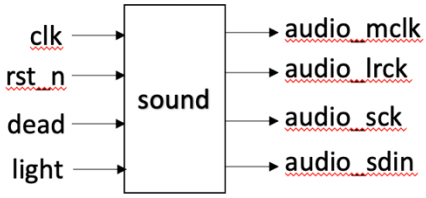


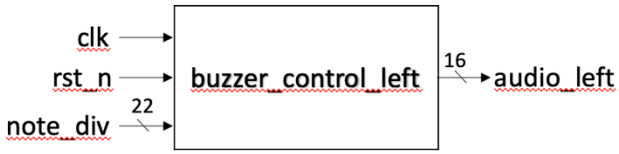


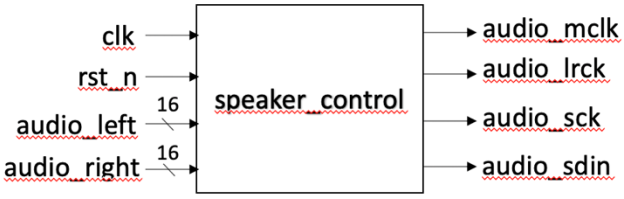


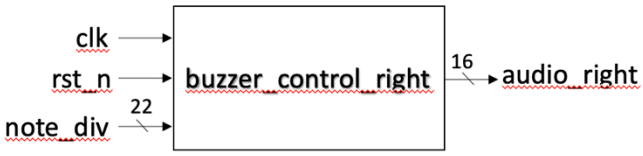


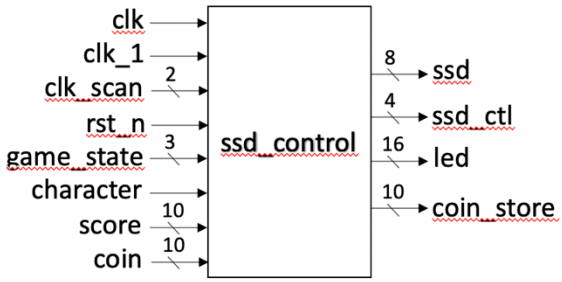


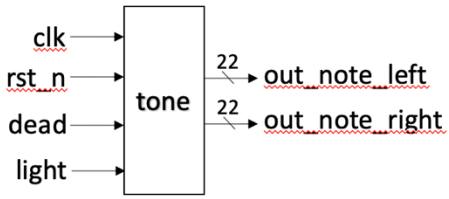




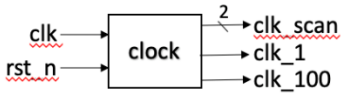


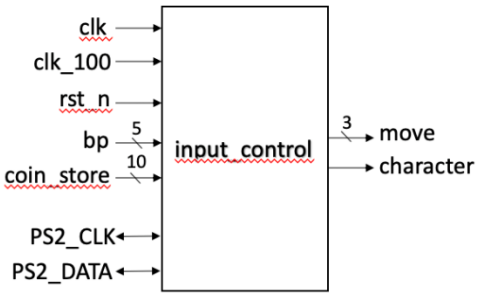






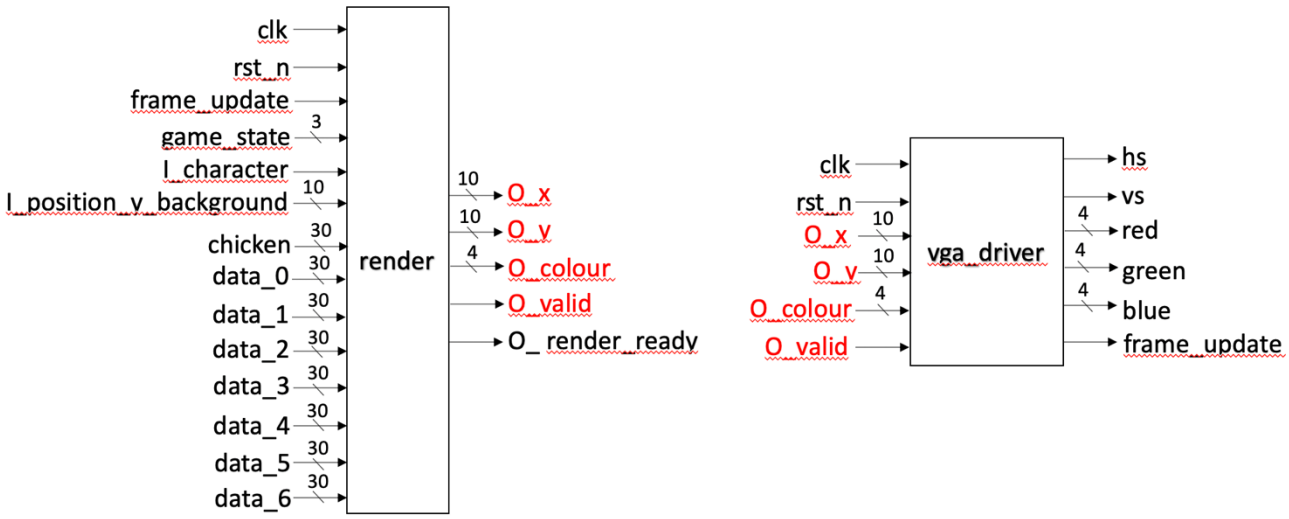
CONTROL PATH





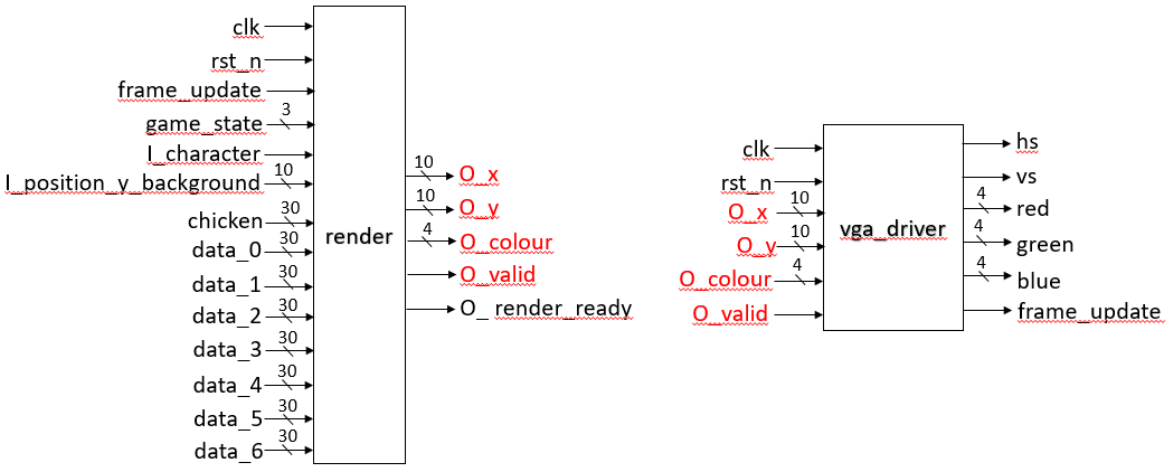
DATA PATH

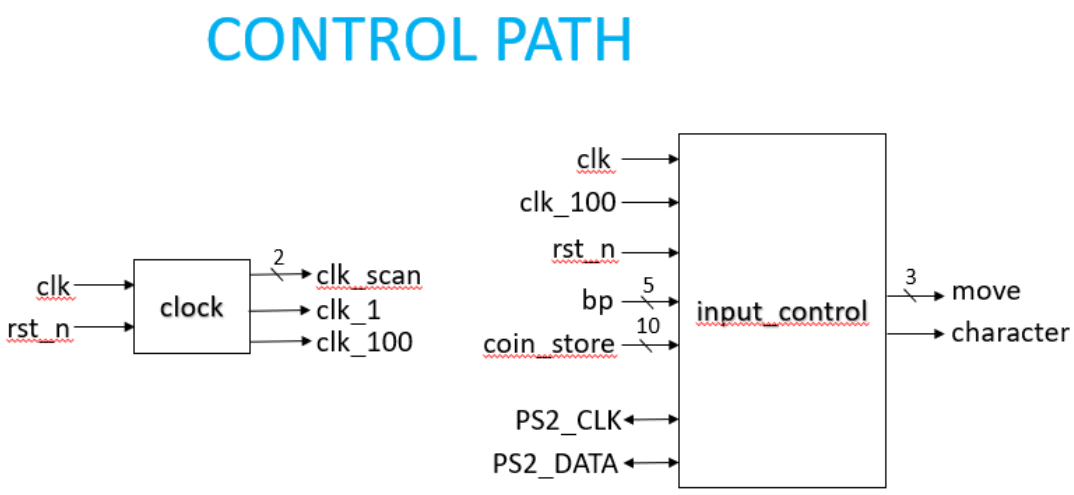
紅色為相連接線



DATA PATH

紅色為相連接線



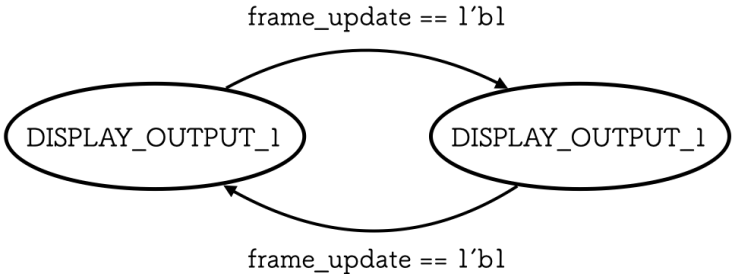


Design Implementation

第一、vga_driver

此 module 的功能為將輸入的訊號儲存並轉換成 vga 輸出訊號的格式輸出。在此 module 我將輸入的訊號格式定義為 I_x, I_y, I_colour, valid，I_x, I_y 為欲顯示的螢幕座標，左上為(0, 0)，右下為(320, 240)，I_colour 為欲顯示之顏色，為 4bits 訊號再經由 module 內的 decoder 解碼為輸出訊號，valid 為此顯示訊號是否有效，輸出訊號有 hs, vs, red, green, blue，hs, vs 為行消影及列消影訊號，red, green, blue 為顏色訊號，另外有一個 frame_update 輸出訊號，用來告知外部其他 module 可以進行新的一幀畫面的改變。

此 module 內結構大致為兩個獨立的 ram 用來儲存獨立的兩幀畫面，並透過一個 FSM 來控制其讀寫，大致如下圖：



DISPLAY_OUTPUT_1 與 DISPLAY_OUTPUT_2 兩個 state 的結構可分為 5 個小部分。

1、ram 的讀寫控制，若 state 為 DISPLAY_OUTPUT_1，ram 的讀取控制為 memeory_display_1（第一組 ram）為 read（讀出）模式，memory_display_2（第二組 ram）在（valid == 1' b1）為 write（寫入）模式，(valid == 1' b0）為

read（讀出）模式，使資料不被寫入。

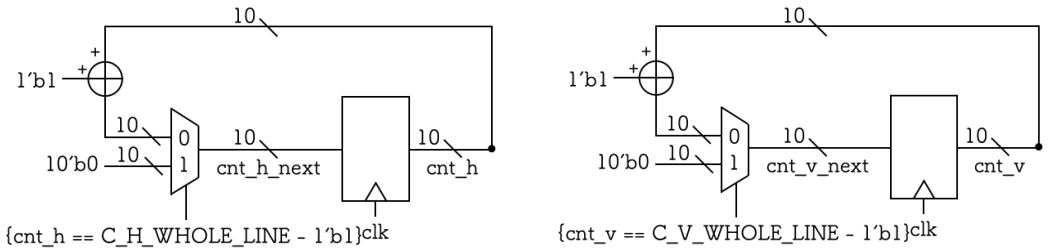
2、ram 的讀取位置，若 state 為 DISPLAY_OUTPUT_1，則 memeory_display_1（第一組 ram）的讀取位置為 addr_output，addr_output 為(O_y >> 1) * (C_H_VISIBLE_AREA / 2) + (O_x >> 1)，O_y 與 O_x 為螢幕掃描位置，因為 FPGA 板上的 ram 沒有那麼多，所以需要將圖片解析度降低再放大讀出來，上讀取位置公式的意思為將長寬皆縮小一半的的圖片，兩倍大小讀取出，memory_display_2（第二組 ram）的讀入位置為 addr_input，addr_input 為(I_y) * (C_H_VISIBLE_AREA / 2) + (I_x)，以此方式將資料讀入 ram 指定位置。

3、ram 的資料輸入，若 state 為 DISPLAY_OUTPUT_1，則 memeory_display_2（第二組 ram）為讀入模式，將外部的資料（I_colour），輸入 memory_display_2 的 dina（memory 資料輸入端口）。

4、ram 的資料讀出，若 state 為 DISPLAY_OUTPUT_1，則 memeory_display_1（第一組 ram）為讀出模式，將資料（O_colour）從 dout（memory 資料輸出端口）輸出。

5、FSM 狀態切換條件，當 frame_update == 1’ b1 時進行狀態的切換，frame_update 為畫面一幀讀取完畢的訊號，當一幀畫面輸出完畢，即將 memory_display_1 與 memory_display_2 的讀取狀態切換。

VGA 行消影與列消影的訊號產生，原理是利用類似於 Frequency Divider 的原理，在指定條件下進行訊號的切換，以便產生行消影(vs)與列消影(hs)的訊號。



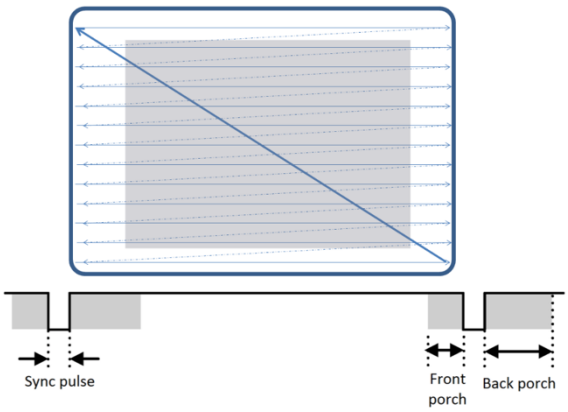
列消影訊號：

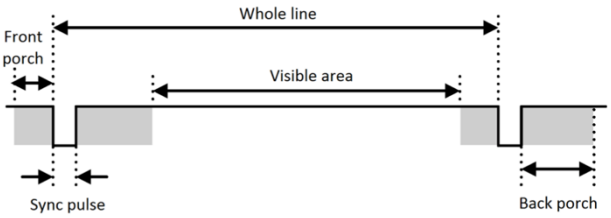
```
assign hs = (cnt_h < C_H_SYNC_PULSE) ? 1'b0 : 1'b1;
```

行消影訊號：

```
assign vs = (cnt_v < C_V_SYNC_PULSE) ? 1'b0: 1'b1;
```

關於行、列消影訊號，及其相關參數可以由下圖瞭解：





Parameter	Ver. Sync		Hor. Sync	
	Lines	Time(ms)	Pixels	Time(μs)
Visible area	480	15.3	640	25
Front porch	10	0.3	16	0.64
Sync pulse	2	0.064	96	3.8
Back porch	33	1.05	48	1.9
Whole line	525	16.7	800	32

另外還有一個訊號 active，是用來判斷是否在顯示區內，若欲顯示之圖像在顯示區內，即將其輸出，避免輸出不受控的訊號，導致螢幕發生不可預期之狀況。

因為 FPGA 板上的 RAM 容量不足，在此我們採用減少色彩 bits 數的方式來增加可儲存的資料，因此從 memory 輸出的顏色訊號(O_colour)就需要經過 decoder 進行解碼才能正確輸出至螢幕上，在此用 case 的語法進行描述，顏色轉換表附在下方。

顏色	Memory 儲存之 4bits 訊號	Vga 輸出訊號
透明	4’ d0	12’ h000
白	4’ d1	12’ hFFF
黑	4’ d2	12’ h000
淡黃	4’ d3	12’ hFEA
黃	4’ d4	12’ hFF0
淺藍	4’ d5	12’ h0AF
深藍	4’ d6	12’ h34C
淺綠	4’ d7	12’ hCF0

深綠	4’d8	12’h0D4
淺咖啡	4’d9	12’hB75
深咖啡	4’d10	12’h743
淺灰	4’d11	12’hCCC
深灰	4’d12	12’h555
紅	4’d13	12’hF00
橘	4’d14	12’hF72

以此方式達成降低儲存空間的方法。

總結以上 vga_driver.v 的功能為接收外部傳入之座標及顏色訊號，將其儲存並轉換為指定的 vga 輸出形式輸出。

第二、data_control.v

此 module 的功能為控制所有背景、障礙物、小雞的位置狀態及遊戲的進行。在下列分點介紹。

1、背景及障礙物的儲存

因為我們將整個畫面分成六排，因此每排的背景類型、障礙物類型、障礙物位置、障礙物移動狀態都需要被獨立儲存，因此我將每排的訊號用一個 30bits 的變數進行儲存，並操作他來達成其他的功能。30bits 的功能將列在下表：

背景類型	障礙物 1	障礙物 2	障礙物 1 座標	障礙物 2 座標	障礙物速度	障礙物方向
2bits	2bits	2bits	10bits	10bits	3bits	1bit

背景類型分為 4 種：

1、草地（2’b00)

2、馬路（2’b01)

3、河流（2’b10)

4、鐵軌（2’b11)

障礙物類型依背景可分為多種：

1、草地（2’b00)

一、空格（2’b00)

二、大樹 (2’ b01)
三、石頭 (2’ b10)
四、金幣 (2’ b11)
2、馬路 (2’ b01)
一、客車 (2’ b00)
二、貨車 (2’ b01)
3、河流 (2’ b10)
一、短木 (2’ b00)
二、長木 (2’ b01)
4、鐵軌 (2’ b11)
一、列車 (2’ b00)

障礙物座標為一組 10bits 的座標訊息，因為 y 座標可以透過處在第幾行得知，因此位置座標為 x 座標。

障礙物速度分成四個等級，3’ b0, 3’ b1, 3’ b2, 3’ b3 分別對應每幀移動 1, 2, 3, 4 格，以達成不同速度的效果。

障礙物方向分為左右邊，左邊為 1’ b1，右邊為 1’ b0。

2、小雞的位置狀態

小雞資料的儲存方法與背景障礙物的儲存有異曲同工之妙，分為小雞類型、小雞 x 位置、小雞 y 位置、小雞 x 方格位置、小雞 y 方格位置及小雞方向，共 30bits，將於下表呈現。

小雞類型	小雞 x 位置	小雞 y 位置	小雞 x 方格位置	小雞 y 方格位置	小雞方向
2bits	10bits	10bits	3bits	3bits	2bits

小雞類型有兩個 bits 可供切換角色使用，但因時間問題，只做出 1 種類型小雞。

小雞 x、y 位置，小雞在螢幕中的 x, y 座標。

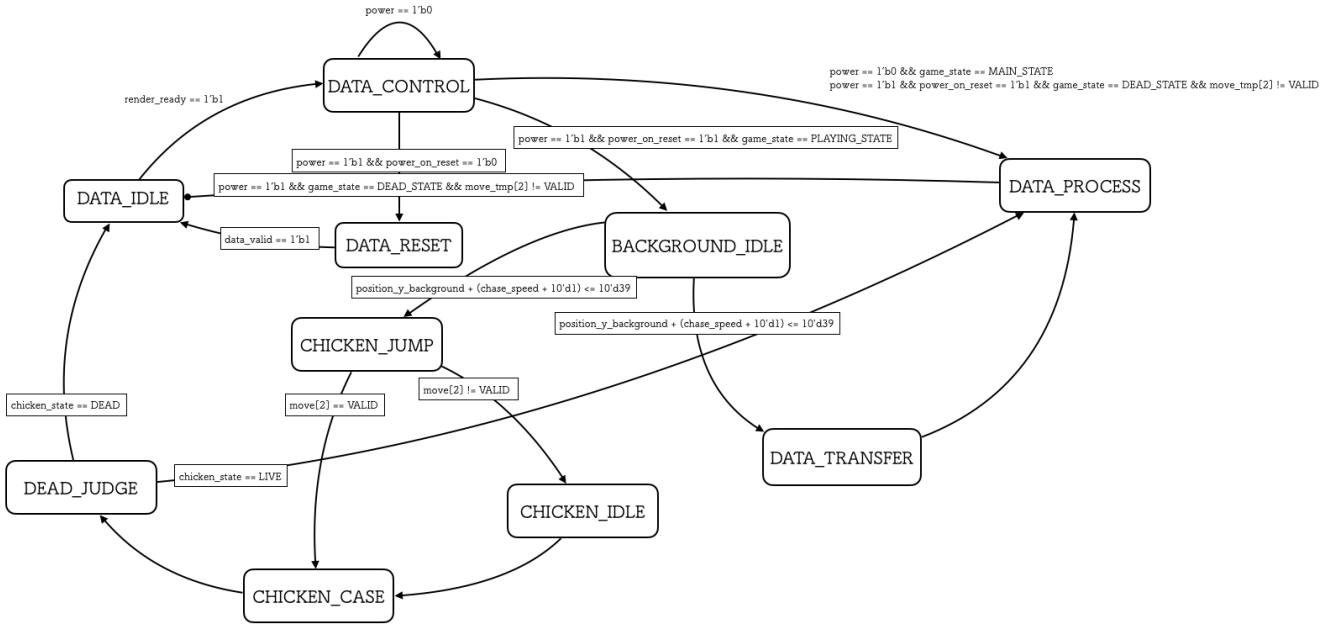
小雞 x、y 方格位置，因為有些障礙物無法跳過，因此需要有方格座標進行判斷，

並且小雞除了在河流的木頭上會隨木頭進行不在方格上的移動，因此紀錄小雞的方格座標有利於遊戲的判斷。

小雞方向，小雞有前、後、左、右四種方向，以 2bits 紀錄以便顯示之。

3、遊戲的進行

因為資料量龐大，有 7 行背景障礙物的資料加上小雞的資料，同時處理可能會造成時序混亂，先後順序很難釐清，因此在遊戲進行的資料處理我使用了一個大型的 FSM 作統一處理，此 FSM 大致可分為，終止狀態、控制狀態、背景障礙物資料處理、小雞資料處理，四種狀態，大致以下圖呈現之。



在此一一介紹各個 state：

1、DATA_IDLE：

DATA_IDLE 存在的目的是為了預防資料在一幀畫面還沒輸出完畢時就進行資料的改變，因此他跳到下一個 state 的控制條件為 render_ready == 1’ b1，也就是一幀已經處理完畢，才能進行下一步動作。

2、DATA_CONTROL：

DATA_CONTROL 的功能是判斷資料下一步要進行什麼處理，若是在電源打

開之初，會先進行 DATA_RESET（資料初始化），若是電源已經打開，且遊戲在 MAIN_STATE（等待開始畫面），則會進行 DATA_PROCESS（背景障礙物的移動），遊戲在 PLAY_STATE（遊玩階段），則會進行 BACKGROUND_IDLE（遊戲畫面的移動），遊戲在 DEAD_STATE，則會進行 DATA_PROCESS（背景障礙物的移動），做這樣的處理是為了，只有在遊戲開始的階段，背景才會逐漸向下移動，其餘狀態則是只有障礙物在左右移動。

3、DATA_RESET :

此 state 為所有資料的重置，獲取新的資料已重置遊戲。

4、DATA_PROCESS :

其是一連串動作的統稱，由 DATA_0 一直到 DATA_6，上七個 state 皆在處理背景障礙物的移動，為了縮減 code 及依序處理資料才將其分成 7 個不同的 state，但每個 state 做的事情皆相同。

透過一連串 case 及 if 語法，他會先偵測背景類型，接著判斷障礙物，再判斷速度及方向來進行障礙物在背景移動的控制。

5、BACKGROUND_IDLE :

此 state 的作用為控制整體背景的移動，在遊戲開始後，背景會逐漸向下移動，其移動速度由小雞所跳躍的格數進行控制，若往前跳躍越多，則移動速度越快，反之亦然。因為背景資料是由 7 個不同變數所控制，因此在背景完全移動至下一排時，須將資料同時傳輸至下一排，因此在背景移動一排後，會進入 DATA_TRANSFER（資料轉移），若尚未完全移動至下一排，則會進入 DATA_PROCESS 進行背景障礙物的移動。

6、DATA_TRANSFER :

此 state 的作用為資料轉移，將上排的資料轉移到下一排，如 data_1_next = data_0。為了 code 的清楚展示，因此沒有將其與 BACKGROUND_IDLE 寫在一起。

7、CHICKEN_JUMP :

此 state 的功能為控制小雞的移動，若有移動訊號(move)傳入，且可跳躍(jump_x_valid == 1’ b1)則會進行小雞的移動，若小雞有成功跳躍則會進入 CHICKEN_CASE（特殊狀況處理），若沒有成功跳躍則會進入 CHICKEN_IDLE（小雞的移動）。此 state 還會紀錄小雞目前向前最多多遠的距離，以作為分數計

算。

8、CHICKEN_IDLE :

此 state 是在控制小雞的移動，若小雞在河流上且沒有死亡，則進行左右的飄移。

9、CHICKEN_CASE :

此 state 是在判斷小雞是否遇到特殊狀況，如吃掉錢幣，並將錢幣從背景中移除，並進行加分。

10、DEAD_JUDGE :

此 state 為小雞的死亡判定，若是撞到障礙物或不在漂浮的木頭上則遊戲結束(game_state == DEAD_STATE)，若是成功存活，則進入 DATA_PROCESS 進行背景的移動。

上述大致為 data_control.v 的功能，data_control.v 中包含兩個小 module，其中一個為 dead_judge.v，另一為 jump_valid.v。

dead_judge.v

通過接收 data_control.v 中所有的 data_x 和 chicken，來進行小雞死亡的判斷，若是小雞被障礙物觸碰或是在水上但未踩在木頭上，則判定死亡，反之則否。

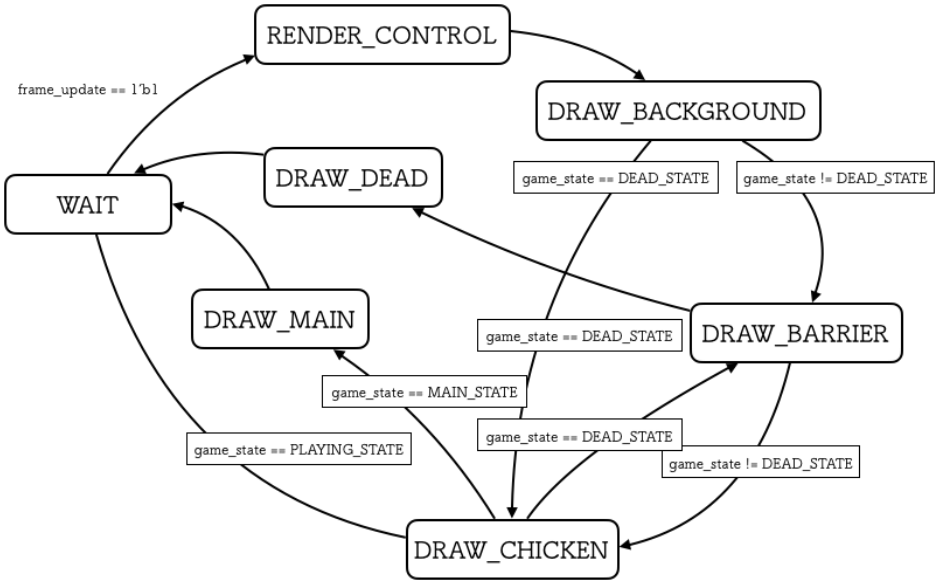
jump_valid.v

亦接收 data_control.v 中所有的 data_x 和 chicken，來判定小雞是否可以進行跳躍，若可以則輸出 jump_x_valid，否之則否。

綜合上述即可以獲得一個可以自由操作資料的 module。

第三、render.v

此 module 的功用為，將 data_control.v 的訊號(data_x, chicken)轉換成顯示訊號，並傳輸至 vag_driver 進行輸出，因為資料量大的關係，因此也是採用 FSM 的方式來進行轉換的工作，FSM 大致流程圖入下。



分別介紹其功能及工作流程。

1、WAIT :

此 state 的功能為，為了保持下一幀的畫面故定，只有在 frame_update（畫面更新後）才進行後續的繪圖工作，並進入下一階段 RENDER_CONTROL。

2、RENDER_CONTROL :

此 state 的功能為，控制繪圖狀態，在此因為功能不多，因此直接進入 DRAW_BACKGROUND（繪製背景）狀態。

3、DRAW_BACKGROUND :

通過接收來自 data_control.v 的七組 data_x 訊號，通過判斷 data_x 的背景資料來依序繪製不同的七行背景，在此統稱 DRAW_BACKGROUND，實際上有 7 個不同的 state 來繪製。繪製完背景後，會通過判定遊戲狀態的不同來進行不同工作，若遊戲不為死亡則會進入 DRAW_BARRIER（繪製障礙物），若遊戲狀態為死亡則會先進入 DRAW_CHICKEN（繪製小雞），因為小雞死亡後會被障礙物蓋過，因此會先繪製小雞才繪製障礙物。

4、DRAW_BARRIER :

通過接收來自 data_control.v 的七組 data_x 訊號，通過判斷 data_x 的障礙物資料來依序繪製不同的障礙物，在此統稱 DRAW_BARRIER，實際上有 7 個不同的 state 來繪製。繪製完障礙後，會通過判定遊戲狀態的不同來進行不同工作，若遊戲不為死亡則會進入 DRAW_CHICKEN（繪製小雞），若遊戲狀態為死亡則會先進入

DRAW_DEAD（繪製死亡畫面）。

5、DRAW_MAIN :

繪製遊戲初始畫面「暴走小雞」。及下方手手圖示。

6、DRAW_DEAD :

繪製遊戲的死亡畫面「再快也是 7 天後到家」。

綜合上述並通過工作流程圖，即可以獲得 render.v 完整功能。

第四、input_control.v

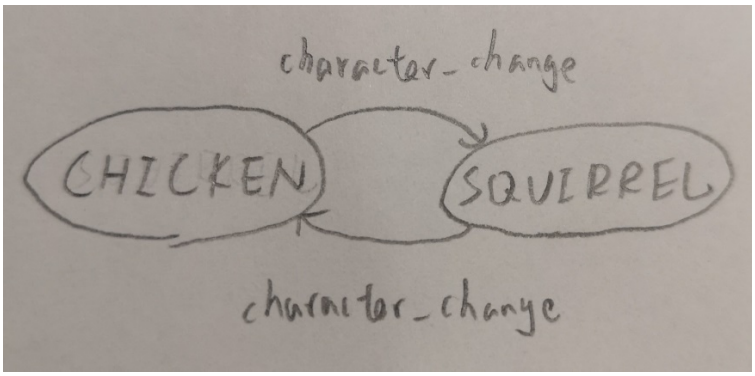
一、將每個 push button 經過 one_pulse 的操作

將五個 push button 的訊號都先丟到 one_pulse.v 的 module 裡。

二、遊戲角色的選定

我們預設的角色為小雞，但當累積的金幣大於 3 個的時候我們就可以使用松鼠，所以當金幣大於 3 時，我們就會自動跳到使用松鼠這個角色來進行遊戲。

之後利用一個只有兩個 state 的小的 FSM，來進行使用角色轉換的控制，如果有 character_change 的訊號傳進來，就會進到不同的 state，不然就是繼續維持現在所使用的角色 state。



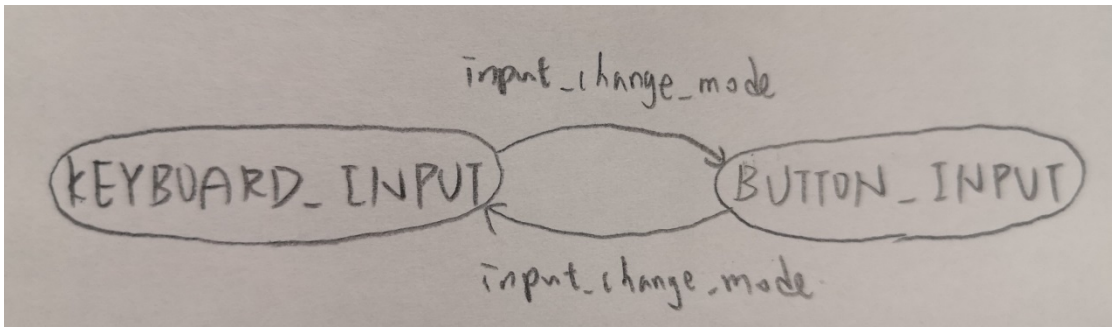
三、遊戲操作方式的選定

我們預設這個遊戲的操作控制是利用 push button，但因為 push button 並沒有那麼好操作，所以我們讓這個遊戲可以有兩種方式(push button 及 keyboard)來進行控制角色的上下左右移動，同樣利用有兩個 state(BUTTON_INPUT 和 KEYBOARD_INPUT)的 FSM 來進行兩者之間轉換使用的控制。

1. 當要使用 push button 時(在 BUTTON_INPUT 的 state)，將四個 button 的訊號分

別丟到 input_up、input_down、input_left、input_right，也就是只要按到對應的 button 就可以對角色進行對應的移動控制，但當有 input_change_mode 的訊號進來後，就會改變遊戲的操作方式，而 state 就會進到利用 keyboard 操作的 state(KEYBOARD_INPUT)。

2. 當要使用 keyboard 時(在 KEYBOARD_INPUT 的 state)，我們利用 key_down 的訊號編碼，將四個按鍵的的訊號同樣分別丟到 input_up、input_down、input_left、input_right，也就是只要按對應的 keyboard 按鍵就可以對角色進行對應的移動控制，但當有 input_change_mode 的訊號進來後，就會改變遊戲的操作方式，而 state 就會進到利用 push button 操作的 state(BUTTON_INPUT)。



四、利用 push button 的長按短按分辨要進行角色選擇還是操作方式選擇

我們利用五個 push button 裡正中間的那個來進行角色選擇及操作方式選擇，先用 if 來進行判斷，當正中間的 bush button 被按著時，經過一個 clock，timer 就會加 1，當成功進行角色或操作方式的改變後，timer 就會歸 0，等到需要再進行改變時再開始進行計算。

之後再利用 if 來判斷，當 timer 的結果在 0~32 之間(短按)時，就是改變角色的選擇，也就是讓 character_change = 1'b1，當 timer 數超過 32(長按)時就是進行操作方式的改變，也就是讓 mode_change = 1'b1，再經過 one_pulse 後的訊號 input_change_mode，就可以當成是前面要改變操作方式的訊號輸入。

五、分辨上下左右的移動

利用 if 來分辨，讓 3 個 bits 的 move 訊號可以因為操作方向的不同而有不同的結果，傳到其他 module 後才知道該如何進行控制，我們將

1.往上為 100，如果有 output_up 的訊號進來的話，move 就會是 3'b100。

2.往下為 101，如果有 output_down 的訊號進來的話，move 就會是 3'b101。

3.往左為 110，如果有 output_left 的訊號進來的話，move 就會是 3'b110。

4.往右為 111，如果有 output_right 的訊號進來的話，move 就會是 3'b111。

第五、ssd_control.v

一、七段顯示器顯示的控制

在這個 module 中我們主要分成五個 state，利用 FSM 讓他們在每個 state 間互相轉換，讓四個七段顯示器可以在對應的情況下顯示出我們所需要的值。

1. STATE_MAIN

這個 state 是在遊戲未開始的主畫面的情況，所以四個七段顯示器會顯示目前的最高分(ssd_1 = digit_1~ssd_4 = digit_4)。

(1) 如果有 PLAYING_STATE 的訊號進來，就會進到開始遊戲的 state(STATE_PLAY)，來顯示 play 的字。

(2) 如果有 DEAD_STATE 的訊號進來，就會進到遊戲結束角色死亡的 state(STATE_DEAD)，來顯示 dead 的字。

(3) 如果持續是 MAIN_STATE 的訊號進來，就會保持住現在的 state(STATE_MAIN)持續顯示最高分。

2. STATE_SCORE_PLAY

這個 state 是在遊戲正在進行時的分數計算情況，會隨著往前跳往上一直增加得分數，所以就是讓目前的得分數顯示在四個七段顯示器上(ssd_1 = digit_1~ssd_4 = digit_4)。

(1) 如果持續是 PLAYING_STATE 的訊號進來，就會維持在現在的 state(STATE_SCORE_PLAY)繼續進行遊戲分數的計算增減。

(2) 如果有 DEAD_STATE 的訊號進來，就會進到遊戲結束角色死亡的 state(STATE_DEAD)，來顯示 dead 的字。

(3) 如果是 MAIN_STATE 的訊號進來，就會進到遊戲主畫面的 state(STATE_MAIN)，也就是顯示目前的最高得分數。

3. STATE_PLAY

這個 state 是讓我們可以從七段顯示器上看出現在遊戲是否已經開始，如果進到這個 state，四個七段顯示器就會出現 play 的字，之後再隨著傳入的訊號不同而跳到不同的 state

(1) 如果是 PLAYING_STATE 的訊號進來，就會進到開始遊戲後，要跟著角色計分的 state(STATE_SCORE_PLAY)來進行遊戲分數的計算增減。

(2) 如果有 DEAD_STATE 的訊號進來，就會進到遊戲結束角色死亡的 state(STATE_DEAD)，來顯示 dead 的字。

(3) 如果是 MAIN_STATE 的訊號進來，就會進到遊戲主畫面的 state(STATE_MAIN)，也就

是顯示目前的最高得分數。

4. STATE_DEAD

這個 state 是讓我們可以從七段顯示器上看出現在角色是否已經死亡，如果進到這個 state，四個七段顯示器就會出現 dead 的字，之後再隨著傳入的訊號不同而跳到不同的 state。

(1) 如果是 PLAYING_STATE 的訊號進來，代表開啟了新的一局，就會進到開始遊戲後，要跟著角色計分的 state(STATE_PLAY)來進行遊戲分數的計算增減。

(2) 如果持續是 DEAD_STATE 的訊號進來，就進到遊戲結束角色死亡顯示最後得分的 state(STATE_SCORE_DEAD)。

(3) 如果是 MAIN_STATE 的訊號進來，就會進到遊戲主畫面的 state(STATE_MAIN)，也就是顯示目前的最高得分數。

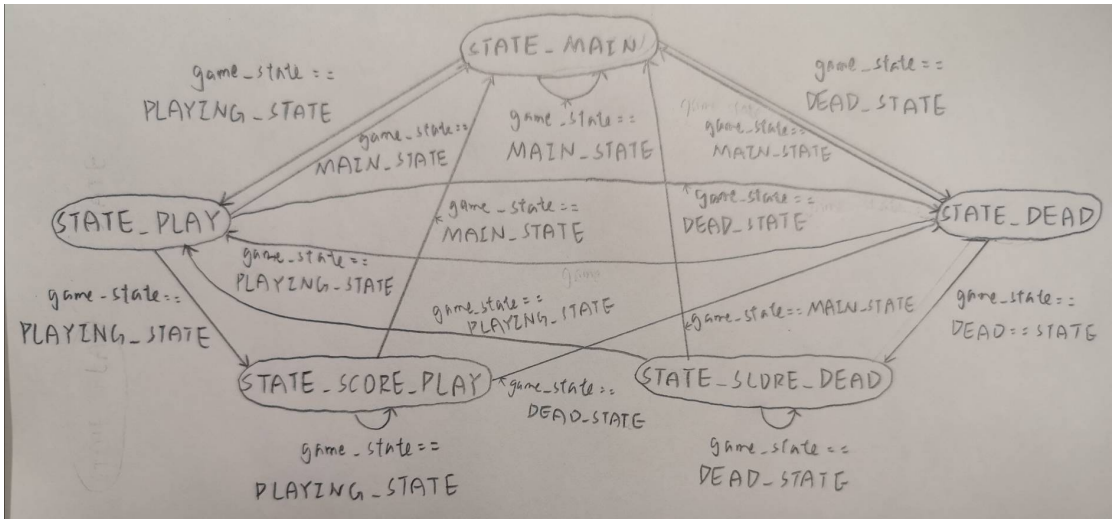
5. STATE_SCORE_DEAD

這個 state 是讓我們可以在角色死亡後在七段顯示器上看到最終的得分，如果進到這個 state，四個七段顯示器就會出現最後的分數(ssd_1 = digit_1~ssd_4 = digit_4)。

(1) 如果有 PLAYING_STATE 的訊號進來，代表開啟了新的一局，就會進到開始遊戲的 state(STATE_PLAY)來顯示 play 的字。

(2) 如果是 DEAD_STATE 的訊號進來，就保持在遊戲結束角色死亡顯示最終分數的 state(STATE_SCORE_DEAD)。

(3) 如果是 MAIN_STATE 的訊號進來，就會進到遊戲主畫面的 state(STATE_MAIN)，也就是顯示目前的最高得分數。

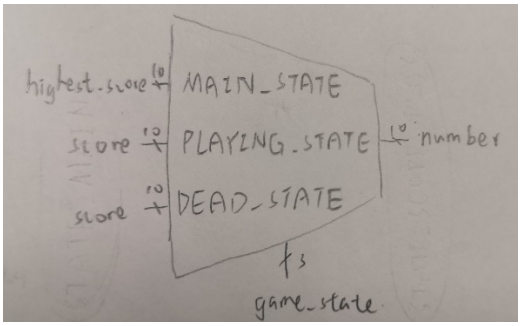


二、分數的顯示

我們將二進位轉成了十進位，讓結果可以以我們平常所熟悉的方式呈現在七段顯示器上。

因為我們會顯示兩種分數，一種是最高分 highest_score，另一種是目前的分數 score，所以我們先用了一個 if 來判斷，在 dFF 存值時就選擇是要維持目前的最高分數還是要進行最高分的更新，如果今天所累計的分數(score)大於 highest_score 時，就會將 score 的值存到 highest_score 來進行最高分的更新。

之後同樣利用 if 來判斷各個不同的 state 要顯示的是最高分(highest_score)還是目前的分數(score)，如果在遊戲的主畫面(MAIN_STATE)時，就是顯示最高分(highest_score)，如果是在遊戲進行中及死亡畫面時(PLAYING_STATE 及 DEAD_STATE)，就是顯示這局目前所得到的分數。



三、金幣數計算及 LED 燈的控制

接下來是獲得金幣數的計算，因為我們預設的角色是小雞，如果想要換成松鼠的話，就必須先累積吃到三個金幣(我們將吃到的金幣數量顯示在 LED 燈上)，使用一次松鼠角色就需要花費三個金幣，所以接下來是金幣計算的部分。

因為遊戲狀態前後的不同會有不同的情況，所以我們將前面 state 的狀態存到 shift register 內，利用六個 bits 就可以存下前後連續兩個 clock 的狀態。

利用 if 來辨識金幣數量的增減:

1. 當 state_recognizer[5:3] == MAIN_STATE && state_recognizer[2:0] == PLAYING_STATE 代表從起始主畫面的 state 進到了開始遊戲的 state，同時又如果 character == 1'b1(代表選了松鼠來進行遊戲)，我們就會將累計的金幣數量減 3。
2. 當 state_recognizer[5:3] == PLAYING_STATE && state_recognizer[2:0] == DEAD_STATE 代表從遊戲進行的 state 進到了遊戲結束的 state，也就是將本局所吃到的金幣數量加到目前累積的數量上面。

最後為了方便看說我們現在可以使用多少次的松鼠，我們讓累積三個金幣後才會亮第一個 LED 燈，累積六個才會亮第一和第二個 LED 燈...以此類推，也就是

coin_store > 10'd3 && coin_store <= 10'd6 會亮一個燈

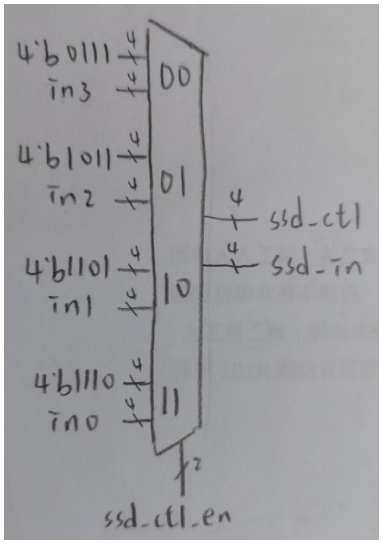
coin_store > 10'd6 && coin_store <= 10'd9 會亮兩個燈…

藉由這樣我們就可以清楚的看出來我們目前的金幣數量可以讓我們使用多少次的松鼠角色。

四、加入 scan_ctl.v 及 display.v

在 scan_ctl 的 clk_ctl 輸入會有四種情況(00,01,10,11)分別代表一個七段顯示器，讓他們輪流亮，並且同時給它們值，也就是將前面的 ssd_1~ssd_4 的結果分別丟到 in0, in1, in2, in3 當成要輸出的值(ssd_in)，再將七段顯示器分別要亮的結果(ssd_ctl)當成最後的輸出。

將每個數字及字母的輸入進行解碼，讓顯示在七段顯示器上時是我們平常所習慣的數字及字母樣子，也就是將 play、dead 跟各個分數的數字，當成是輸入(bin)，最後將解碼後的結果當成是輸出(seg)，即可成功顯示在七段顯示器上。



數字與字母在七段顯示器顯示的對應解碼:

0	1	2	3	4	5
00000011	10011111	00100101	00001101	10011001	01001001
6	7	8	9	a	
01000001	00011111	00000001	00001001	00110001	
b	c	d	e	f	
11100011	00010001	10001001	10000101	01100001	

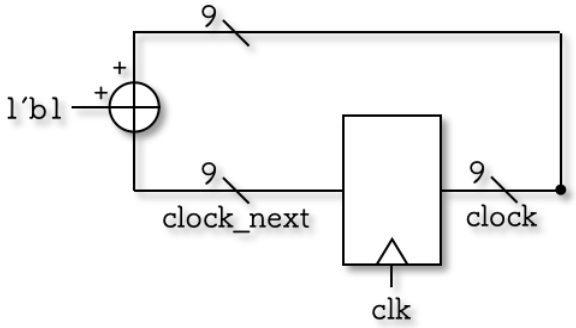
第六、sound.v

整個遊戲過程中會出現三種音樂:遊戲的背景音樂、火車要來的提示聲、角色死亡時的音樂，整個過程我分成以下三部分

一、speaker_control.v

此為一個音訊資料 parallel to serial 轉換器，並輸出 ADC 所需之 clock 訊號，25Mhz 的 master clock(audio_mclk)，(25Mhz/128)的 Left-Right clock(audio_lrck)，和(25Mhz/4)的 sampling clock(audio_sck)。

此設計分成兩個小部分，第一個部分負責產生不同頻率的輸出訊號，因為輸出頻率都是石音振盪器產生的 100Mhz 頻率(clk)，除以 2 的次方倍，master clock(audio_mclk)為(100Mhz/4)，Left-Right clock(audio_lrck)為(100Mhz/512)，sampling clock(audio_sck)為(100Mhz/16)，因此只要設計一個 up-counter，並使之以 100Mhz(clk)的頻率運行，並在每個週期都往上數一，因為 2 進位的特性，在經過一個 flip-flop 後即可獲得頻率除以的訊號，因此只要將 audio_mclk 接在 counter 的第 2 腳(clock[1])，audio_sck 接在 counter 的第 4 腳(clock[3])，audio_lrck 接在 counter 的第 9 腳(clock[8])，即可以產生欲得到頻率的訊號。



Logic function for audio_mclk:

audio_mclk = clock[1]

Logic function for audio_sck:

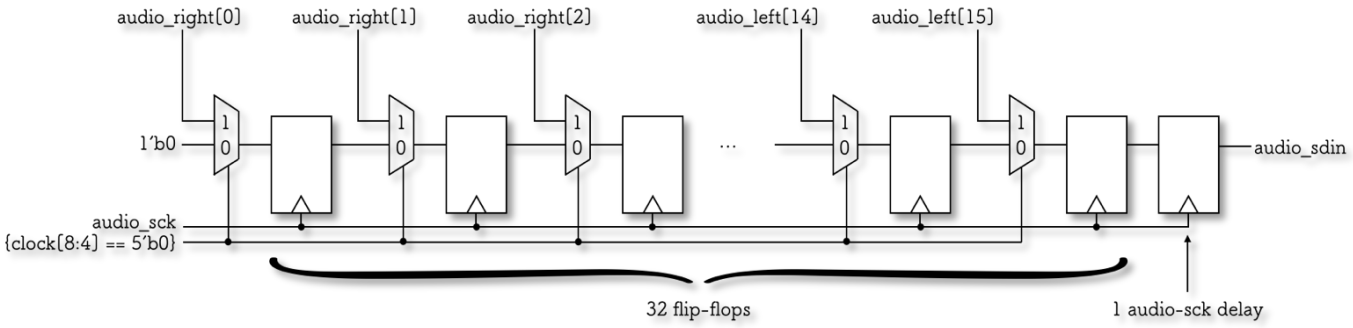
audio_sck = clock[3]

Logic function for audio_lrck:

audio_lrck = clock[8]

第二部分為 parallel to serial 的轉換器，想法是每一個 audio_lrck 週期將 audio_left 與 audio_right 的 16bits 訊號輸入至一個 shift register，並使 shift register 以 audio_sck 的頻率運行，在此獲得 audio_lrck 的週期方式為，當 counter 的第 5~9 位皆為 0 時，則

週期會恰為 audio_lrck(100Mhz/512)，再將 shift register 的輸出輸入一個 flip-flop 做 1 個 audio_sck 的 delay，以符合 protocol 的規定，最後得到 serial 輸出 audio_sdin。



綜合上述，即可以獲得題目所要求的音訊轉換器。

二、tone.v

1.三種音樂每個音出現頻率為 frequency divider 所除出的頻率，其中我們將 clk_1 設為「背景音樂」以及「火車的提示聲」的頻率，clk_3 設為「角色死亡音樂」的頻率。

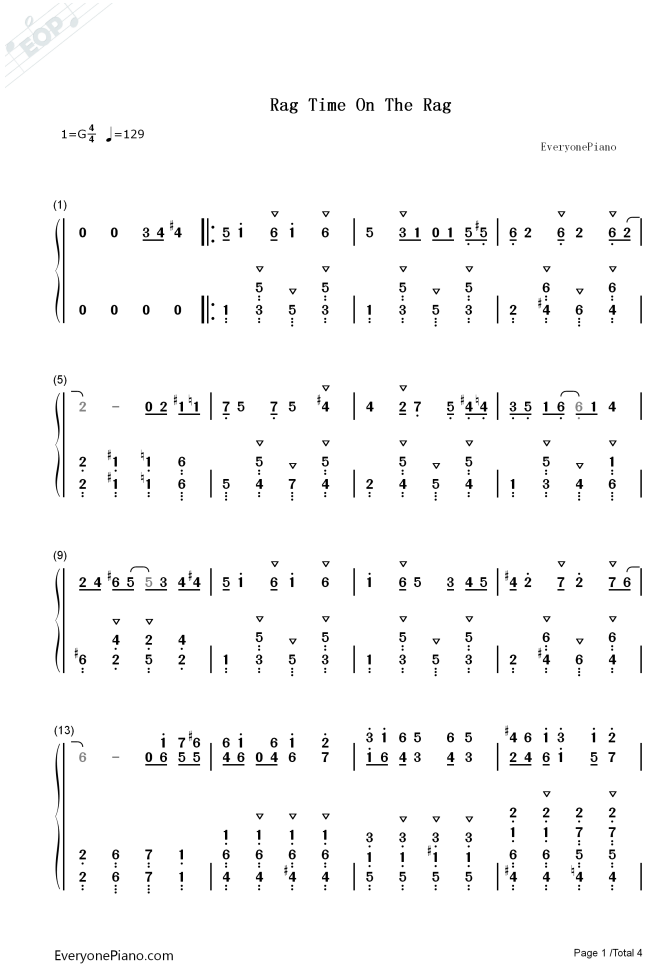
(1)要先有一個頻率為 5 的 clock，讓等等的音樂可以以 0.2 秒一個音的速度播出，所以我寫了一個 frequency divider 來進行除頻，在以原本石英震盪器的頻率數到 10000000(十分之一)時，將我所設的 clk_1 翻轉一次，這樣完整 clk_1 的一個 clock 就會是兩個 0.1，也就是 0.2 秒，最後將 clk_1 的結果輸出，就有一個頻率為 0.2 的 clock 了。

(2) 要先有一個頻率為 10 的 clock，讓等等的音樂可以以 0.1 秒一個音的速度播出，所以我們寫了一個 frequency divider 來進行除頻，在以原本石英震盪器的頻率數到 5000000(二十分之一)時，將我所設的 clk_3 翻轉一次，這樣完整 clk_3 的一個 clock 就會是兩個 0.05，也就是 0.1 秒，最後將 clk_3 的結果輸出，就有一個頻率為 0.1 的 clock 了。

2.先將三種音樂進行編碼，隨後利用四個 shift register(q_left、q_right、q_light、q_dead)將三首音樂的音存下來，再一個音一個音播出。

(1)第一首歌為背景音樂「黃金傳說(Rag Time On The Rag)」，因為我們將這首歌的主旋律和副旋律分為左右聲道進行輸出，所以我將主旋律存入 q_left 將副旋律存入 q_right，利用 clk_1 的訊號，一次將第一個音移到最後一個，再將其他的依序往前遞補，並把第一個音 q_left[639:635]的結果丟到 note_left 以及 q_right[639:635]丟到 note_right。

背景音樂所使用的譜:



背景音樂主旋律音符編號:

音符(簡)	0	5	6	7	1	2	3	4
編號	0	1	2	3	4	5	6	7
音符(簡)	5	6	7	1̇	2̇	3̇	d4	
編號	8	9	10	11	12	13	14	
音符(簡)	#4	#5	d1	#1	#4	#6	3	
編號	15	16	17	18	19	20	21	

背景音樂副旋律音符編號:

音符(簡)	0	4	5	6	7	1	2	3
編號	0	1	2	3	4	5	6	7
音符(簡)	4	5	2	#1	#4	#6	d1	d4

編號	8	9	10	11	12	13	14	15

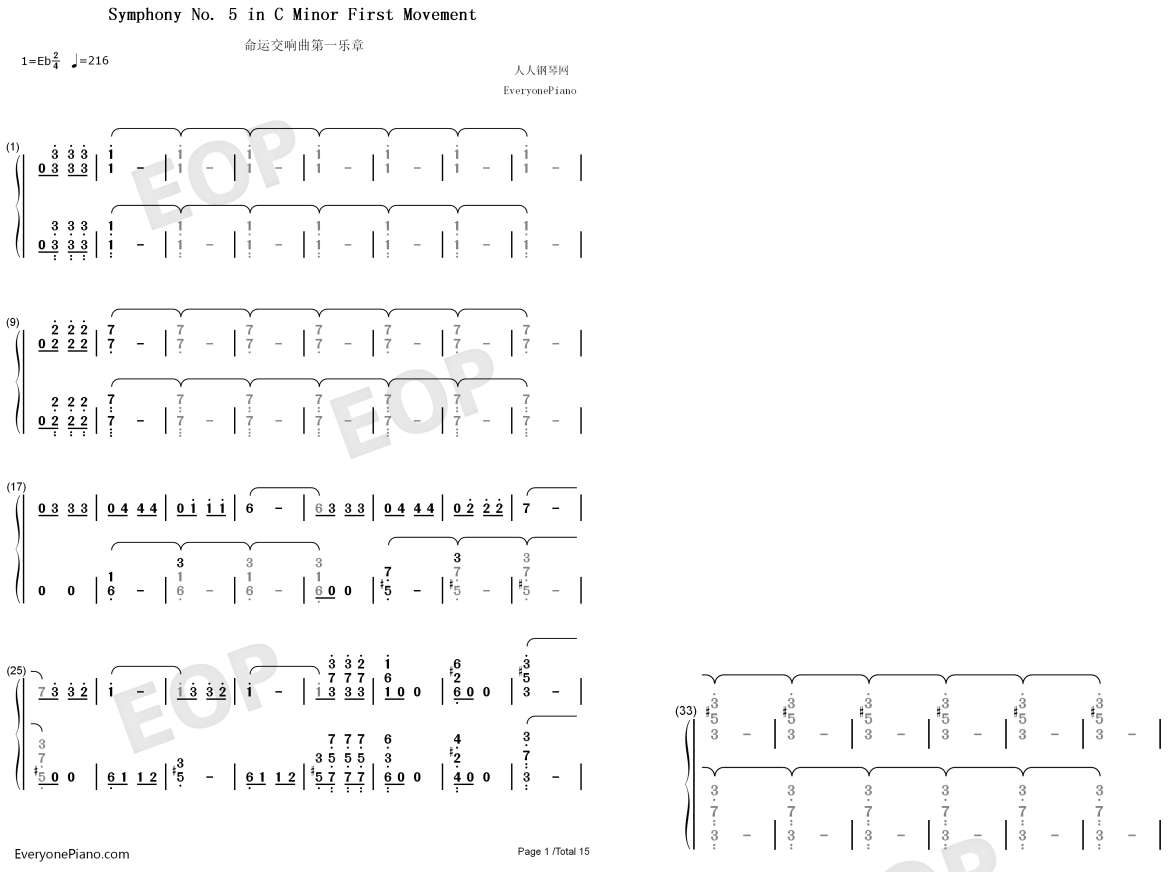
(2)再來我們火車的提示聲為單個音和沒聲音，展現出現實中噹噹噹的樣子，所以這裡 q_light 較為簡單，就只有兩個數字利用 clk_1 的訊號在互相交換，同樣也將第一個音 q_light[1]的結果丟到 note_light。

火車提示音音符編號:

音符(簡)	0	1
編號	0	1

(3)最後的是死亡時的音樂「命運交響曲」，我們同樣將這首歌的主旋律存到 q_dead 中，利用 clk_3 的訊號，一次將第一個音移到最後一個，再將其他的依序往前遞補，並同樣把第一個音的結果 q_dead[775:772]丟到 note_dead。

死亡音樂所使用的譜(僅取主旋律的部分):



死亡音樂音符編號:

音符(簡)	0	$\underline{6}$	$\underline{7}$	1	2	3
編號	0	1	2	3	4	5
音符(簡)	4	6	7	$\dot{1}$	$\dot{2}$	$\dot{3}$
編號	6	7	8	9	10	11

3.利用 if 來進行判斷現在要播的是哪首音樂

(1)當有 dead 訊號(角色死亡)的時候，播的音樂為命運交響曲，所以就將前面得到的 note_dead 進行解碼的動作，判斷那個音的頻率後，將結果存在 out_note_left 和 out_note_right 就可以將結果輸出，就可以得到我們現在需要的頻率。

背景音樂主旋律編號代表頻率:

編號	0	1	2	3	4	5	6	7
頻率	0	255102	227272	202478	191571	170648	151515	143266
編號	8	9	10	11	12	13	14	
頻率	127551	113636	101215	95420	85034	75758	303380	
編號	15	16	17	18	19	20	21	
頻率	270270	240790	202478	180388	135139	107259	303380	

背景音樂副旋律編號代表頻率:

編號	0	1	2	3	4	5	6	7
頻率	0	1145475	1020408	909091	809848	764409	681013	606722
編號	8	9	10	11	12	13	14	15
頻率	572672	510204	341577	721501	540541	429037	809848	606722

(2)當有 light 訊號(有火車要來了)的時候，左聲道播(維持)的是背景音樂的主旋律，右聲道會變成火車的提示聲，所以兩邊要分開進行解碼，利用 note_left 進行解碼並將結果存在

out_note_left，利用 note_light 進行解碼並同樣將結果存在 out_note_right(火車提示聲就固定為單個音 Do)，之後就可以將結果輸出，就可以得到我們現在需要的頻率。
火車提示音編號代表頻率:

編號	0	1
頻率	0	95420

(3)當沒有 dead 及 light 的訊號時，就是在一般的狀態，那就是左耳播背景音樂的主旋律，右耳播背景音樂的副旋律，所以也需要分開進行解碼，利用 note_left 進行解碼並同樣將結果存在 out_note_left，利用 note_right 進行解碼並將結果存在 out_note_right，之後就可以將結果輸出，就可以得到我們現在需要的頻率。

死亡音樂編號代表頻率:

編號	0	1	2	3	4	5
頻率	0	227272	202478	191571	170648	151515
編號	6	7	8	9	10	11
頻率	143266	113636	101215	95420	85034	75758

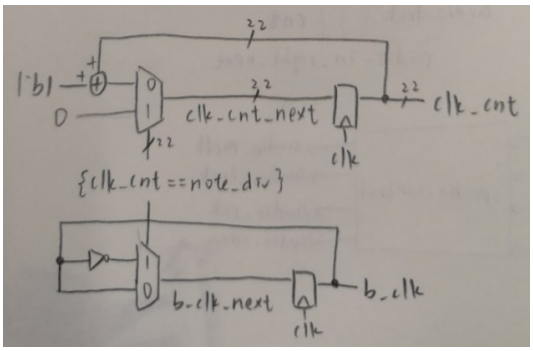
三、buzzer_control_left.v + buzzer_control_right.v

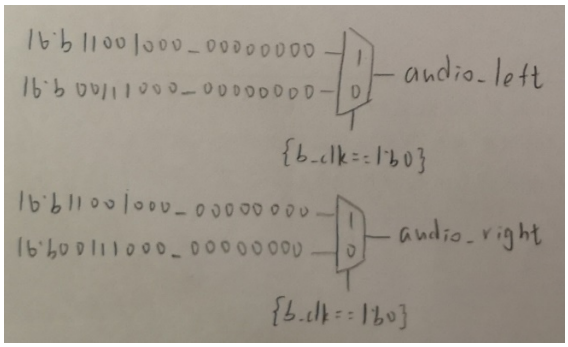
在 buzzer_control_left(buzzer_control_right)中我希望可以控制聲音的大小聲和高低，也就是藉由控制振幅和頻率來達到目的。

我先將左右兩邊分別我所想要的頻率數值傳進來(note_div)，然後寫一個 counter，在 counter 數到所需要的頻率的 1/2 時，就將所設定的 b_clk 翻一次，這樣完整的 b_clk 週期就會是我想到的頻率，也就是我想要的音調。

然後在 b_clk 為 0 的時候，在 audio_left(audio_right)丟入一個負數，讓波型是在平衡線底下的，在 b_clk 為 1 的時候，在 audio_left(audio_right)丟入一個正數，讓波型是在平衡線上面的，然後藉由丟入 audio_left(audio_right)的數值絕對值大小來控制大小聲，所以丟入的負數的絕對值要等於丟入的正數，這樣聲音才不會有忽大忽小的情況。

最後將控制左(右)聲道的 audio_left(audio_right)設為輸出，就會是我所想要的音調，同時又可以控制大小聲了。





Discussion

VGA 的時序有問題，看老師的講義後自己還花了一段時間摸索才成功。

FSM 的 latch 要相當注意，要把所有情況都寫進去來消除。

FPGA 板的儲存空間有限，所以我們在能省的地方會盡量節省，導致我們在畫全部的圖的時候都只有用 16 種顏色就將它完成。

vivado 合成花了很多時間，邊跑就要開始編想要在改甚麼地方，一直在跟時間賽跑，滿累的。

Conclusion

陳均豪

這一學期下來，跟別人比起來，我好像進步的幅度好小，我寫的 code 跑一次 generate bitstream 的時間，比別人完成完整的一份 final project，包含 code、錄影、結報的時間還要長，我不知道是寫 verilog code 的能力有問題，還是某些人真的太厲害了，有家裡的哥哥或姊姊可以協助他們，讓他們一個小時內可以完成的事，我卻要花上一個月，真的是感到很錯愕。

施品瑄

這一整個學期做了很多的 lab，在最後的 final project 就是將我們所能想到的所有東西結合在一起，還滿有成就感的，雖然延後了繳交時間，但還是覺得很趕，看來寫 verilog 的能力必須要再繼續好好加強才行。