

Final project: zombie killer

電機系四年級華班 (107011133) 王靖淳

電資班一年級 (110060034) 沈安婕

專題題目講解:

本期末專題將設計與實現殺殭屍遊戲，遊戲會有三條跑道，每條跑道上會隨機產生殭屍，並設定我們能夠殺到殭屍的範圍，當跑道上的殭屍跑進該範圍，我們可以利用鍵盤按下對應該跑道的按鈕以殺死殭屍，除了成功殺死所有殭屍外，我們也設立了另一項門檻，限制射殺次數，不能夠有誤殺的狀況發生，當沒有殭屍出現在射擊範圍內不能按鍵盤按鈕。

遊戲的計分方式分為兩種，一種是成功打到所有進入射擊範圍的殭屍，所以這裡會記錄出現的殭屍總數以及殺到的殭屍數，若有沒殺到的殭屍遊戲仍會繼續直到設定的時間結束，可以看到自己殺到的數量與實際出現的殭屍總數來衡量自己在遊戲中是否有進步。另一種是是否有誤殺的狀況發生，若該範圍沒有殭屍出現，卻按下對應的按鈕表示誤殺，這裡會記錄出現幾次誤殺的狀況。當遊戲結束時成功殺掉所有殭屍且沒有誤殺就贏了，但只要上述兩種條件有一種沒達成就輸了。

功能:

透過 LCD 螢幕顯示 3 跑道與 3 個殭屍的出現與否，經由 Keyboard 的數字鍵 1、2、3 對應三個跑道的射擊，判斷射殺數量是否和出現總數相同以及是否有誤殺發生決定是否過關。七段顯示器顯示遊戲的輸贏結果(LOSE/WIN)，利用 FPGA 板上的按鈕做切換，可在七段顯示器看到自己的射殺數量、實際出現的總數以及誤殺的次數。以殭屍出現的速度分為兩關卡，DIP SWITCH 用來調整關卡難度，LED 顯示關卡難度以及遊戲的狀態(起始畫面/說明 level/開始遊戲/遊戲結束/顯示射殺數量/顯示誤殺次數)，SPEAKER 會在遊戲狀態為遊戲中時重複地播放歌曲，VGA 顯示遊戲畫面。

Design Specification:

Input: clk, rst, btn, btn_score, btn_miss, switch

Inout: PS2_CLK, PS2_DATA

Output: [3:0] vgaRed, [3:0] vgaGreen, [3:0]vgaBlue,
hsync, vsync, switch_led, [5:0] led, [3:0] an, [7:0] ssd, audio_mclk,
audio_lrck, audio_sck, audio_sdin

Input	name	Function	Output	name	Function
Button S1	btn	切換遊戲狀態	7-segment digit 0	[3:0] an, [7:0] ssd	顯示第 0 位數成 績或是輸贏結果
Button S2	btn_score	切換七段顯示器 以顯示射殺數量	7-segment digit 1		顯示第 1 位數成 績或是輸贏結果
Button S3	btn_miss	切換七段顯示器 以顯示誤殺次數	7-segment digit 2		顯示第 2 位數成 績或是輸贏結果
DIP switch1	switch	選擇關卡難度	7-segment digit 3		顯示第 3 位數成 績或是輸贏結果
Keypad1	PS2_CLK, PS2_DAT A	射擊左跑道殭屍	LED light D1	Switch_led	顯示關卡難度
Keypad2		射擊中跑道殭屍	LED light D2~D7	[5:0] led	顯示遊戲狀態
Keypad3		射擊右跑道殭屍	LCD Display	[3:0] vgaRed, [3:0] vgaGreen, [3:0]vgaBlue, hsync, vsync	射擊殭屍遊戲
			Speaker	audio_mclk, audio_lrck, audio_sck, audio_sdin	播音樂

The diagram illustrates a complex game console system architecture, divided into several functional blocks:

- System Control & Timing:** Includes a **clock divider** and **clock divider LFSR** for clock generation. A **VGA controller** manages video output, receiving **valid** and **sync** signals. A **mem. addr. generator** and **blk. mem. gen.** handle memory addressing, with a **17 pixel-addr** signal.
- Game Logic & State:** The **LFSR** (Linear Feedback Shift Register) provides random numbers. A **game minute** counter is updated by **rst** and **clk**. The **stop-tag** and **stop-tag onepulse** signals are used for state transitions. The **FSM** (Finite State Machine) manages game states, receiving **btn.score**, **btn.miss**, and **btn** (change state) inputs.
- Audio & Sound:** The **audio** block processes **sound.top**, **sound.mid**, and **sound.bot** signals, outputting **audio.mch**, **audio.trk**, **audio.sok**, and **audio.sdin**.
- Input & Output:** The **keyboard decoder** processes **clk**, **rst**, **PS2-DAT**, and **PS2-CLK** signals. It outputs **key.down** and **key.up** signals to the **MUXs** (Multiplexers). The **MUXs** also receive **input** and **state (sel)** signals.
- Gameplay & Scoring:** The **zombie select** block manages game elements, receiving **total**, **score**, **miss**, and **miss_num** signals. It outputs **total**, **score**, **miss**, and **miss_num** signals. The **value** block processes **value0**, **value1**, **value2**, and **value3** signals. The **score-to-ssd** block converts scores to SSD (Solid State Drive) format, outputting **ssd** signals.
- Display & Graphics:** The **RGB** block outputs **RGB** signals to the **VGA** controller. The **ssd** block outputs **ssd** signals to the **ssd** controller.

主架構思想：

先用 LFSR 決定跑道上端是否出現殭屍，並讀取 keyboard 的訊號殺死殭屍，zombie_select 依照 LFSR 與 keyboard 訊號決定分割的區域是否有殭屍(起始使否有殭屍、打到殭屍、未打到殭屍皆會影響該區域的殭屍數目，此 module 中有多個 mux 來判斷各種情況)，並進行射殺數量 score，殭屍總數 total，誤殺次數 miss 的數值累計，game_minute 則控制遊戲進行時間以及使遊戲結束時停止產生殭屍。透過 vga 影像處理產生遊戲畫面，還有透過 bottom 調控模式切換(控制 FSM state)和 switch 調控關卡難度(改變 clock)，最後再處理音樂、led 亮暗、7-seg-display 的顯示。

Bottom---module: debounce, onepulse, fsm

按下 bottom 來切換 state(起始畫面(idle)、說明 level(speed)、開始遊戲(play)、遊戲結束(stop))，每按一次 bottom 就會跳至下一個 state。

遊戲結束(stop)時，七段顯示器會顯示結果: win or lose。

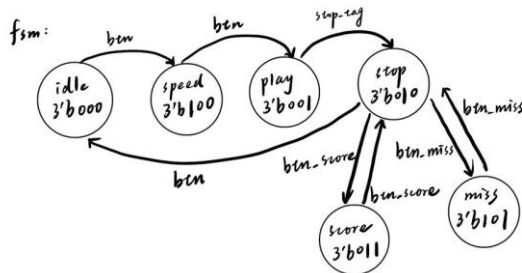
此時按下 score bottom 可顯示殺死殭屍個數和總共出現殭屍數(state: score)。

再按下 miss bottom 顯示錯殺次數(未出現殭屍卻按下按鍵)(state: miss)。

製作:(參照 lab5)

將 btn, btn_score, btn_miss 經過 debounce 和 onepulse 的處理，並用產生的 3 個 onepulse 來切換 fsm 的 state(idle/ speed/ play/ stop/ score/ miss)，即達成按下特定按鈕,切換至特定 state 的成果

以下說明 fsm module:



其中 game_minute 產生 stop_tag_onepulse 表示遊戲真正結束，作為 fsm 的 select 從 state: play 跳至 state: stop，切換到停止畫面，此處不須按 bottom。

在停止畫面時，按下 score bottom 可跳至 state: score，再按一次同一顆 bottom 可再回到 state: stop。

在停止畫面時，按下 miss bottom 可跳至 state: miss，再按一次同一顆 bottom 可再回到 state: stop。

7-seg-display--module: value, clock_divisor_ssd, bcd_to_ssd, ssd_ctl

功能:

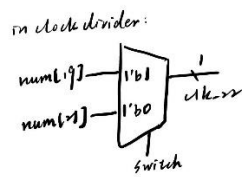
顯示結果:win or lose 以及顯示數值:score number, total number 和 miss number

製作:(參照 lab8)

由 value module 分別將 win or lose 轉變成 4 個 4-bits 的訊號，還有將 score_num, total_num, miss_num 分別轉變成 2 個 4-bits 的訊號，表示 BCD form 的十位和個位，將前述訊號放入 value0, value1, value2, value3，並用 state 做 select 決定放入的訊號，再把 value0~value3 接到 bcd_to_ssd 和 ssd_ctl。另外，clock_divisor_ssd 產生 clock 接到 ssd_ctl 作為 ctl_en 來控制顯示器亮燈位置，即可在七段顯示器的四個位置上看到數字或字母,且可透過 bottom(state 切換)改變顯示的數值。

* value module coding in topmodule 127~172

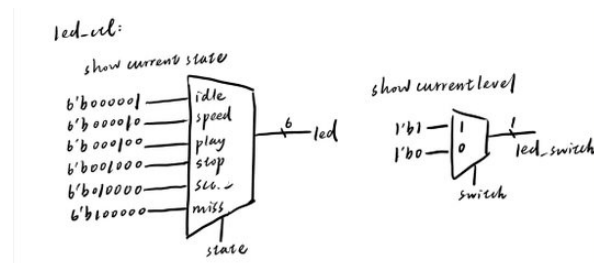
Switch—module: clock_divisor, clock_divisor_LFSR, game_minute, sound_top



功能：調控關卡難度(下扳:level1(slow), 上扳:level2(fast))。

製作：clock_divisor 中，做一個 mux 用 switch 做 select，設計 level2(fast)的 clk_22 的頻率為 level1(slow) 的 4 倍，clk_22 接到 clock_divisor_LFSR 和 game_minute，

使 LFSR 能因應 level 而改變產生數字的速度，game_minute 的結束時間也能隨 level 調整，而 sound_top 中的 freq_div 產生的頻率也由 switch 判斷，設計 level2(fast)的頻率為 level1(slow)的 2 倍，因此也能因 level 而改變播音速度

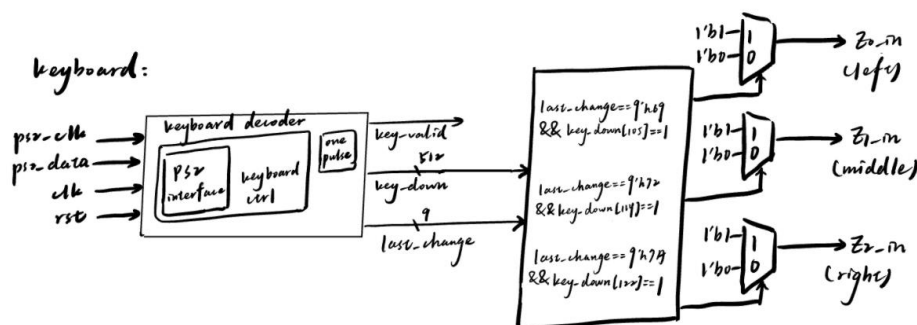


Led—module:led_ctl

功能：表示目前的 state, 6 個 state 分別以 6 顆 led 燈表示。另外，還有 1 顆 led 表示關卡難度，隨 switch 上下扳而亮暗。

製作：做一個 mux 用 state 做 select 並 assign switch_led == switch

keyboard—module:KeyboardDecoder



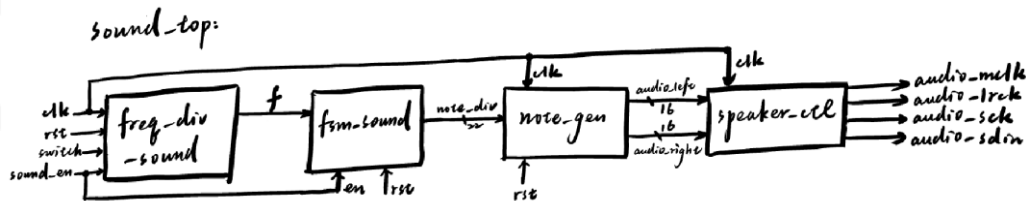
功能：

當殭屍由左跑道掉落到第一條紅線時，按下數字 1，殭屍即被殺死(消失)，由中跑道掉落到第一條紅線時，按下數字 2，殭屍消失，由右跑道掉落到第一條紅線時，按下數字 3，殭屍消失。

製作:(參照 lab8)

做 mux 判斷 KeyboardDecoder 接出的訊號 last_change(判斷按下哪一個鍵)和 key_down(判斷按鍵是否按下)，產生 z0_in, z1_in, z2_in，經 debounce, onepulse 處理接到 zombie_select 使殭屍消失。

Music—module: music_top

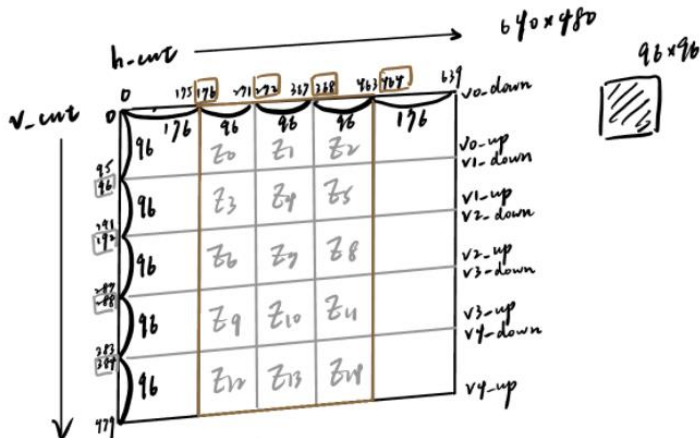


功能:遊戲開始時，音樂播出。

製作:(參照 lab7)

設計 fsm 讓 1 個 state 播出 1 個音(輸出 1 個 note_div 數值)，可藉由調整 state 個數或 fsm 的 clock 改變音樂播放速度，另外，2 句歌詞間可加入 1 個 note_div=0 的 state，聽起來就會有中斷感。

vga—module: clock_divisor, vga_controller, mem_addr_gen, blk_mem_gen, RGB_ctl



功能: 使螢幕顯示 4 張圖片(起始畫面、關卡難度說明、無殭屍(全黑)、有殭屍)並在遊戲進行時能使分割的區塊移動。

製作:(參照 lab9)

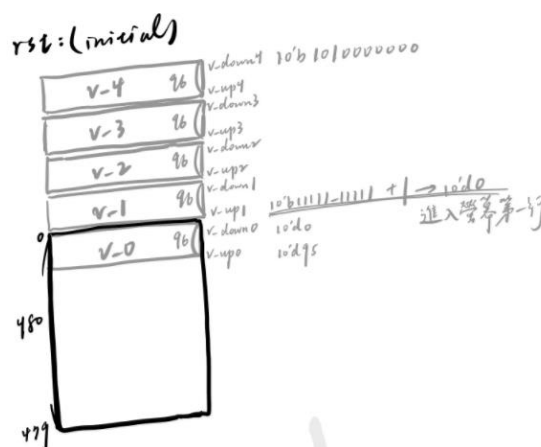
起始畫面與關卡難度說明分別為兩張 100*150 像素大小的照片，會分別在起始畫面與說明 level 的 state 顯示，並且降低解析度將照片放大為 200*300 的大小。將螢幕按照上圖分割，設定每張照片為 96*96 像素大小，這裡用到的是有殭屍及無殭屍兩張圖，需要判斷上圖 Z0~Z14 位置要放的是有殭屍或是無殭屍的圖。圖片是從網路上找再用小畫家做一些調整後匯出，再用 pictran 轉換成 96*96 的 coe 檔。將 4 個的 coe 檔的內容全部都集中到一個 coe 檔，產生一個 memory IP。

接著，計算螢幕分割處和圖片分割處的 h_cnt, v_cnt 值，用 mux 選擇需要顯示的螢幕區域，對應到要顯示圖片位置，將計算好的 pixel address 輸入 memory ip，即可使螢幕顯示圖片。

以下說明 **mem_addr_gen module**:

計算圖片顯示的界線(h_cnt, v_cnt 臨界值)，依照上圖的版面配置，當 h_cnt : 0~175 或 434~639 不顯示圖片(黑屏)， h_cnt : 176~271 為左跑道位置, h_cnt : 為中跑道位置, h_cnt : 368~463 為右跑道位置，再將每條跑道依照 v_cnt 每經過 96 分割成 5 塊區域，分別為 $z0 \sim z14$ ，因為殭屍會往下移動，所以下面會計算每一行殭屍的起始點位置，運用 selector 去選擇對應的各區快要顯示哪張圖片，並找到權重值($z=0$ 表示有殭屍或 $z=1$ 表示沒殭屍，此數值由 `zombie_select` 產生)，即可得出各位置的 `pixel_address`。

Pixel address 計算:



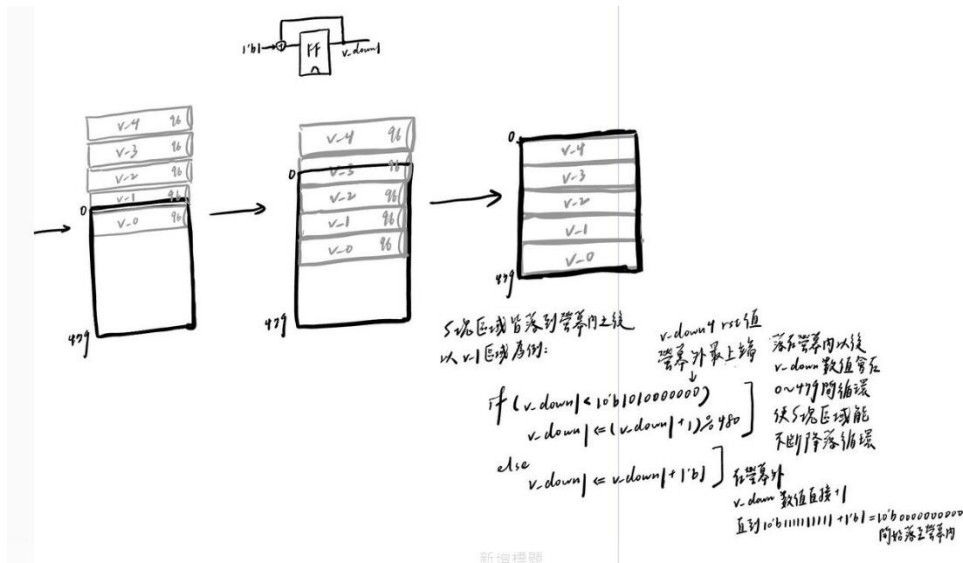
Verilog coding:

```
// divided into 5 area
// v_down: the top line of an area
// v_up: the bottom line of an area
always @(posedge clk or posedge rst) begin
  if (rst) begin
    v_down0 <= 10'd0;
    v_up0 <= 10'd95;
    v_down1 <= 10'b1101000000;
    v_up1 <= 10'b1111111111; // 10'b1+1'b1 = 10'd0 (being sliding down in screen)
    v_down2 <= 10'b1101000000;
    v_up2 <= 10'b1100111111;
    v_down3 <= 10'b1011100000;
    v_up3 <= 10'b1100111111;
    v_down4 <= 10'b1010000000;
    v_up4 <= 10'b1011111111;
  end
  else begin
    v_down0 <= (v_down0 + 1)%480;
    v_up0 <= (v_up0 + 1)%480;
    if (v_down1 < 10'b1010000000 && v_up1 < 10'b1010000000) begin // if all 5 area slide into the screen
      v_down1 <= (v_down1 + 1)%480; // v_up and v_down slide down 1 row when clk trigger
      v_up1 <= (v_up1 + 1)%480; // and will be the cycle in the region 0-479
    end
    else begin
      v_down1 <= v_down1 + 1; // if the area outside the screen
      v_up1 <= v_up1 + 1; // plus 1 until carry into 10'd0 (slide into the screen)
    end
  end
end
```

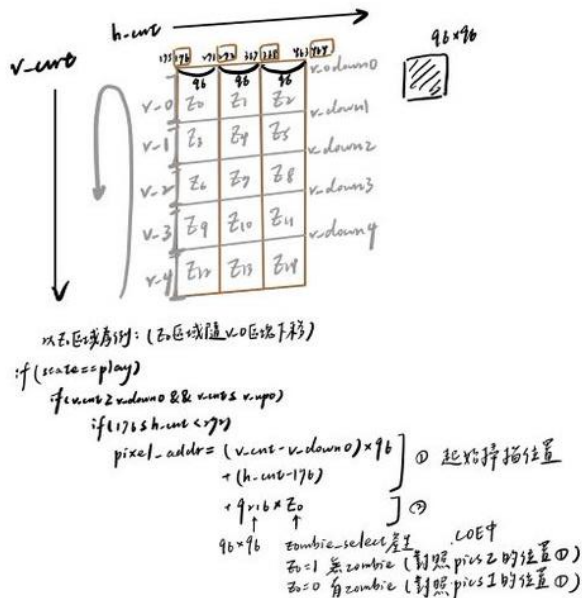
做數個 flipflop 來控制同一行的殭屍上下界位置:

v_0, v_1, v_2, v_3, v_4 區域如圖所示，在遊戲起始時僅 v_0 在螢幕內，另外 4 區域在螢幕以上，將各區域 v_down, v_up 放入 rst 值，接著 v_down, v_up 隨 `clk trigger + 1'b1`，移至下一行，使各區域不斷下滑，直到各區域落入螢幕內之後就能以 `trigger + 1'b1` 後計算 480 的餘數的方式使各區域在螢幕內循環。

以下用 v_down1 為例說明：



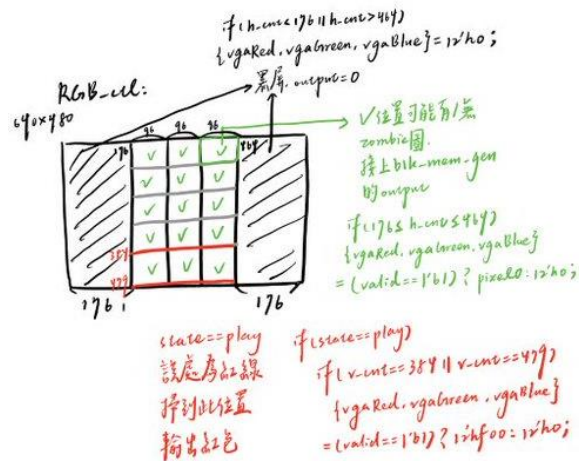
接著處理 h_cnt 的 select 部分，以 $z0$ 區域為例說明：



Verilog coding:

```
always@*
  if (state == idle) begin
    if (h_cnt > 170 && h_cnt < 470) // pics displayed on h_cnt:170 ~ 470 ,v_cnt:140-340
      if (v_cnt >= 140 && v_cnt <= 340) pixel_addr = ((h_cnt - 170)>>1)+150*((v_cnt - 140)>>1) + 18432; // 300 * 200 --> 150 * 100
    end
  else if (state == speed) begin
    if (h_cnt > 170 && h_cnt < 470) // pics displayed on h_cnt:170 ~ 470 ,v_cnt:140-340
      if (v_cnt >= 140 && v_cnt <= 340) pixel_addr = ((h_cnt - 170)>>1)+150*((v_cnt - 140)>>1) + 33432; // 300 * 200 --> 150 * 100
    end
  else if (state == play) begin
    if (v_cnt >= v_down0 && v_cnt <= v_up0)
      if (h_cnt < 176)
        pixel_addr = 0;
      else if (h_cnt < 272)
        pixel_addr = (v_cnt - v_down0) * 96 + h_cnt - 176 + 9216 * z0; // 96 * 96 = 9216(pics pixel)
      else if (h_cnt < 368)
        pixel_addr = (v_cnt - v_down0) * 96 + h_cnt - 272 + 9216 * z1; // if z = 1, pixel_pos + 9216, 2nd pics in .coe(black screen)
      else if (h_cnt < 464)
        pixel_addr = (v_cnt - v_down0) * 96 + h_cnt - 368 + 9216 * z2; // else z = 0, pixel_pos + 0, 1st pics in .coe(zombie)
      else pixel_addr = 0;
    end
  end
```


另外，說明 RGB_ctl_module:



功能:控制不同欄與列的 RGB 值

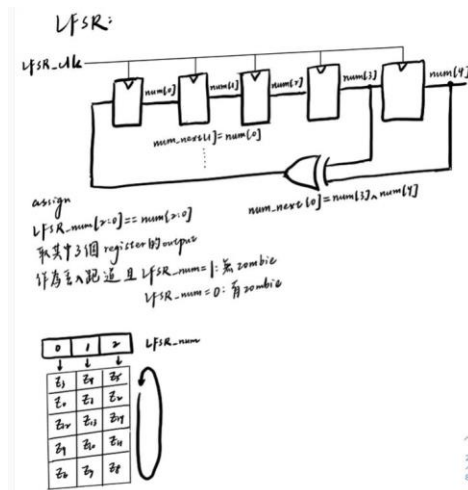
製作:同 mem_addr_gen 算出各區域 h_cnt, v_cnt 分界，做多個 mux 選擇該區域的 RGB 填入值，這裡除了殭屍的圖片部分會直接取 blk_mem_gen 輸出的 pixel 外，還設置了判斷在 v_cnt = 384、479 的位置要顯示紅色。

*RGB_ctl module coding in topmodule 234~258

*vga_controller module the same content as lab09 source code

接著說明 block diagram 中的幾個 module:

LFSR:



功能:

產生隨機數字，將數字放入跑道，決定該跑道是否產生殭屍 (LFSR_num = 1: 有殭屍將掉落，LFSR_num = 0: 無)

製作:用 5 個 register 產生隨機數字，取前面 3 個 register 的 output 當作

LFSR_num，同時丟入 3 個跑道。LFSR_num 接到 zombie_select 處理跑道產生殭屍的部分

**用多個 register 取其中幾位可延長隨機數字的規律，使其較不易被觀察出

Game_minuite:

功能:停止產生殭屍和控制遊戲進行時間

製作:接上 clock_divisor 產生的 clk_22(level2(fast)的 clk_22 較快)再經過 clock_divisor_LFSR 產生的 clk_LFSR, 依照遊戲的不同速度設定停止產生殭屍的時間(利用變數 zombie_minute 來設定要出幾輪殭屍以控制時間), 用 clk_LFSR(對應到殭屍出現的速度)數到 zombie_minute, 此時停止產生殭屍 (no_zombie = 1), 再用同一個 clock 多數 5(延遲時間), 此時遊戲真正停止 (stop_tag_onepulse = 1), 切換到遊戲停止畫面

**數到 zombie_minute 不再產生殭屍, 但接著會有一小段延遲時間, 在掉落中的殭屍仍會繼續掉落至紅線, 以確保所有殭屍掉落完, 且能被打到, 過完延遲時間後, 遊戲真正停止, 7-seg-display 顯示 win or lose, 音樂停播, 畫面為黑屏。

Zombie_select:

功能:產生殭屍、讓殭屍消失並計算 score_num, total_num, miss_num

製作: 1.殭屍產生與消失(以 z0 區為例):

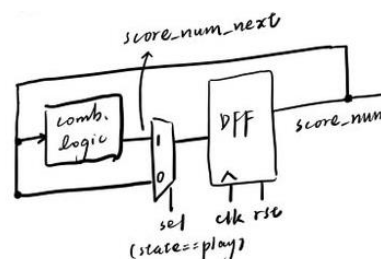
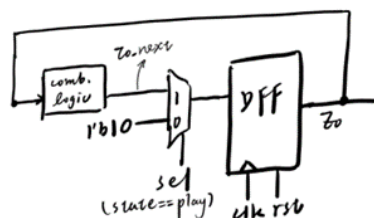
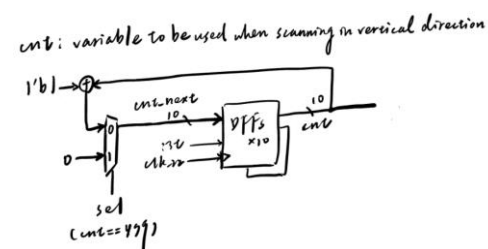
cnt:掃描時用於表示鉛直方向位置的變數, 由數個 flipflop 產生

z0_tag: 用於標記殭屍掉落過程中 miss_num 是否已經被加過(每一個區塊在掉落過程中最多只會被誤按一次(掉落在紅線間的區域時, 沒有殭屍在此區域, 仍按下按鍵, 才算誤按))

z0: 用於表示殭屍的出現與消失(z0=1'b1 無殭屍, z0=1'b0 有殭屍)

先處理 cnt, z0_tag, z0 的 sequential block:

做 flipflop 存數值並用 mux 選擇是否需經 combinational block 運算(參考 lab3, 4)



* z0_tag, z0 的 sequential block 結構相同; score_num, total_num, miss_num 的 sequential block 結構相同

combinational block 需要計算 score_num, total_num, miss_num:

計算 total_num:

Verilog coding:

```
always @(*) begin
    if (state == idle) begin
        total_next = 10'd0;
    end
    else if (state == play) begin
        if (cnt == 10'd479 || cnt == 10'd495 || cnt == 10'd491 || cnt == 10'd287 || cnt == 10'd383) begin //v_cnt of line to sepearte each area
            if (no_zombie) total_next = total; // no_zombie = 1: no zombie generate
            else total_next = total + {9'd0, ~LFSRnum[0]} + {9'd0, ~LFSRnum[1]} + {9'd0, ~LFSRnum[2]}; // LFSRnum = 0: exist zombie
            end else total_next = total; // when counting zombie num, need an invertor
        end else total_next = total;
    end
end
```

說明:在遊戲進行，且利用 5 區塊的每一塊要重新滑入螢幕內時設定給該區塊的 LFSR_num，計算 total_num。

相加時，LFSR_num 前需加 invertor，因 LFSR_num = 1'b0 表示有殭屍，所以在計算殭屍數值時須 invert 後再相加。

計算 score_num:

Verilog coding:

```
always @(*) begin
    score_num_next = score_num;
    if (cnt == 10'd479) begin // v_0 area is going to slide from the top
        if (no_zombie) begin // no_zombie = 1: no zombie, all z = 1
            z0_next = 1'b1;
            z1_next = 1'b1;
            z2_next = 1'b1;
        end else begin // assign LFSR_num to z0, z1, z2 to decide whether zombie exist at the top of screen
            z0_next = LFSRnum[0];
            z1_next = LFSRnum[1];
            z2_next = LFSRnum[2];
        end
    end
    else if ((v_down0 >= line_down && v_down0 <= line_up) || (v_up0 <= line_up && v_up0 >= line_down)) begin // in area between red lines
        if (z0_in) begin // zombie exists, press keyboard to kill zombie, then score + 1 and zombie disappear
            if (z0 == 1'b0) score_num_next = score_num + 1'b1;
            z0_next = 1'b1;
        end else begin
            z0_next = z0; // if not press keyboard, zombie still exist
        end
        if (z1_in) begin
            if (z1 == 1'b0) score_num_next = score_num + 1'b1;
            z1_next = 1'b1;
        end else begin
            z1_next = z1;
        end
        if (z2_in) begin
            if (z2 == 1'b0) score_num_next = score_num + 1'b1;
            z2_next = 1'b1;
        end else begin
            z2_next = z2;
        end
    end
    else begin
        z0_next = z0;
        z1_next = z1;
        z2_next = z2;
    end
end
```

以 miss_num 圖示情況說明(z0 落至最底端，即將從頂端滑下)

先放入 LFSR_num 在跑道頂端，依照此數字決定跑道頂端是否將出現殭屍。

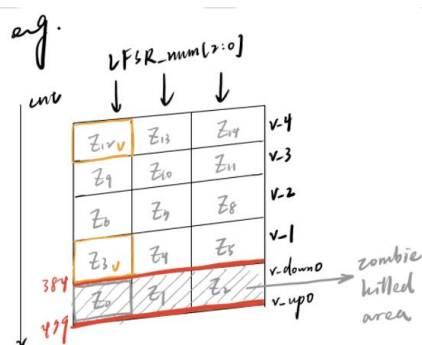
當區塊從頂端下滑到 2 條紅線間(灰色斜線算分區)時，計算得分，

若按下按鍵(z0_in = 1'b1)，且此刻有殭屍落至算分區(cnt: 384~ 479)，

殭屍會消失(被殺死)(z0 = 1'b1)，score_num + 1。

計算 miss_num:

以圖示狀況說明:(v_0 落在兩條紅線間)























































Verilog coding:

```
// z_tag used to mark whether miss_num had been added when player making mistake
// comb. logic of miss_num
always @(*) begin
    miss_next = miss_num;
    if (cut == 10'd79) begin
        if (no_zombie) begin // stop to generate zombie, no miss_num to be add
            z0_tag_next = 1'b0;
            z1_tag_next = 1'b0;
            z2_tag_next = 1'b0;
        end else begin // though z0 slides down to the bottom, but the game continues.
            z0_tag_next = 1'b1; // when it returns to the top, it is a new turn to count miss_num
            z1_tag_next = 1'b1;
            z2_tag_next = 1'b1;
        end
    end else begin
        if ((v_down0 >= line_down && v_down0 <= line_up) || (v_up0 <= line_up && v_up0 >= line_down)) begin // in the area between two red lines
            if (z0_in) begin // press 1 on keyboard (wrong press)
                if (z0 == 1'b1 && z0_tag == 1'b1 && ((z12 == 1'b1 && (v_down4 >= line_down && v_down4 <= line_up)) || (z3 == 1'b1 && (v_up1 <= line_up && v_up1 >= line_down)))) miss_next = miss_num + 1'b1;
                z0_tag_next = 1'b0; // already add miss_num, miss can't be count again // condition to count miss: z0 = 1'b1 && z12 = 1'b1 && z3 = 1'b1; these three area are no zombie, but player press,
                z0_tag == 1'b1: miss_num has't been added
            end else begin
                z0_tag_next = z0_tag;
            end
        end
        if (z1_in) begin
            if (z1 == 1'b1 && z1_tag == 1'b1 && ((z13 == 1'b1 && (v_down4 >= line_down && v_down4 <= line_up)) || (z4 == 1'b1 && (v_up1 <= line_up && v_up1 >= line_down)))) miss_next = miss_num + 1'b1;
            z1_tag_next = 1'b0;
        end else begin
            z1_tag_next = z1_tag;
        end
        if (z2_in) begin
            if (z2 == 1'b1 && z2_tag == 1'b1 && ((z14 == 1'b1 && (v_down4 >= line_down && v_down4 <= line_up)) || (z5 == 1'b1 && (v_up1 <= line_up && v_up1 >= line_down)))) miss_next = miss_num + 1'b1;
            z2_tag_next = 1'b0;
        end else begin
            z2_tag_next = z2_tag;
        end
    end else begin
        z0_tag_next = z0_tag;
        z1_tag_next = z1_tag;
        z2_tag_next = z2_tag;
    end
end
endmodule
```

當區塊從頂端下滑到 2 條紅線間(灰色斜線算分區)時，計算誤按次數。

若按下按鍵(z0_in = 1'b1)，但是 z0 區沒有殭屍，z3(z0 上方區)、z12(z0 下方區)也沒有殭屍，及代表誤按，miss_num + 1 且 z0_tag_next == 1'b0(加過誤按次數了)。誤按除了必須檢查算分區快以外，正上方區和正下方區也需檢查，避免其實是錯按但是被歸入殺死了上一格或下一格的殭屍的情形。另外，若 no_zombie = 1，代表不再產生殭屍，不須計算誤按，z_tag_next = 1'b0。

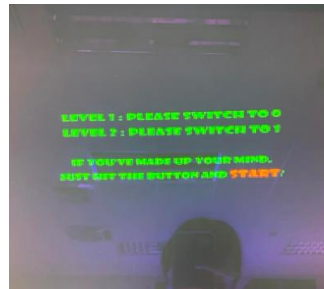
I/O pins:

▼  vgaBlue (4)	OUT			▼  an (4)	OUT		
 vgaBlue[3]	OUT		J18	▼  an[3]	OUT		W4
 vgaBlue[2]	OUT		K18	▼  an[2]	OUT		V4
 vgaBlue[1]	OUT		L18	▼  an[1]	OUT		U4
 vgaBlue[0]	OUT		N18	▼  an[0]	OUT		U2
▼  vgaGreen (4)	OUT			▼  led (6)	OUT		
 vgaGreen[3]	OUT		D17	▼  led[5]	OUT		U16
 vgaGreen[2]	OUT		G17	▼  led[4]	OUT		E19
 vgaGreen[1]	OUT		H17	▼  led[3]	OUT		U19
 vgaGreen[0]	OUT		J17	▼  led[2]	OUT		V19
▼  vgaRed (4)	OUT			▼  led[1]	OUT		W18
 vgaRed[3]	OUT		N19	▼  led[0]	OUT		U15
 vgaRed[2]	OUT		J19	▼  ssd (8)	OUT		
 vgaRed[1]	OUT		H19	▼  ssd[7]	OUT		W7
 vgaRed[0]	OUT		G19	▼  ssd[6]	OUT		W6
▼  Scalar ports (15)				▼  ssd[5]	OUT		U8
 audio_lrc	OUT		A16	▼  ssd[4]	OUT		V8
 audio_mclk	OUT		A14	▼  ssd[3]	OUT		U5
 audio_sck	OUT		B15	▼  ssd[2]	OUT		V5
 audio_sdin	OUT		B16	▼  ssd[1]	OUT		U7
 btn	IN		T18	▼  ssd[0]	OUT		V7
 btn_miss	IN		W19				
 btn_score	IN		U18				
 clk	IN		W5				
 hsync	OUT		P19				
 PS2_CLK	INOUT		C17				
 PS2_DATA	INOUT		B17				
 rst	IN		R2				
 switch	IN		V17				
 switch_led	OUT		L1				
 vsync	OUT		R19				

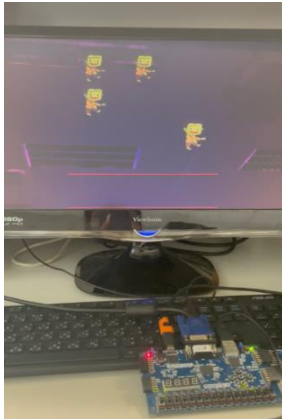
Pics:



起始畫面



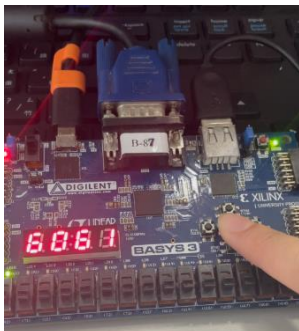
level 說明畫面



遊戲進行中畫面



遊戲停止，顯示結果



顯示 score/ total



顯示 miss(按錯次數)

碰到的問題及解決辦法:

一開始研究很久的是 VGA 的部分，因為不知道到底怎麼控制好圖片，讓他可以在我要的範圍內移動，而且在不同區域內還要顯示不同張照片。

解決方法是我做了一次 lab9，熟悉一下怎麼控制位置，但這裡主要學會的是在起始畫面與選擇速度的畫面時我為了減少儲存在 ROM 的資料量，所以只能存較小的照片，並在實際顯示以降低解析度來放大圖片。操作遊戲進行中的畫面更難，以 lab9 學到的內容經過不斷的修改、嘗試，調整每一個參數記錄其造成的改變才更加知道怎麼控制，研究出我要的顯示樣子。

Discussion:

本次 project 最有挑戰性的地方是顯示圖片以及圖片的移動，圖片顯示的部分必須算清楚各個分割區域的分界值、決定好螢幕的版面配置、調整圖片像素、用 pictans 產生 coe 檔、匯入 ip、設定 pixel_address 值以及決定 pixel 的 RGB 值等等，剛開始連用 pictrans 產生 coe 檔都會出現錯誤，後來經過不斷嘗試、練習作業，才比較熟悉操作流程。體悟到每個 lab 都要實際做過真的很重要，只有課堂上的講解常常以為懂了，但操作起來才發現根本就誤會意思，而且從簡單的題目練起可以更容易知道在做甚麼，像是沒先做 lab9 就開始做 project 時真的無從下手。

Conclusion:

我從本次 lab 學習到:

1. vga 使用、coding、運作原理
2. bottom, switch, keyboard 的運用和 coding
3. led, 7-seg-display, speaker 的運用和 coding
4. LFSR 運作原理及 coding
5. fsm 設計、block diagram 設計

在 final project 真的用到了每一個 lab 的內容，沒想到所有內容最後能組合成這麼可愛的遊戲~最後，非常謝謝老師和辛苦的助教!

References:

Lab1~Lab9 和邏設實驗講義