Final Project《Tetris 俄羅斯方塊》：Report

一、簡介

實作經典遊戲：Tetris 俄羅斯方塊。

將遊戲畫面呈現在螢幕上，透過鍵盤控制遊戲操作。

包含俄羅斯方塊有的掉落、旋轉、消除、保留、預覽、直接落下與簡易計分等功能。

二、Project 設計

- **遊戲主程式**

  依照 module 的功能與分工，將所有遊戲主程式 module 分成三大類：

  1. Top module

     控制遊戲當前要執行的操作，並將遊戲呈現所需要的資訊（遊戲版面）輸出給螢幕顯示的 module。

     屬於本類型的 module：

     - GameEngine

  2. Transition module

     根據 Top module 狀態（欲執行的操作），調用 Function module 來更新遊戲資訊。主要針對無法直接調用 Function module 處理的更新。

     屬於本類型的 module：

     - generate_piece_TRAN
     - get_input_TRAN
     - hold_TRAN
     - clear_line_TRAN

  3. Function module：

     實現遊戲執行所需的核心功能。

     屬於本類型的 module：

     - board_controller
     - falling_piece_update
     - random_tetris

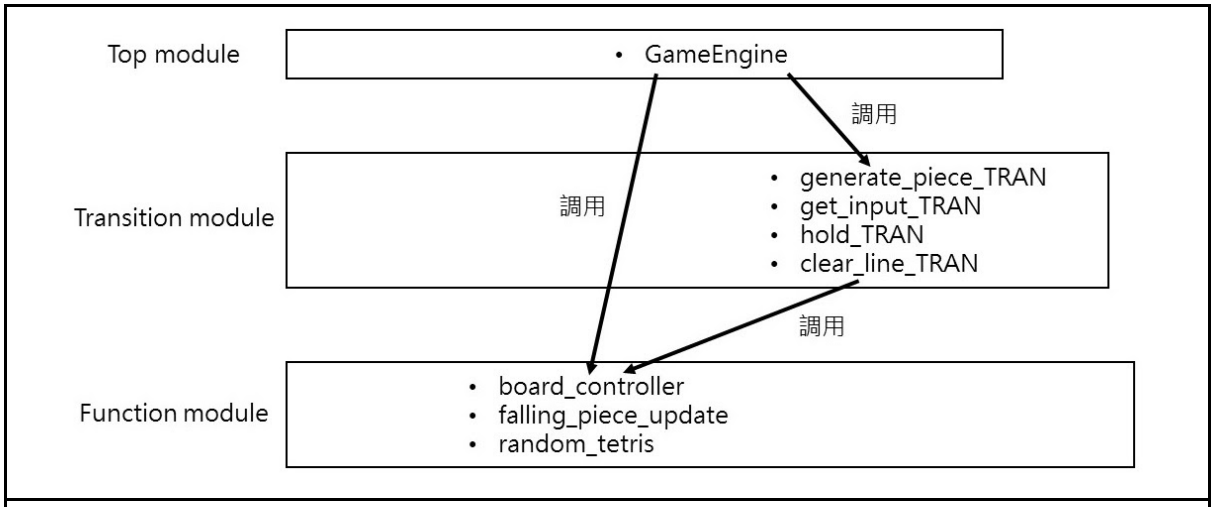

圖 2-1：各 module 間的分工

下面依照分類（自頂向下）逐個介紹各 module。

【Top module】GameEngine

——調用其他 module 來運行遊戲，並將版面、分數、掉落預覽輸出

作為遊戲主程式的 Top module，**其 STG 同時亦為遊戲主程式的 STG** 。

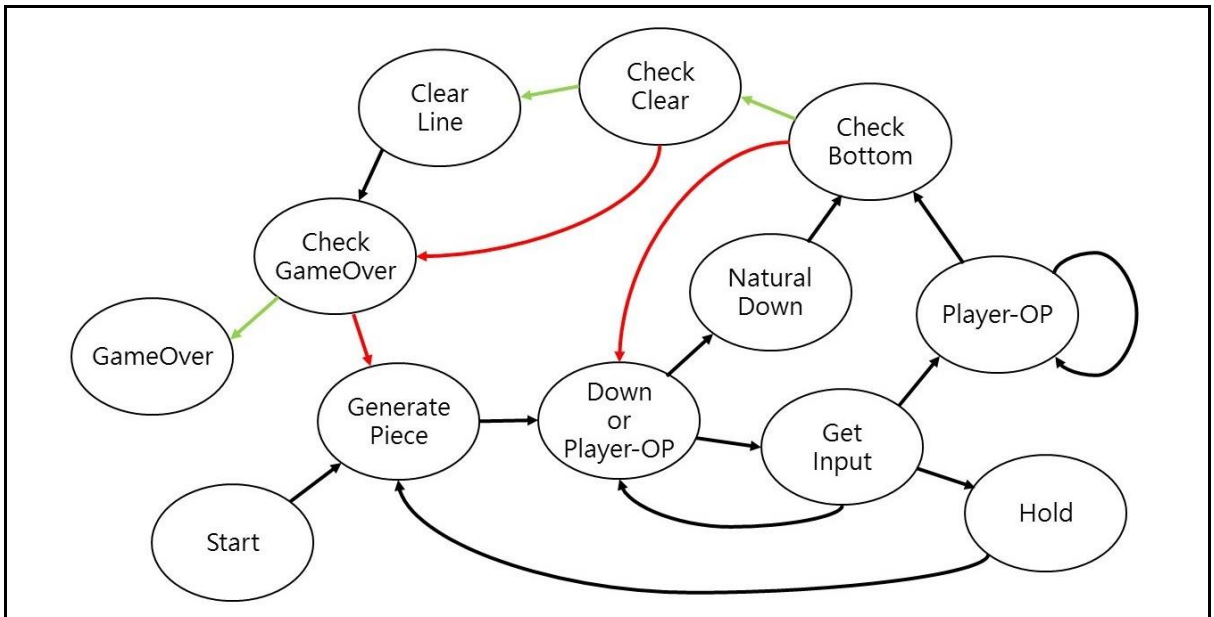

圖 2-2：GameEngine STG

GameEngine 維護遊戲所需的幾乎所有資訊，主要資訊有：

- **state**
  當前 GameEngine STG 的狀態
- **falling_piece_i 與 falling_piece_j**
  當前掉落方塊位置的座標

- **falling_piece_type**
  當前掉落方塊的類型（Ex：T 方塊、L 方塊等）
- **falling_piece_angle**
  當前掉落方塊的角度
- **op_type**
  當前欲對掉落方塊執行的操作類型
- **hold_valid**
  當前可不可以 hold（ hold 一次後要等下個方塊生成才能再用）
- **drop_cnt**
  自然掉落計時器
- **drop_cd**
  自然掉落間隔（drop_cnt = drop_cd 時使掉落方塊下移一格）
- **score**
  當前玩家的分數

特別地，**遊戲版面由 board_controller module 維護**。

GameEngine 各狀態與其對應執行的操作與狀態轉移：

狀態	執行的操作
Start	GameEngine 初始狀態，reset = 1 時狀態轉移至此。 下個狀態為 Generate Piece。
Generate Piece	**調用 generate_piece_TRAN 以生成新的掉落方塊。** 詳細說明請參考下面 generate_piece_TRAN 部分。 下個狀態為 Down or Player-OP。
Down or Player-OP	STG 的分歧點，判斷要落下或是執行玩家的操作。 若 drop_cnt = drop_cd，狀態轉移至 Natural Down，否則轉移到 Get Input。
Natural Down	將 op_type 設為 `DOWN_MOVE_OP，並透過 falling_piece_update module 使掉落方塊下移一格 下個狀態為 Check Bottom。

<table>
<tr><td>Get Input</td><td>調用 get_input_TRAN 獲取輸入並更新 op_type。
詳細說明請參考下面 get_input_TRAN 部分。

根據 op_type，轉移至不同狀態：

<table><tr><td>op_type 值</td><td>下個狀態</td></tr><tr><td>`NULL_OP</td><td>Drop or Player-OP</td></tr><tr><td>`HOLD_OP</td><td>Hold</td></tr><tr><td>其他</td><td>Player-OP</td></tr></table></td></tr>
<tr><td>Player-OP</td><td>透過 falling_piece_update module 來執行使用者輸入的操作。
若操作為落至底部，則重複此狀態直到到底。

下個狀態為 CheckBottom。</td></tr>
<tr><td>Hold</td><td>調用 hold_TRAN 來 hold 住當前的掉落方塊。
詳細說明請參考下面 hold_TRAN 部分。
將 hold_valid 拉成 0。

下個狀態為 Generate Piece。</td></tr>
<tr><td>Check Bottom</td><td>透過 falling_piece_update 「模擬」方塊下移一格，
並透過操作合法性來得知方塊是否已至底部。

方塊以至底部時下個狀態為 Check Clear，否則下個狀態為 Down or Player-OP。</td></tr>
<tr><td>Check Clear</td><td>透過 board_controller 維護的 line_full 來判斷是否有消除行。有消除時，更新分數（score）。

有消除行時下個狀態轉移為 Clear Line，否則下個狀態為 Check GameOver。</td></tr>
<tr><td>Clear Line</td><td>調用 clear_line_TRAN 清除該消除的行。
詳細說明請參考下面 clear_line_TRAN 部分。

下個狀態為 Check GameOver。</td></tr>
</table>

Check GameOver	透過 generate_piece_TRAN 「模擬」生成新的方塊，並透過操作合法性得知遊戲是否結束（即生成的方塊若與現有方塊重合時，遊戲即結束）；此機制保證了 Generate Piece 狀態時必可生成方塊。 遊戲結束時下個狀態為 GameOver，否則下個狀態為 Generate Piece。
GameOver	遊戲結束，凍結畫面。 將遊戲結束訊號傳給整個系統的 Top module。 沒有下個狀態（僅能透過 reset 重置狀態與遊戲）。

【Transition module】generate_piece_TRAN

──生成新的掉落方塊

啟用（enable = 1）後，會將新生成的方塊四格座標逐個輸出給 GameEngine（每個 Clock Period 輸出一格），之後再透過 board_controller 來更新版面（寫入）。四格都輸出後，再將 done 拉成 1。

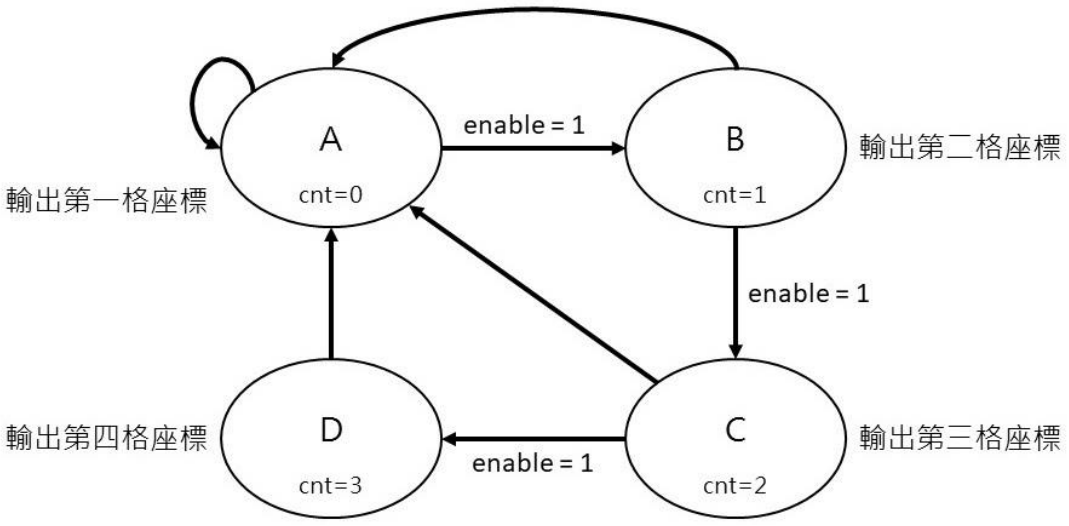

圖 2-3：generate_piece_TRAN STG

新生成的掉落方塊，透過 random_tetris module 取得，並在生成後拉起對應的訊號（next_took），驅動 random_tetris 更新下個方塊。

另外由於 " hold " 時，GameEngine 會將其狀態轉移至 Generate Piece ，但除了遊戲的首次 hold 以外，透過這樣方法轉移到的 Generate Piece 不會生成新的方塊，因次加入了非首次 hold 的判斷來正確維護 next_took。

最後，遊戲預覽下次的方塊是透過 generate_piece_TRAN module 將 random_tetris 的下個方塊傳給 GameEngine。

【Transition module】get_input_TRAN

——處理玩家的輸入及決定接下來執行的操作類型

紀錄玩家的輸入，並更新 GameEngine 的 op_type 以執行操作。

在不能 " hold " 時（hold_valid = 0）會忽略玩家的 hold 操作。

op_type 類型與對應的（玩家）操作：

op_type 值	對應的操作
`NULL_OP	無操作 / 玩家未輸入任何操作
`HOLD_OP	hold 住掉落的方塊
`DOWN_MOVE_OP	另掉落的方塊直接降到底部
`SPIN_OP	旋轉掉落方塊的角度
`RIGHT_MOVE_OP	將掉落方塊右移一格
`LEFT_MOVE_OP	將掉落方塊左移一格

【Transition module】hold_TRAN

——處理玩家的 " hold " 操作

啟用（enable = 1）後，會將當前掉落方塊四格座標逐個輸出給 GameEngine（每個 Clock Period 輸出一格），之後再透過 board_controller 來更新版面（移除）。四格都輸出後，再將 done 拉成 1。

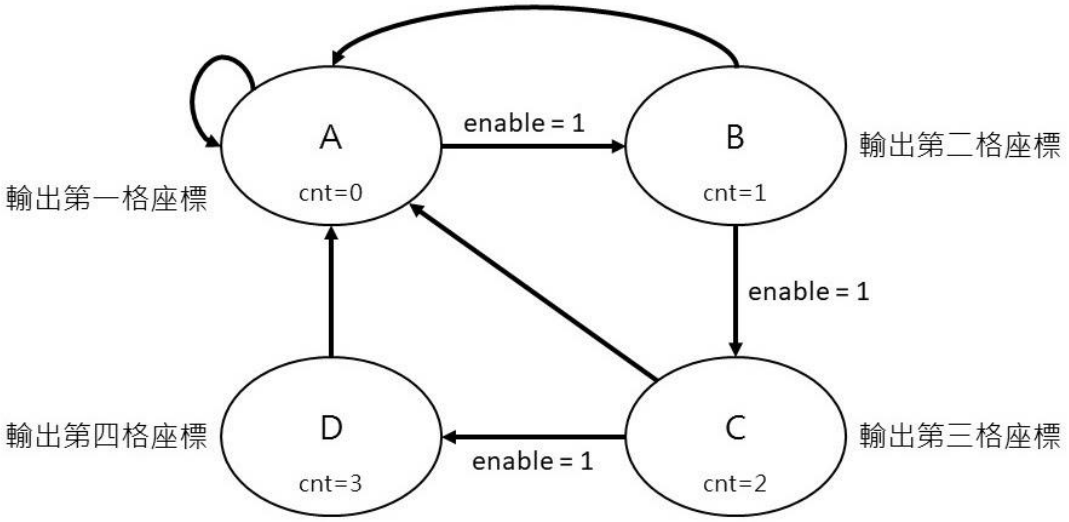

圖 2-4：hold_TRAN STG

【Transition module】clear_line_TRAN

──消除應被消除的行

啟用（enable = 1）後，**重複地**將最下面應消除行的十格座標逐個輸出給 GameEngine（每個 Clock Period 輸出一格），之後再透過 board_controller 來更新版面（移除），接著將該行以上的版面往下平移一格（一樣透過輸出座標給 GameEngine，再透過 board_controller 平移）；一直重複消除、平移直到 line_full = 0 為止，最後將 clear_line_done 拉成 1。

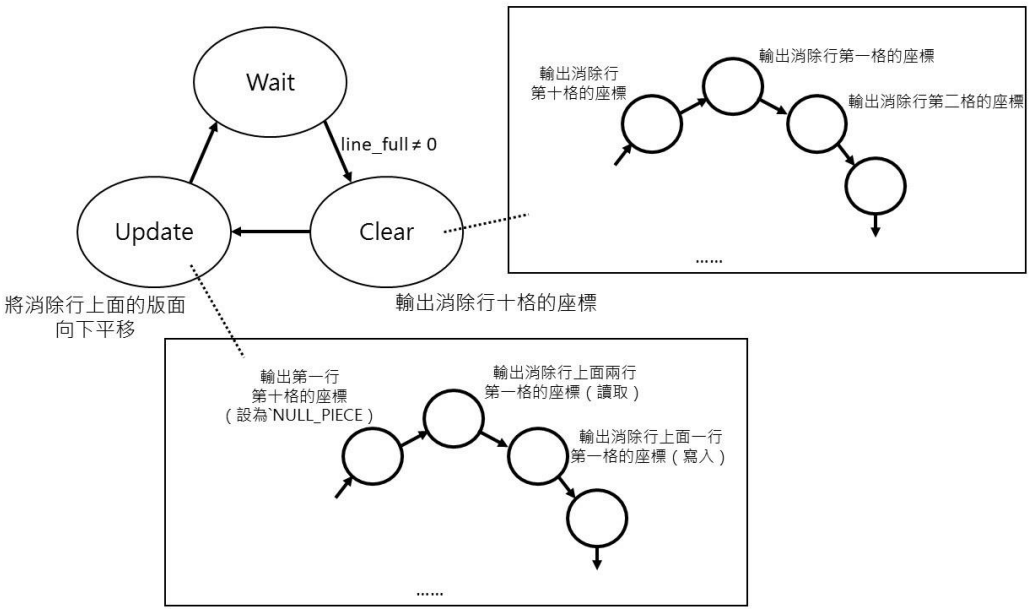

圖 2-5：clear_line_TRAN STG

【Function module】board_controller

——更新遊戲版面與提供遊戲版面的資訊

使用 600-bit 的 register 紀錄 20 x 10 的遊戲版面，每格使用 3-bit 紀錄該格的方塊類型（共 7 種方塊 + 無方塊 = 8 = 23 種）。

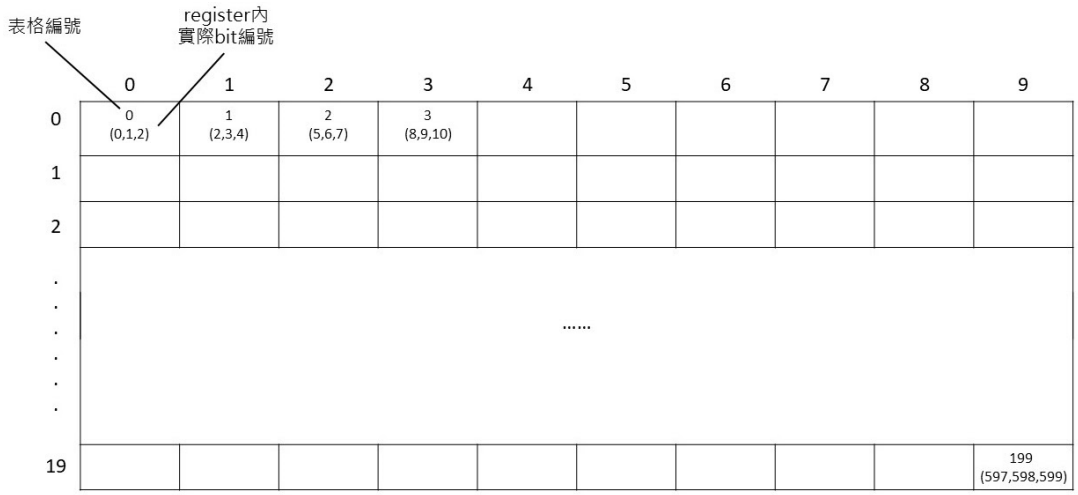

圖 2-6：遊戲版面座標編碼規則

在 wrtie_enable = 1 時，可更新座標（pos_i, pos_j）該格的方塊類型（每個 Clock Period 可更新一格）；在 write_enable = 0 時，則會輸出座標（pos_i, pos_j）該格的方塊類型。

另外維護各行（line）的方塊數量 piece_number_in_line（在 write_enable = 1 時會更新），並透過此維護「滿」了的行 line_full，並輸出給 GameEngine。clear_line_TRAN 便是透過 line_full 來消除行。

而 GameEngine 於狀態 Check Bottom 與 Check GameOver 的「模擬」行為，便是透過讀取（write_enable = 0）模擬時各格的方塊狀態來判斷操作的合法性。

為了在螢幕上顯示遊戲版面，還會額外根據螢幕顯示需求給定對應座標的方塊類型。

【Function module】falling_piece_update

——更新掉落方塊

當要更新掉落方塊時（op_type ≠ `NULL_OP），先判斷該種操作的合法性（透過 board_controller module，類似 GameEngine 的 Check Bottom 與 Check GameOver 的「模擬」）；若合法，則再執行更新操作（一樣透過 board_controller）：將當前掉落方塊的四格移除後（設為`NULL_PIECE），再於新的四格寫入原本的方塊類型。

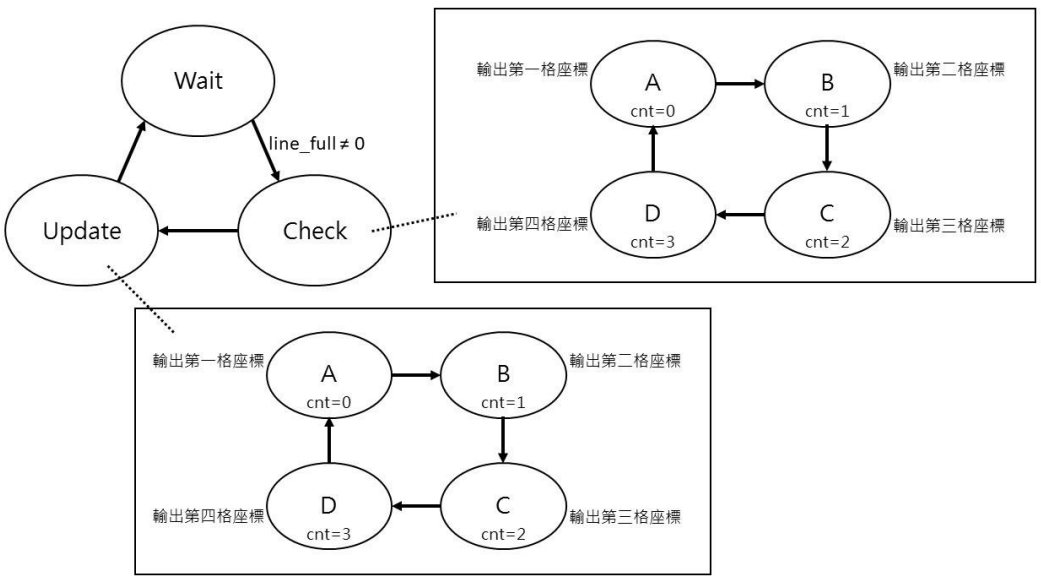

圖 2-7：falling_piece_update STG

寫入新的四格方塊時，需考量：

1. 執行的操作（下移一格、旋轉、左移一格或右移一格）
2. 掉落方塊的類型
3. 掉落方塊的角度

故共有 474=112 種可能，且掉落方塊有四格要處理。

考量到維護難度與後續延展功能（Wall Kick）方便性，使用前面更新四格的 STG，再透過 C++ 生成組合電路的 verilog code。

```
12  vector<vector<pos> > Q = {{{0,1},{0,2},{1,1},{1,2}},{{0,1},{0,2},{1,1},{1,2}},{{0,1},{0,2},{1,1},{1,2}},{{0,1},{0,2},{1,1},{1,2}}};
13  vector<vector<pos> > S = {{{0,1},{0,2},{1,0},{1,1}},{{0,1},{1,1},{1,2},{2,2}},{{1,1},{1,2},{2,0},{2,1}},{{0,0},{1,0},{1,1},{2,1}}};
14  vector<vector<pos> > Z = {{{0,0},{0,1},{1,1},{1,2}},{{0,2},{1,1},{1,2},{2,1}},{{1,0},{1,1},{2,1},{2,2}},{{0,1},{1,0},{1,1},{2,0}}};
15  vector<vector<pos> > T = {{{0,1},{1,0},{1,1},{1,2}},{{0,1},{1,1},{1,2},{2,1}},{{1,0},{1,1},{1,2},{2,1}},{{0,1},{1,0},{1,1},{2,1}}};
16  vector<vector<pos> > L = {{{0,2},{1,0},{1,1},{1,2}},{{0,1},{1,1},{2,1},{2,2}},{{1,0},{1,1},{1,2},{2,0}},{{0,0},{0,1},{1,1},{2,1}}};
17  vector<vector<pos> > J = {{{0,0},{1,0},{1,1},{1,2}},{{0,1},{0,2},{1,1},{2,1}},{{1,0},{1,1},{1,2},{2,2}},{{0,1},{1,1},{2,0},{2,1}}};
18  vector<vector<pos> > I = {{{1,0},{1,1},{1,2},{1,3}},{{0,2},{1,2},{2,2},{3,2}},{{2,0},{2,1},{2,2},{2,3}},{{0,1},{1,1},{2,1},{3,1}}};
19
20  vector<vector<vector<pos> > > tetris{Q,S,Z,T,L,J,I};
21  string name = "QSZTLJI";
```

```
114     cout << tab[st+0] << "UPDATE_STATE: begin" << endl;
115     cout << tab[st+1] << "case(op_type)" << endl;   // op_type
116     for(auto op_case : {make_tuple("`LEFT_MOVE_OP","+ 0","- 1",0),make_tuple("`RIGHT_MOVE_OP","+ 0","+ 1",0),
117                         make_tuple("`DOWN_MOVE_OP","+ 1","+ 0",0),make_tuple("`SPIN_OP","+ 0","+ 0",1)}) {
118         cout << tab[st+2] << get<0>(op_case) << ": begin" << endl;
119         cout << tab[st+3] << "case(falling_piece_type)" << endl;    // type of falling piece
120         for(int type_t = 0 ; type_t < 7 ; type_t++) {
121             cout << tab[st+4] << "`" << name[type_t] << "_PIECE: begin" << endl;
122             cout << tab[st+5] << "case(falling_piece_angle)" << endl;   // angle of falling_piece
123             for(int angle_t = 0 ; angle_t < 4; angle_t++) {
124                 cout << tab[st+6] << "2'd" << angle_t << ": begin" << endl;
125                 cout << tab[st+7] << "case(cnt)" << endl;   // cnt (check by order)
126                 for(int cnt_t = 0 ; cnt_t < 9 ; cnt_t++) {
127                     cout << tab[st+8] << "4'd" << cnt_t << ": begin" << endl;
128
129                     if(cnt_t < 4) {
130                         pos now = tetris[type_t][angle_t][cnt_t];
131                         cout << tab[st+9] << "pos_i = falling_piece_i + " << now.delta_i << ";" << endl;
132                         cout << tab[st+9] << "pos_j = falling_piece_j + " << now.delta_j << ";" << endl;
133                         cout << tab[st+9] << "write_enable = 1'b1;" << endl;
134                         cout << tab[st+9] << "new_piece_type = `NULL_PIECE;" << endl;
135                         cout << tab[st+9] << "next_cnt = cnt + 1'b1;" << endl;
136                     }
```

圖 2-8：使用 C++ 生成 falling_piece_update code（部分截圖）

【Function module】random_tetris

──隨機生成新的掉落方塊

決定下一個方塊為何，在傳進來更新的訊號為 1 前，維持住該方塊。

規則中，每輪會掉落七種不同的方塊，並依照隨機的順序掉落，一輪結束後會再開始新的一輪（換句話說，最多只會有兩個相同的方塊接連掉落）。

因此，在每個 posedge clk ， arr 陣列會更新目前 arr 陣列中 7 種方塊的排序（以 label 判斷， label 每個 posedge clk 會 + 1 ），更新方式以 label = 0 和 label = 1 為例， label = 0 時將陣列中第 0 個方塊跟第 4 個方塊對調， label = 1 時將陣列中第 1 個方塊跟第 5 個方塊對調(也就是該 label 往下數 4 互相對調)。

另一個 now 陣列則是當前要出現的 7 種方塊順序。當更新訊號為 1 時，即從 now [cnt] 取出方塊，並將 cnt + 1 ，如果 now 的方塊都被取完，更新 now 為當前的 arr ，以達成不同順序的功能。

不過更新 arr 的方法只會產生 84 種可能，比起真正隨機的順序的 128 種，有一些是順序不會出現的，因此當前 7 種方塊用完時，以經過的 clk 數量 mod 7 來決定 cnt （也就是不會都是從 now 的開頭取方塊，因為需要產生新方塊的時間是不固定的，藉此達成隨機的效果），還沒取完時皆是取後將 cnt + 1 （超過 6 將 cnt 歸零）。

out	輸出下一個方塊為何
isseven	當前這輪的方塊數到第幾個
now[6:0]	當前這輪方塊的順序
arr[6:0]	不斷更新方塊順序，當前這輪方塊用完時更新給 now
cnt	決定新一輪 now 中要從哪個位置開始取方塊
howmanyclk	紀錄經過幾個 clk
label	決定當前 arr 更新方式

- **遊戲顯示（螢幕）& 鍵盤**

螢幕顯示的部分藉由 keyboard 和 button 的訊號傳入 module scene 判斷當前該顯示的畫面（ ex ：起始畫面按下 down button 為 1P 、 up button 為 2P ）再傳入 Pixel Generater 裡根據 state_scene 、 h_cnt 、 v_cnt 決定 RGB 的值。

讀取鍵盤按鍵後傳入 KeyboardDecoder ，以 keydown 來判斷按鍵是否有按下。1P 設置上下左右為 I、K、J、L，直接落下和 Hold 方塊為 G 和 H。2P 設置上下左右為 5、2、1、3，直接落下和 Hold 方塊為 0 和 6。

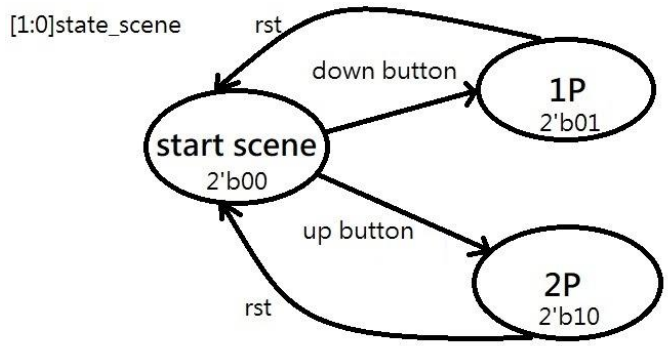

圖 2-9：scene STG

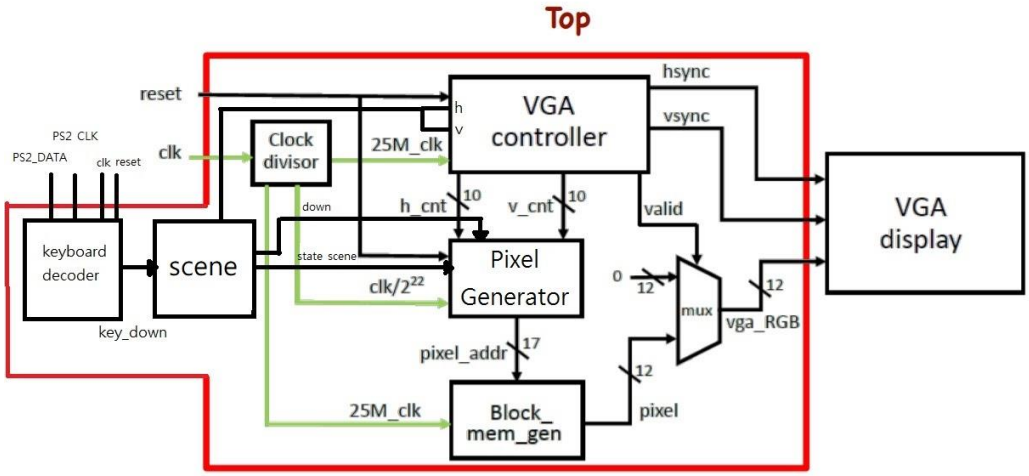

圖 2-10：VGA Block Diagram

【Pixel Generator】

如上所述，根據 state_scene 、 h_cnt 、 v_cnt 來決定 RGB 。

藉由一開始規劃好在 640 * 480 的畫面中各個 pixel 該顯示的圖片（ ex ： 210 <= h_cnt < 435 、 45 <= v_cnt <= 180 的話，顯示圖片 Tetris ），以 mux （ if-else ）來決定要分別傳給每個 block_memory_generator 的 address 。

圖 2-11：各個俄羅斯方塊（取自《Tetris Battle》）

- 起始畫面（ state_scene = 2’ b00 ）

  根據接收 module changestartscene 的 start_state 來決定輸出。

  start_state 每秒會變動，達到讓畫面中的方塊轉動的效果。

  起始畫面中 pixel 如沒有設定的圖片，則讀取的圖片為遊戲中背景的黑色方格。

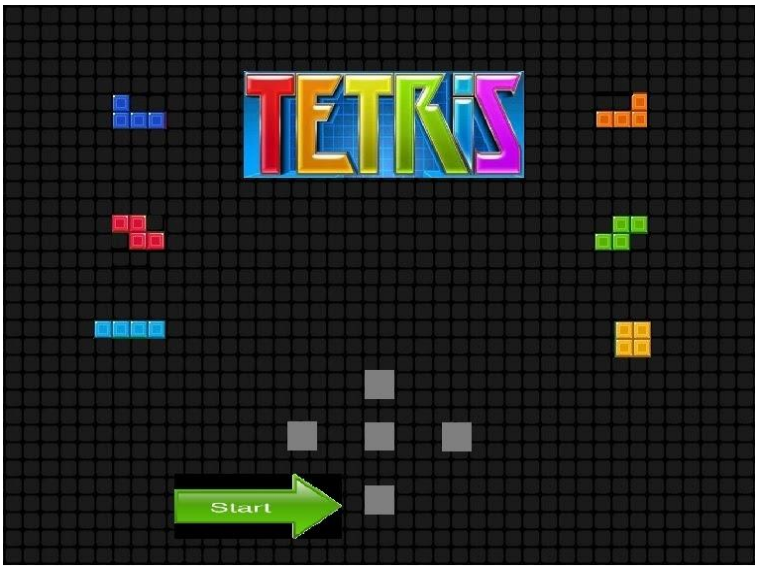

圖 2-12：起始畫面

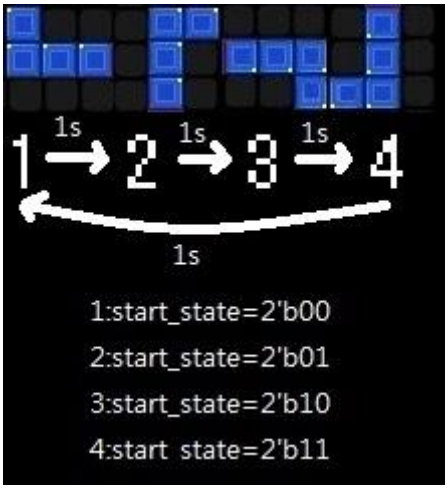

圖 2-13：以左上角的 J 為例，每個 start_state 顯示的樣子。

- 1P、2P（ state_scene = 2’ b01 、 state_scene = 2’ b10 ）

  1P 和 2P 的差別只在於圖片顯示的位子。

  將 VGA_controller 產生的 h_cnt 以及 v_cnt 傳入遊戲主程式（GameEgine ），來得到俄羅斯方塊遊戲 20 * 10 方格中各個方塊為何（實際上我設定每個方塊佔 15 * 15 個 pixel ，因此 1 個玩家的版面總共會佔 300 * 150 個 pixel），總共 7 個方塊加上背景黑色方格， 3 個 bit 即足夠判斷（以 case 來達成）。

  同時會從 GameEgine 讀入消除行數、下一個方塊以及暫時存放的方塊。

  數字的部分也是讀取圖片，將消除行數傳入 module cal_digit ， cal_digit 決定給 blk_mem_gen 的 address 來顯示消除行數。

下一個及暫時存放的方塊則是 GameEgine 傳訊號（何種方塊）給 Pixel Generator ，直接判斷並決定讀取的圖片以及顯示位置，而不是藉由遊戲版面的方式（傳入 h_cnt 和 v_cnt 得到該方格是哪個方塊）來決定讀取的圖片。

其他圖片 line 、 next 、 hold 也是設定好位置算出 address 傳給 blk_mem_gen 得到 RGB 的數值。

1P & 2P 中 pixel 如沒有設定的圖片，則讀取的圖片為背景圖。

圖 2-14：數字的讀取圖片

圖 2-15：hold、line、next 的讀取圖片

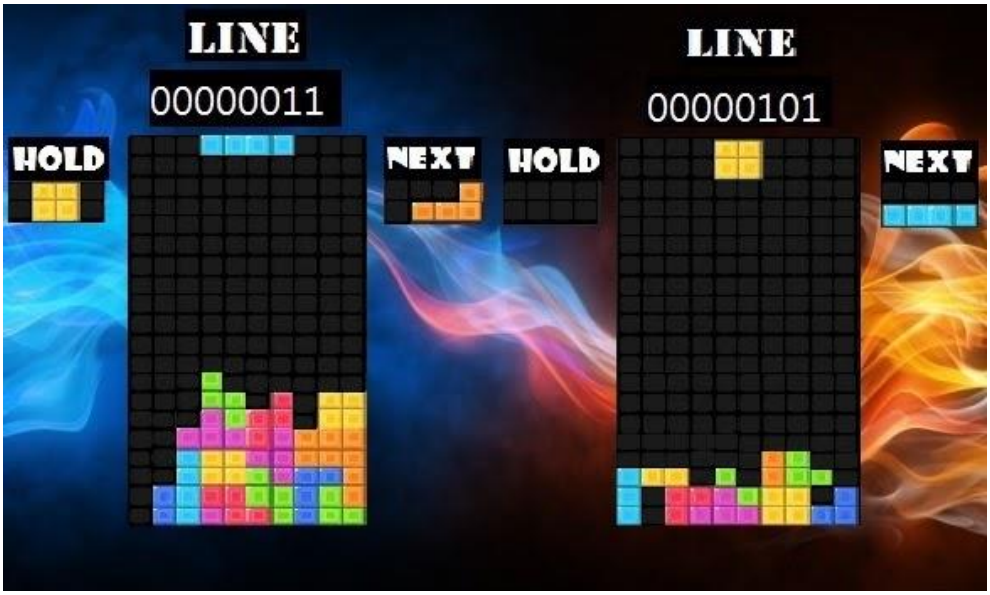

圖 2-16：2P 遊玩畫面

- **音樂（module musicmain）**

音樂的部分用的是俄羅斯方塊的經典配樂＂貨郎＂。遊戲中消除行數越多，掉落速度會越快，同時音樂的節奏也會變快， 2P 模式則是取兩人消除行數較高的人決定音樂節奏。

藉由傳入 BEAT_FREQ 給 PWM_gen 輸出音樂的速度， PlayerCtrl 依照這個速度傳訊號給 Music ， Music 再根據訊號輸出 freqence of tones 傳給 PWM_gen 輸出特定頻率。只要用 mux 控制 BEAT_FREQ 即可達到控制音樂速度的功能。

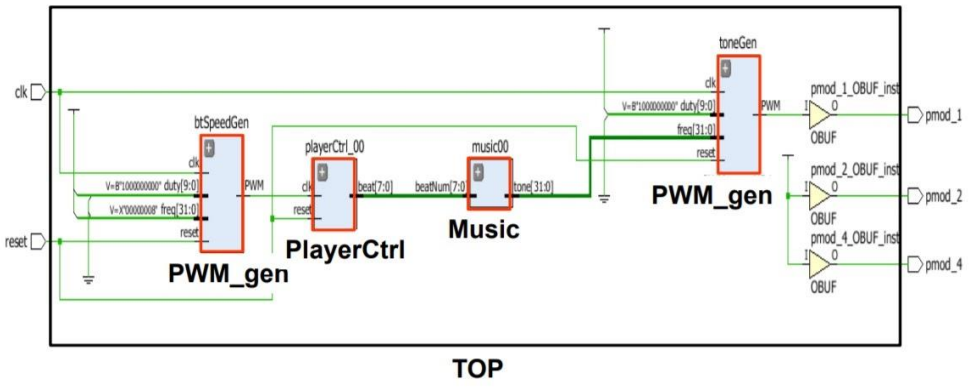

圖 2-17：musicmain block diagram

三、尚未完成功能

- **遊戲主程式**
    1. 方塊掉落至底部時，延遲一定時間後才生成新方塊
    2. Wall Kick
    3. 時間倒數計時
    4. 更完善的計分方式
    5. 更完善的調整方塊掉落速度
- **遊戲顯示**
    1. 勝負場景切換
    2. 消除特效
    3. 更換方塊造型