108061248_葉軒瑜：功能構想、報告撰寫、影片剪輯、說明書製作
108061112_林靖：功能設計、程式撰寫、錯誤處理、技術指導、diagram 繪製

「猜數字」遊戲機使用說明

※插上電源開啟開關後，請先將標號 4 的開關先關掉再打開重設機器。若遊戲中遇到當機或是鍵盤沒有反應的問題，一樣將標號 4 的開關先關掉再打開就能解決囉！
※建議接上音響設備，遊戲體驗更佳！
※需外接非機械式鍵盤進行遊戲

本遊戲機共有四個預設模式，共八種自定義模式可供遊玩！

模式/開關設置	**標號 1**	**標號 2**	**標號 3**
玩家 VS 電腦	ON	OFF	ON
電腦 VS 玩家	OFF	ON	OFF
電腦 VS 電腦	ON	ON	ON
玩家 VS 玩家	OFF	OFF	OFF

本遊戲共有四個階段：

階段一、等待重新開始遊戲
這時候顯示面板上會顯示最新一次猜測的 AB 結果以及完成遊戲所花費的時間！
按下 Enter 來重新開始遊戲，進入階段二

階段二、輸入目標待猜數字(關閉標號 1 開關才會有此階段)
將你想讓電腦或是其他玩家猜的數字輸入進去吧！按下 Enter 來確認送出進入階段三。
※用右側鍵盤進行輸入
※注意不能有重複的數字，否則 Enter 無效

階段三、輸入猜測
標號 2 開關關閉：
輸入你想猜的數字吧！按下 Enter 來確認送出進入階段四。
※用右側鍵盤進行輸入
※注意不能有重複的數字，否則 Enter 無效

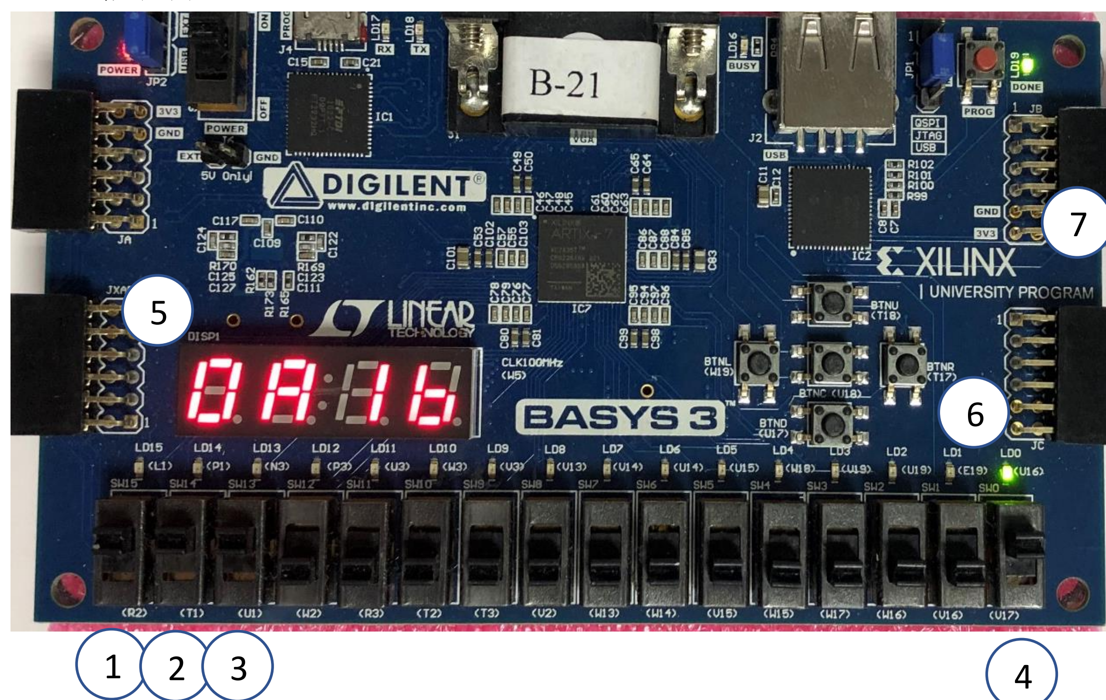
標號 2 開關打開：
電腦會經過縝密的思考後(大約 0.01 秒)將它猜的數字顯示在螢幕上～
按下 Enter 進入階段四。

階段四、顯示結果
標號 3 開關關閉：
比對待猜數字與猜測來輸入是幾 A 幾 B 吧～按下 Enter 重新回到階段三。
※用右側鍵盤進行輸入
※注意 A+B 不能大於 4(不包含)，否則 Enter 無效
※可以按鍵盤上的 X 或 Y 來確認待猜數字(X)與猜測(Y)
標號 3 開關打開：
猜測結果顯示在螢幕上啦，絞盡腦汁用這個結果猜對答案吧！
按下 Enter 重新回到階段三。

若在階段四時得到了 4A0B 的結果，螢幕會顯示花費時間，並播放慶祝音樂為你祝賀！重新回到階段一。
可以拿花費時間或是 LED 燈的數量(使用回合數)去跟其他朋友較勁誰是猜數字大師！

其他功能：見下方鍵盤及 FPGA 說明

FPGA 板說明



標號

1：設定自動產生要猜的目標數字（關：由玩家輸入、開：電腦隨機產生）

2：設定自動產生猜的數字（關：由玩家輸入、開：電腦自動猜）

3：設定自動產生結果（關：由玩家判斷後輸入結果、開：電腦自動比對結果）

4：Reset 重置系統按鍵

5：顯示面板

6：LED 亮燈數表示現在回合數

7：音源接頭

鍵盤互動

左側：

Enter：確認按鍵

0~9、Q(10)、W(11)、E(12)、R(13)：顯示該回合猜過的數字

0~9、Q(10)、W(11)、E(12)、R(13) + 左 Shift：顯示該回合相對應的結果

-、+：調整音量

<、>：移動標點(小數點)位置

Backspace：刪除標點上的數字

T：顯示現在耗時

X：顯示要猜的目標數字

Y：顯示現在回合猜的數字

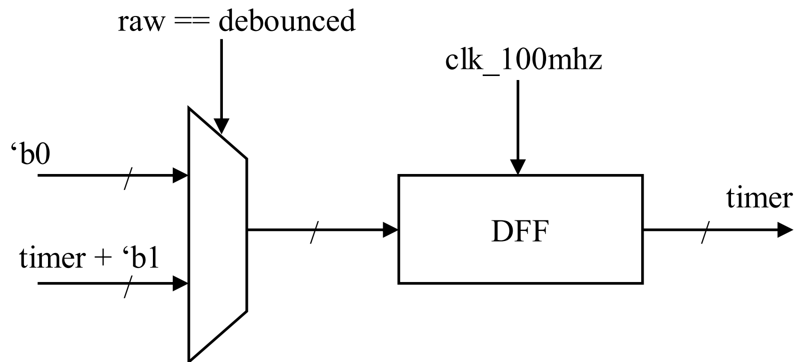
右側：輸入數字的鍵盤

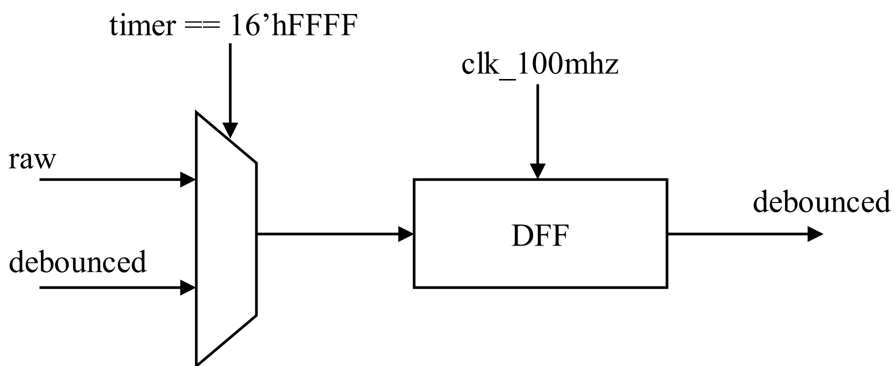
1. Specification

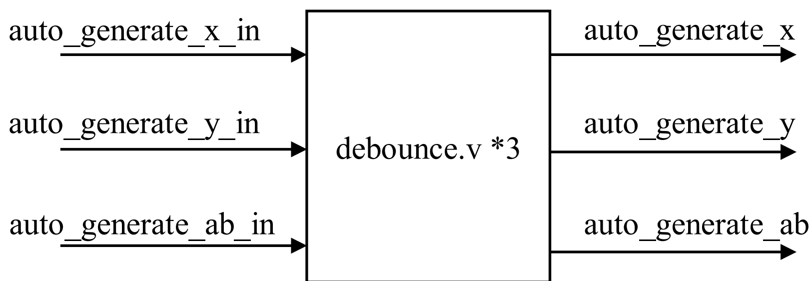
```
module final_project_TOP(

  // SSD
  output [7:0] ssd_cathode,
  output [3:0] ssd_anode,

  // I2S (speaker)
  output audio_mclk, // master clock
  output audio_lrck, // left-right clock
  output audio_sck,  // serial clock
  output audio_sdin, // serial audio data input

  // LED
  output reg [15:0] led,

  // PS2 (keyboard)
  inout PS2_DATA,
  inout PS2_CLK,

  // DIP switch
  input auto_generate_x,
  input auto_generate_y,
  input auto_generate_ab,

  // miscellaneous
  input clk_100mhz,
  input rst_n // active low reset
);
```

2. Block Diagram

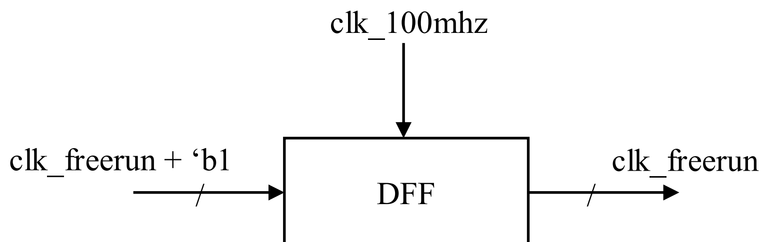
(1) debounce.v



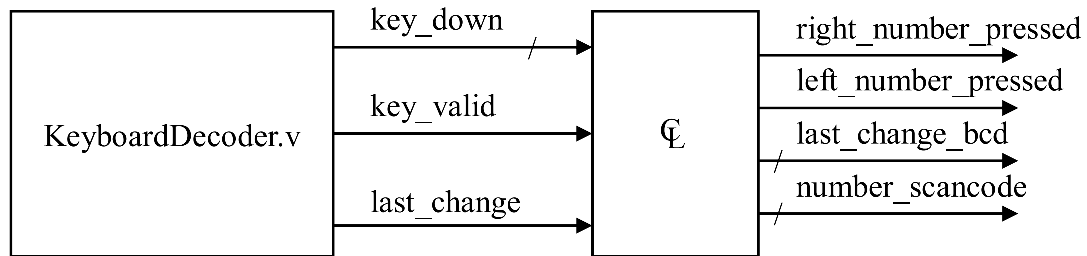




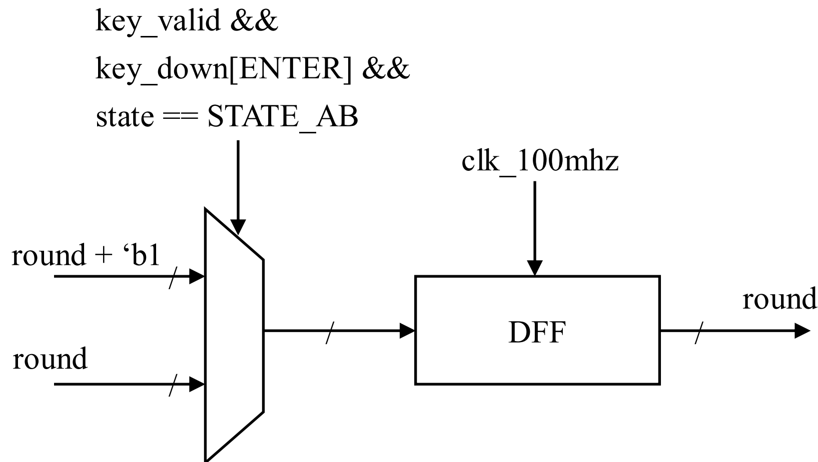
(2) clk_freerun



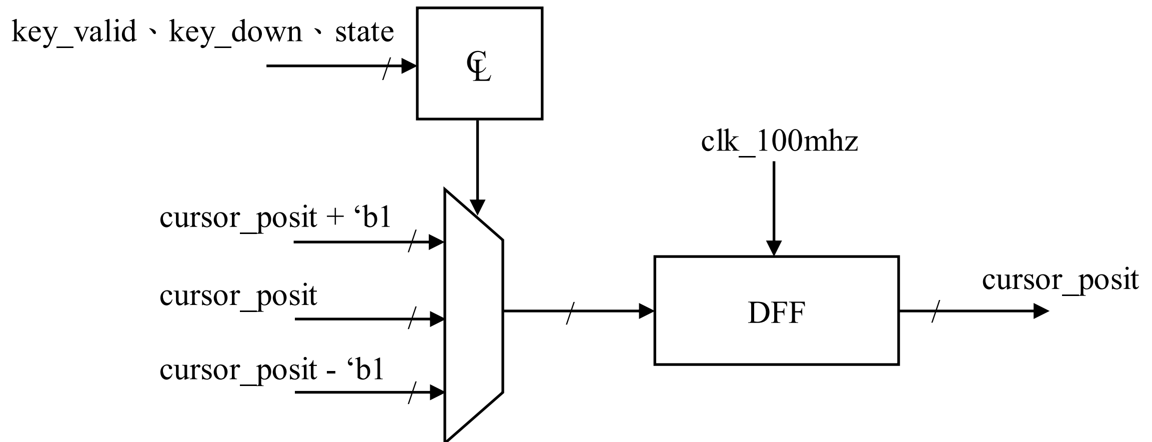
(3) Keyboard



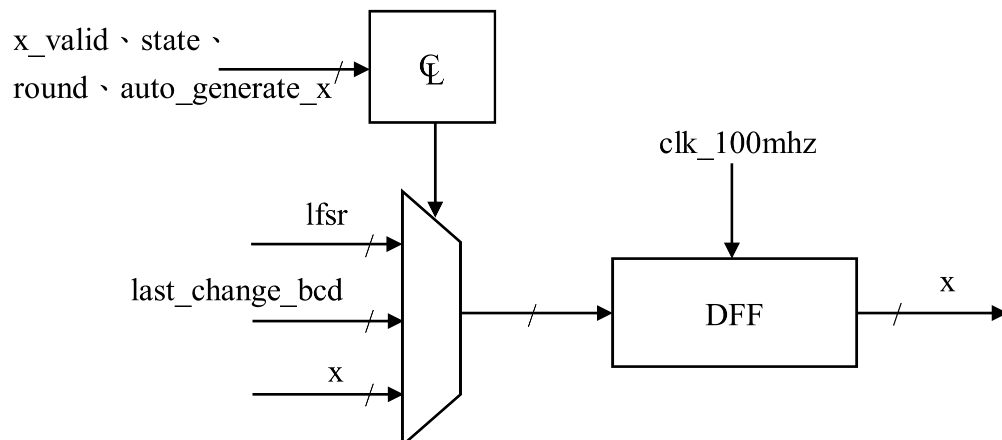
(4) Round

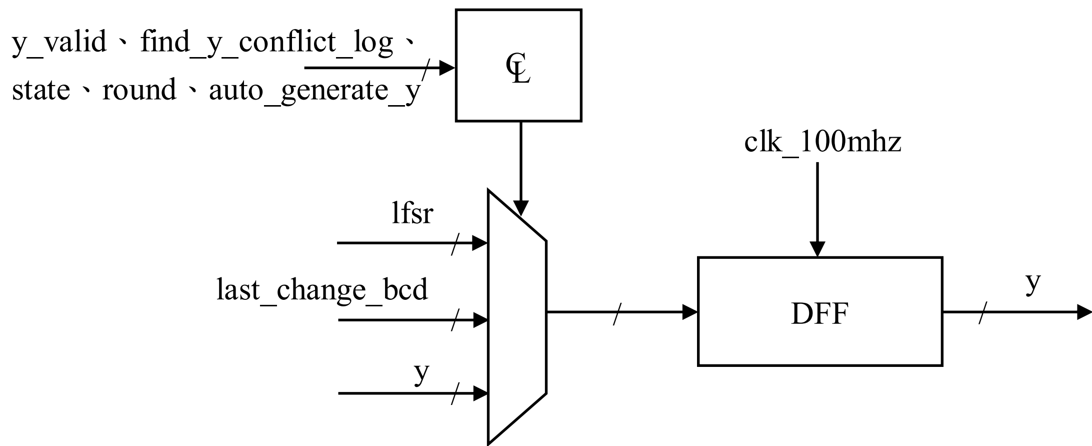


(5) Cursor_posit

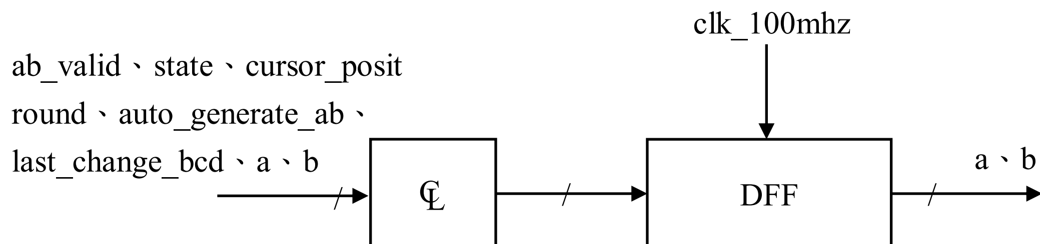


(6) X&Y

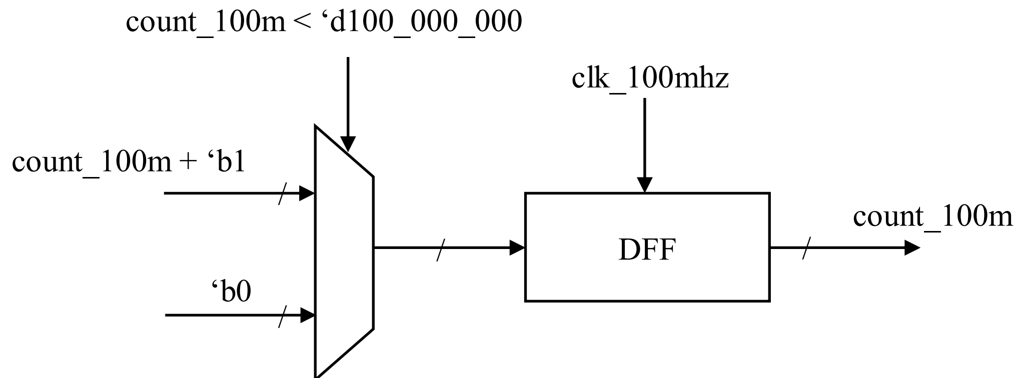


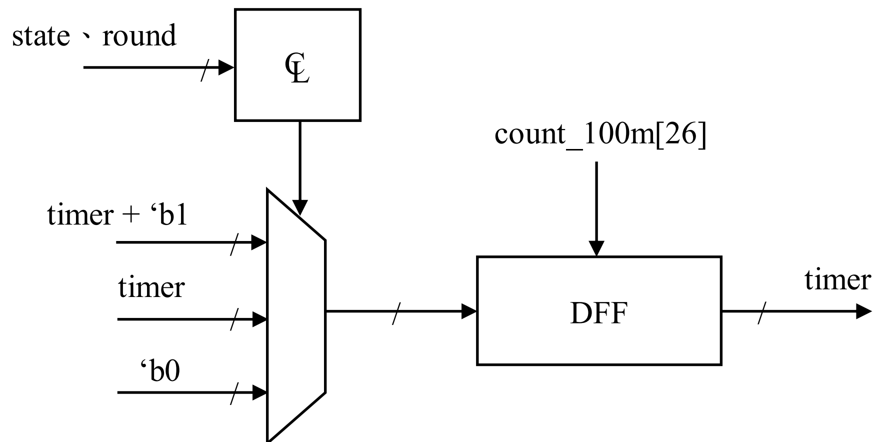


(7) A、B

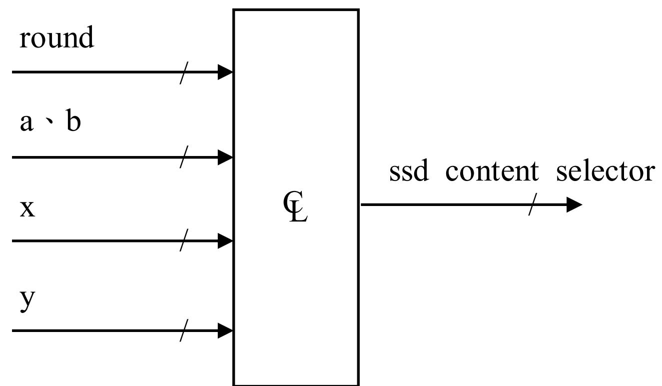


(8) Timer



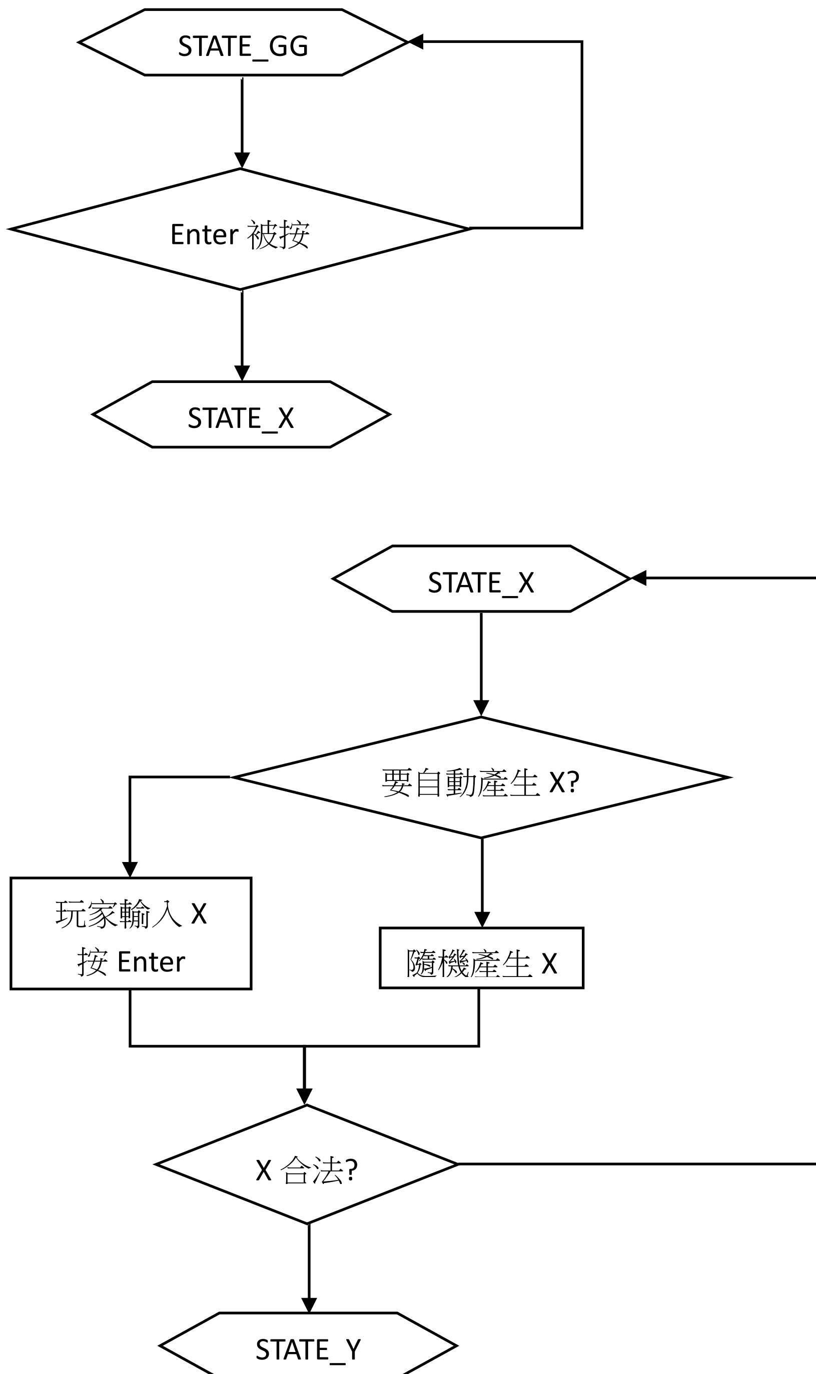


(9) ssd_content_selector

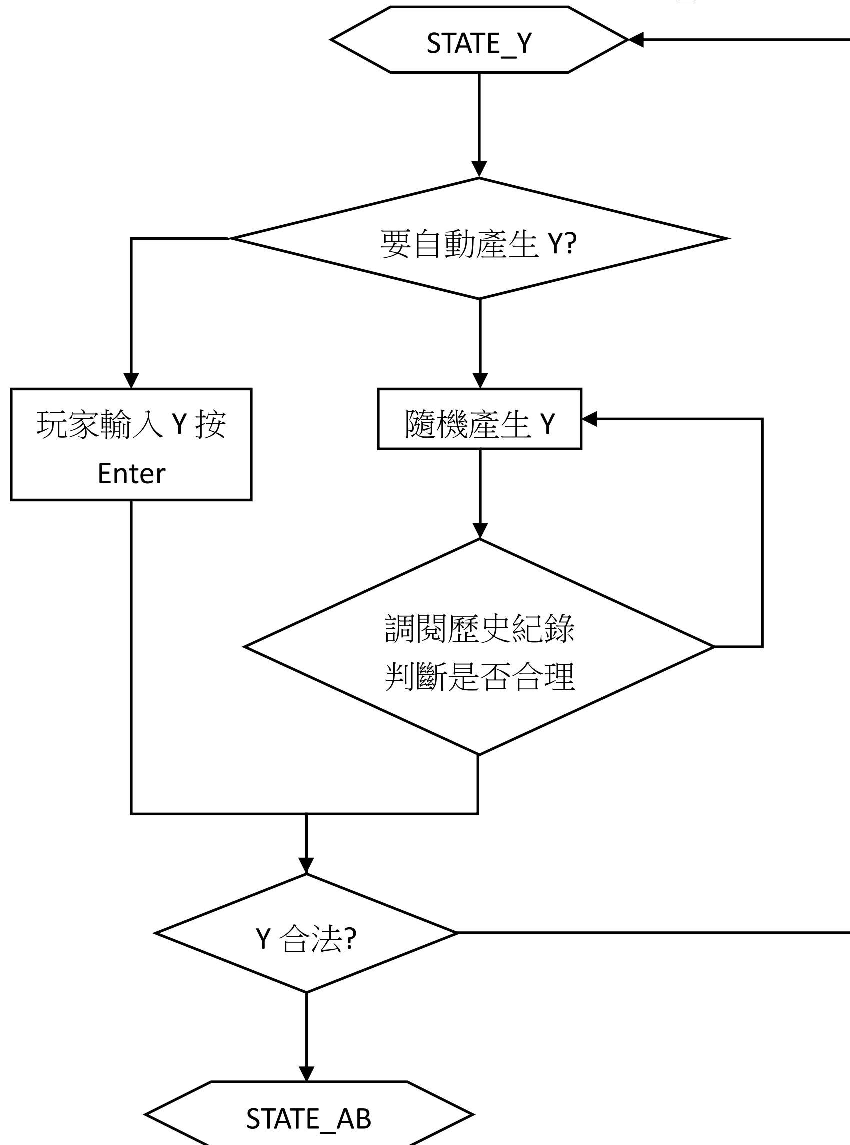


3. Finite state machine

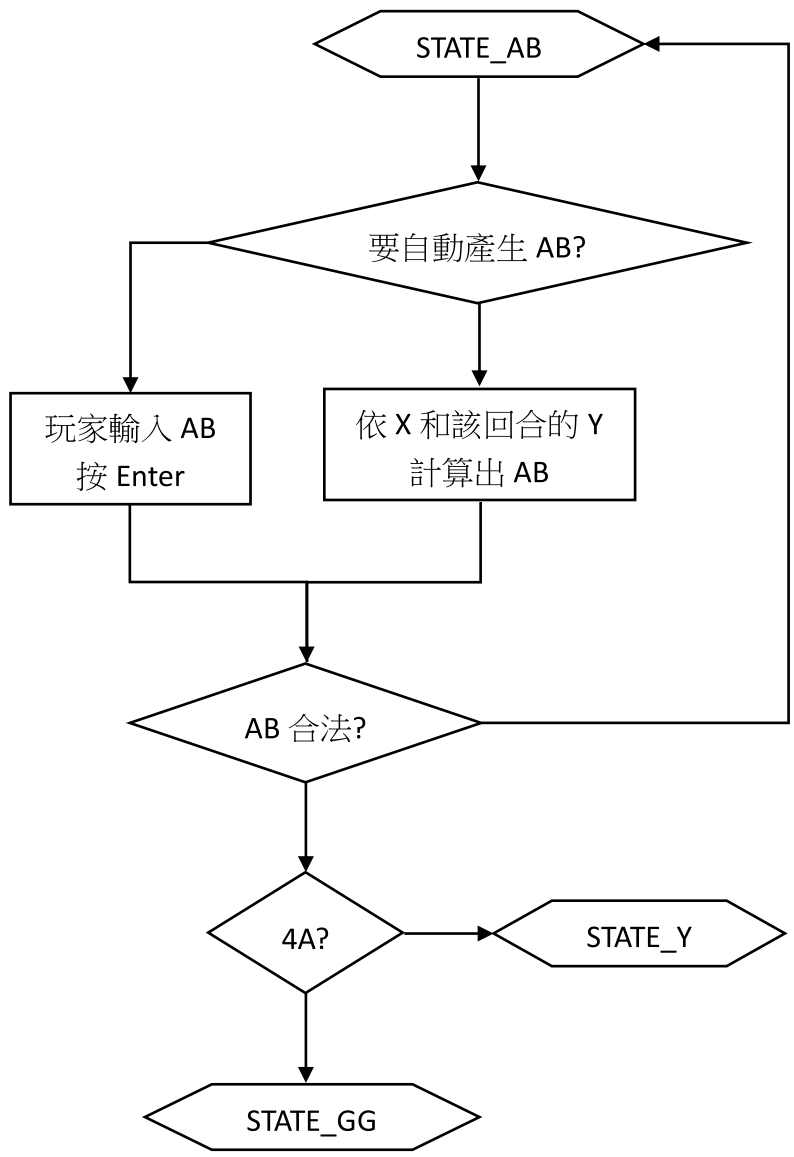
遊戲機總共有四個狀態：遊戲結束(STATE_GG)、產生目標 X(STATE_X)、產生該回合的猜測 Y(STATE_Y)、產生該回合的結果 AB(STATE_AB)。從開機到遊戲結束的過程，依序為：GG→X→Y→AB→Y→AB→Y→AB→……→GG。一開始「GG→X→」屬於遊戲機的初始化階段；玩家按下 Enter 開始遊戲之後，如果要自動產生 X(即 auto_generate_x == TRUE)，遊戲機會自動產生 X，並自動進入下一個狀態 STATE_Y；否則遊戲機會等待玩家使用鍵盤右側的數字鍵輸入四位數字，並於玩家按下 Enter 之後進入下一個狀態。



緊接著是第一回合「Y→AB→」。狀態 STATE_Y 時，如果要自動產生 Y(即 auto_generate_y == TRUE)，遊戲機會自動產生 Y，並顯示在七段顯示器上給玩家看；否則遊戲機會等待玩家使用鍵盤右側的數字鍵輸入四位數字。確定 Y 之後，玩家可以按 Enter 來進入下一個狀態 STATE_AB。



狀態 STATE_AB 時，如果要自動產生 AB(即 auto_generate_ab == TRUE)，遊戲機會根據 X 和該回合的 Y，自動計算出 AB，並顯示在七段顯示器上給玩家看；否則遊戲機會等待玩家使用鍵盤右側的數字鍵輸入 A 和 B。確定 AB 之後，玩家可以按 Enter 來進入下一回合的狀態 STATE_Y。



遊戲機每一回合都會歷經一次狀態 STATE_Y 和一次狀態 STATE_AB。如果於某一回合的狀態 STATE_AB 時，發現該回合的結果為 4A0B，則判定為遊戲結束，並自動進入狀態 STATE_GG。

4. Implement

(1) 猜數字遊戲中的一些重要變數

遊戲機需要記住的變數包含以下三項：目標 X、每回合的猜測 Y、每回合的結果 AB。在 code 裡，我們 infer 了一些 registers 的多維陣列來儲存這些變數，以方便隨時存取。目標 X 的 registers 陣列較單純，以 reg [3:0] x[0:3]宣告，可以 x[m]存取；此 m 介於 0~3，表示我們想要存取個位數、十位數、百位數、還是千位數。而猜測的數字 Y 則較複雜，因為每回合都有不同的 Y 要儲存，所以會多一維，以 reg [3:0] y[0:15][0:3]宣告，以 y[n][m]存取；此 n 介於 0~15，表示我們想要存取第幾回合的 Y 值；而此 m 與 X 的用法相同，表位數。每回合的結果 AB 則與 Y 相似，只差在每組 registers(相當於 RAM 的寬度)只需要用到 3 位元。

(2) debounce

由於 auto_generate_x、auto_generate_y、auto_generate_ab 的來源都是 DIP switch，所以要先經過 debounce 的模組，過濾掉雜訊，才能安心的使用。

```
12  always @(posedge clk or negedge rst_n) begin
13    if (~rst_n) begin
14      timer <= 'b0;
15    end else if (raw == debounced) begin
16      timer <= 16'b0;
17    end else begin
18      timer <= timer + 16'b1;
19    end
20  end
```

此計時器的時間代表「輸入值已經改變了多久」。如果新的輸入值持續足夠長的時間，則表示新的輸入已經穩定。

```
22  always @(posedge clk or negedge rst_n) begin
23    if (~rst_n) begin
24      debounced <= 'b0;
25    end else if (timer == 16'hFFFF) begin
26      debounced <= raw;
27    end else begin
28      debounced <= debounced;
29    end
30  end
```

如果新的輸入已經穩定，則以新的輸入覆寫掉舊的訊號。

(3) [31:0] clk_freerun;
這是一個三十二位元的除頻器。因為連接的時脈為 100M Hz，所以可以產生各種較緩慢的 10^6/2^n Hz 時脈，提供給其他有需要的模組使用。

(4) KeyboardDecoder KeyboardDecoder
老師提供的模組，用來解讀鍵盤的訊號。

(5) Localparam：KEY_???
這些是 scancode（鍵盤掃描碼），是鍵盤傳送給 FPGA 的一串數字，讓 FPGA 知道現在有那些按鍵被按下。像是 KEY_0R 表示右邊的 0 被按下、KEY_T 表示英文字母 T 被按下、KEY_LSHIFT 代表左邊的 shift 被按下等等...

(6) right_number_pressed & wire left_number_pressed;
這兩個訊號分別表示鍵盤左側和右側的數字鍵是否有被按下。如果偵測到右邊鍵盤的 0~9 之中，任何一個數字鍵被按下，right_number_pressed 就會變成 1'b1，否則為 1'b0。

(7) [3:0] last_change_bcd
用 LUT 將數字鍵的 scancode 轉換成 BCD，表示最後按下右邊鍵盤的 BCD 數值，輸入數字時就從此變數中存取數字。

(8) [7:0] number_scancode[0:15];
類似 LUT，將數字鍵的 scancode 儲存在一寬度為八位元、深度為十六位元的 RAM，以方便存取。可直接表示查看回合的相對應按鍵，(考慮到 code 的簡潔度與可讀性)使硬體描述方便用 for 迴圈表示。

(9) [15:0] lfsr;
16 位元的線性反饋移位暫存器(XOR 亂數產生器)，在「自動產生 X」及「自動產生 Y」時用來產生 X 與 Y。

(10) [3:0] round

```
28 always @(posedge clk_100mhz or negedge rst_n) begin
29   if (~rst_n) begin
30     round <= 'b0;
31   end else if (key_valid && key_down[KEY_ENTER]) begin
32     if (state == STATE_GG) round <= 16'h0;
33     if (state == STATE_AB) round <= round + 16'b1;
34   end else begin
35     round <= round;
36   end
```

round 紀錄的是現在進行到的回合數，在開始新的一局遊戲時，歸零。每當按下 Enter 確認 AB 後(要產生下一個 Y 時)，加一。

(11) [1:0] cursor_step & [1:0] cursor_posit

```
31   assign cursor_step = (state == STATE_AB) ? (2'd2) : (2'd1);
32
33   always @(posedge clk_100mhz or negedge rst_n) begin
34     if (~rst_n) begin
35       cursor_posit <= 'b0;
36     end else if (key_valid) begin
37       case (1'b1)
38         right_number_pressed:    cursor_posit <= cursor_posit - cursor_step;
39         key_down[KEY_BACKSPACE]: cursor_posit <= cursor_posit + cursor_step;
40         key_down[KEY_MORE_THAN]: cursor_posit <= cursor_posit - cursor_step;
41         key_down[KEY_LESS_THAN]: cursor_posit <= cursor_posit + cursor_step;
42         key_down[KEY_ENTER]:     cursor_posit <= 2'd3;
43         default:                 cursor_posit <= cursor_posit;
44       endcase
45     end else begin
46       cursor_posit <= cursor_posit;
47     end
48   end
```

此段硬體描述負責控制 cursor 的位置(在七段顯示器上以閃爍的小數點表示)。輸入數字時、或按下大於時，cursor 會向右移動；按下 backspace 時、或按下小於時，cursor 會向左移動；按下 Enter 則會使 cursor 回到最左邊。

(11) x_valid & y_valid & ab_valid

上面的三個 vaild 訊號是判斷 x、y、ab 是不是合法輸入。(X、Y 不合法意指在不同位數出現重複數字，AB 不合法意指 A 和 B 的總和大於 4)

(12) 自動生成 Y(電腦玩家)

```
458    reg [3:0] count_16;
459
460    always @(posedge clk_100mhz or negedge rst_n) begin
461      if (~rst_n) count_16 <= 'b0;
462      else        count_16 <= count_16 + 4'b1;
463    end
464
465    wire find_y_conflict_log;
466
467    assign find_y_conflict_log = (count_16 < round) &&
468      (
469        (y[round][0] == y[count_16][0]) +
470        (y[round][1] == y[count_16][1]) +
471        (y[round][2] == y[count_16][2]) +
472        (y[round][3] == y[count_16][3]) != a[count_16]
473      ||
474        (y[round][0] == y[count_16][3]) +
475        (y[round][0] == y[count_16][2]) +
476        (y[round][0] == y[count_16][1]) +
477        (y[round][1] == y[count_16][3]) +
478        (y[round][1] == y[count_16][2]) +
479        (y[round][1] == y[count_16][0]) +
480        (y[round][2] == y[count_16][3]) +
481        (y[round][2] == y[count_16][1]) +
482        (y[round][2] == y[count_16][0]) +
483        (y[round][3] == y[count_16][2]) +
484        (y[round][3] == y[count_16][1]) +
485        (y[round][3] == y[count_16][0]) != b[count_16]
486      );
```

此 final project 的一項特色是：具備猜玩家的數字 x 的功能。而這一段硬體描述就是能達到此功能的一大關鍵。我們的目標是建構出一個簡單而高效的演算法，在一局遊戲中，根據已經收集到的資料（每一回合的猜測 y 與對應的結果 ab），推測出哪些四位數字組合為潛在的目標 x、哪些四位數字組合絕對不可能為目標 x。因為使用硬體來實作，所以可以直接採用試誤法，意即：隨機產生出一個四位數並假設此隨機四位數就是目標 x，再一一跟遊戲紀錄 pair(y, ab)進行比對。如果發現有任何一回合的紀錄帶入後算出來的 ab 不符合(檢查出衝突)，就代表該隨機四位數不可能是真正的目標 x，並在下一個時脈取出另一個隨機四位數再行比對，如此反覆執行。由於我們的隨機數來自十六位元長度的 LFSR，所以保證進行至多 2^16-1 次以上的檢查，就可以利用此試誤法找到一組隨機四位數，符合所有歷史紀錄中的 pair(y, ab)。

為了簡化 code，每一個 100MHz 時脈內，只檢查一個回合的 pair(y, ab)。我們先建構一個四位元的上數計數器 count_16 。count_16 的數值介於 0 到 15 之間，表示當下要檢查哪一回合。仿照上述演算法的思路，先假設當下的 y 就是真正

的目標 x，把 count_16 與當下的 y 代入 find_y_conflict_log 以查詢該回合的 ab。如果發現與紀錄不符合，就代表此 y 不可能是真正的目標 x。對於每一個回合都進行一次檢查，這種作法可以保證所有十六回合中的每一組紀錄都在十六個 100MHz 時脈內被檢查到。且由於此 final project 的十六位元 LFSR 的週期是 2^16-1 的 100MHz 時脈，考慮 2^16-1 與 16 互質，可推知所有 LFSR 的組合都終將在極短時間內被(歷遍)檢查到。(排除 16'b0，不合法的 y 數值)。

(12) x、y、ab 的儲存

```
510   always @(posedge clk_100mhz or negedge rst_n) begin
511     if (~rst_n) begin
512       x[0] <= 'b0;
513       x[1] <= 'b0;
514       x[2] <= 'b0;
515       x[3] <= 'b0;
516     end else if (state == STATE_GG && key_valid && key_down[KEY_ENTER]) begin
517       x[0] <= 4'hF;
518       x[1] <= 4'hF;
519       x[2] <= 4'hF;
520       x[3] <= 4'hF;
521     end else if (state == STATE_X) begin
522       if (auto_generate_x) begin
523         if (!x_valid) begin
524           x[0] <= lfsr >>  0;
525           x[1] <= lfsr >>  4;
526           x[2] <= lfsr >>  8;
527           x[3] <= lfsr >> 12;
528         end
529       end else begin // !auto_generate_x
530         if (key_valid) begin
531           if (right_number_pressed)      x[cursor_posit] <= last_change_bcd;
532           if (key_down[KEY_BACKSPACE]) x[cursor_posit] <= 4'hF;
533         end
534       end
535     end else begin
536       x[0] <= x[0];
537       x[1] <= x[1];
538       x[2] <= x[2];
539       x[3] <= x[3];
540     end
541   end
```

這段 code 描述 x 的數值。當狀態為 GG 時(gameover)，等待玩家按下 enter 鍵，並在玩家按下 enter 的那個時脈，將 x 的數值歸零(四位數字皆預設為 4'hF，代表皆未指定數值)，此行為是初始化，每次重新開始一局遊戲都會執行一次。

當狀態為 X 時，要更變 x 的數值：如果要讓 FPGA 自動產生一組四位數字來當作 x(意即 auto_generate_x == TRUE)，就從 LFSR 中取出一組隨機四位數字賦值，並在每一個時脈中檢查 x 是否合法(看 x_valid，即檢查上一個時脈所取到的四位數字是否合法)。如果當下的 x 不合法，就從 LFSR 中取出另一組四位隨機數字賦值，並在下一個時脈再次檢查取到的四位數是否合法。

用軟體語言可以比喻此過程，如下：

```
do {
    x = random();
} while (x is not valid)
```

另，如果不想要讓 FPGA 自動產生 x(意即 auto_generate_x == FALSE)，就等待玩家使用鍵盤輸入。如果玩家按下了任一個右側數字鍵，就把該數字的數值(last_change_bcd)存入當下的 cursor 所在位置。如果玩家按下退格建(backspace)，就將 cursor 所在位置的位數改成 4'hF(預設值，代表未指定)。

x 的數值在其餘狀態下則保持不變。

```
579        if (auto_generate_y) begin
580          if (find_y_conflict_log || !y_valid) begin
581            y[round][0] <= lfsr >>  0;
582            y[round][1] <= lfsr >>  4;
583            y[round][2] <= lfsr >>  8;
584            y[round][3] <= lfsr >> 12;
585          end
```

描述 y 的方法跟描述 x 的方法差不多，比較需要注意的差異處如下：如果要自動產生 y，不但要如 x 一樣檢查 y 是否合法(是否有重複的位數?是否有介於 0~9 以外的數?)，還要另外檢查當下的 y 是否符合所有已經收集的 AB 結果，就如前面 find_y_conflict_log 提到的一樣，藉此達到讓電腦猜對數字。

描述 AB 的方法與前面的 Y 差不多，遊戲結束時重置，非自動產生 AB 時由玩家輸入。自動產生時，檢查當回合的 Y 與 X 的關係，每出現數字位置皆一樣會使 A+1，數字一樣但位置不同會使 B+1。而 AB 同樣具備儲存歷史紀錄的功能，也是一個多維陣列。

(13) timer

```
676   reg [26:0] count_100m;
677
678   always @(posedge clk_100mhz or negedge rst_n) begin
679     if (~rst_n) count_100m <= 'b0;
680     else        count_100m <= (count_100m < 27'd99_999_999) ?
681                               (count_100m + 27'b1) : (27'b0);
682   end
683
684   reg [11:0] timer;
685
686   always @(posedge count_100m[26] or negedge rst_n) begin
687     if (~rst_n) begin
688       timer <= 'b0;
689     end else if (round == 16'h0 && state == STATE_Y) begin
690       timer <= 12'd0;
691     end else if (state == STATE_GG) begin
692       timer <= timer;
693     end else begin
694       timer <= timer + 12'b1;
695     end
696   end
```

這是計時器的部分，先建構一個一秒的 counter，再用 state 判斷是否要開始計時。考慮到第一回合理論上不需要思考時間，在第二回合時計時器才會開始計時。遊戲結束時，timer 不會歸零，而是保存時間顯示到螢幕上，再次開始時才會歸零。

(14) [3:0] ssd_content_selector[0:3];

ssd_content_selector 可以比喻成一個容器。此段硬體描述可決定哪個時候要在七段顯示器上顯示什麼資訊。像是按下 T 時顯示現在耗時，按下左側鍵盤上的數字會顯示相對應回合的 Y record，若同時按下左側 shift 與左側數字則會顯示相對回合的 AB record，以及在各個狀態顯示各自該顯示的資訊。

(15) [6:0] ssd;

這裡是顯示字元轉換到七段顯示器的 decoder。infer 一個類似 LUT 的結構將 BCD 轉換成欲顯示的符號(cathode pattern)。

(16) cursor 顯示

```
855    wire hide_cursor;
856
857    assign hide_cursor = (key_down[KEY_T])
858                      || (state == STATE_GG)
859                      || (state == STATE_X  && auto_generate_x)
860                      || (state == STATE_Y  && auto_generate_y)
861                      || (state == STATE_AB && auto_generate_ab);
862
863    wire point;
864
865    assign point = (clk_freerun[25]) // blinking cursor
866                 & (clk_freerun[19:18] == cursor_posit)
867                 & (~hide_cursor);
868
869    assign ssd_cathode = {ssd, ~point}; // merge cathode
870
871    assign ssd_anode = ~( 'b1 << clk_freerun[19:18] ); // anode 1-hot enable
```

此段簡短的程式就完整的描述了 cursor 的顯示。hide_cursor 判斷現在是否需要隱藏 cursor(反向就是是否需要顯示 cursor)，還 AND 了 clk_freerun[25]讓 cursor 會閃爍，更清楚的提示玩家 cursor 的位置。

而 ssd_anode 是利用視覺暫留，讓七段顯示器的那四個 digits 看起來可以同時顯示四個不同的符號。

(17) LED

```
950    always @* begin
951      led = ~(16'b1111_1111_1111_1111 << round);
952      led[round] = clk_freerun[25]; // blinking
953    end
```

此段硬體描述將回合數以 LED 燈顯示。例如：第一回合會閃爍最右邊的 LED；第三回合會亮起右邊三顆，其中左邊閃爍的那顆表示當前回合。

(18) 音效

接著介紹音效部分，一開機就能聽到開機音樂 secret monkey island，按下 enter 開始後會聽到一串開始音效，而在輸入右側鍵盤的數字時會發出按電話的聲音，這個還是國家統一過的雙音頻單音。在完成遊戲後會重頭播放 secret monkey island。

程式部分，先用 for 迴圈加上二維陣列建構樂曲，配合之前 LAB 學習過的音訊處理、counter 及狀態判斷，就能在進到 state_GG 及第一次進到 state_X 時播放出音樂。

另外建構 dtmf_high(low)_freq 設定按下數字按鍵時要發出的雙頻率，在按下數字時播放。
按下左側鍵盤的-跟+可以調整音效的音量。

```
990    localparam WIDTH_OF_FREQUENCY = 16;
991    localparam NUMBER_OF_NOTE = 1440;
992    localparam NON_TONE = 16'd1;
993
994    reg [15:0] duration_list[0:NUMBER_OF_NOTE-1]; // ms
995    reg [15:0] freq_list     [0:NUMBER_OF_NOTE-1]; // Hz
```

使用此種寫法可以 infer 一個單純的組合邏輯，利用 for loop 使 logic synthesizer 合成出 RAM。由於 code 的長度較長，未放入此頁，詳見.v 檔案。duration_list 和 freq_list 這兩個是寬度為 16、深度為 1440 的 RAM，分別記錄了慶祝音樂的音符音高(頻率)和音符時值(持續多久)，可以視為樂譜。

```
1025   reg  [31:0] duration_count;
1026   reg  [15:0] note_index;
1027
1028   always @(posedge clk_100mhz or negedge rst_n) begin
1029     if (~rst_n) begin
1030       duration_count <= 'b0;
1031     end else if (state == STATE_AB) begin
1032       duration_count <= 32'b0;
1033     end else if (duration_count < duration_list[note_index] * 100_000) begin
1034       duration_count <= duration_count + 32'b1;
1035     end else begin
1036       duration_count <= 32'b0;
1037     end
1038   end
1039
1040   always @(posedge clk_100mhz or negedge rst_n) begin
1041     if (~rst_n) begin
1042       note_index <= 'b0;
1043     end else if (state == STATE_AB) begin
1044       note_index <= 16'b0;
1045     end else if (duration_count == duration_list[note_index] * 100_000) begin
1046       if (note_index < NUMBER_OF_NOTE) note_index <= note_index + 16'b1;
1047       else                             note_index <= 16'b0;
1048     end else begin
1049       note_index <= note_index;
1050     end
1051   end
```

我們需要建構出一些專門用來解讀儲存在 RAM 裡的樂譜的硬體，以便將各個頻率在適當的時間傳送給 Pic24 所規範的 parallel to serial convertor。由於樂譜已經紀錄了音符音高(頻率)和音符時值(持續多久)，所以可以善用 index 來追蹤當下播放的音符是第幾個音符。

如.v 檔案中所示，duration_count 會從 0 一直數到(duration_list[note_index] * 100_000)，然後歸零。每一次歸零 note_index 都會加一；每一次歸零都代表一個音符的結束、另一個音符的開始。而 note_index 則表示當下撥放的音符是整首樂曲的第幾個音符，由於此曲目有 1440 個音符，故 note_index 的數值介於

0~1439。因為要循環播放，所以在超過 1439，會重設成 0，從頭重播。

```
1065    wire [15:0] th_left_freq_list[0:11];
1066    wire [15:0] th_right_freq_list[0:11];
```

與 duration_list 和 freq_list 類似地，th_left_freq_list 和 th_right_freq_list 也是儲存在 RAM 裡的樂譜。比較顯著的不同是，這兩個 RAM 所儲存的資料一個是左聲道的音符音高、一個是右聲道的音符音高；而這 12 個音符的時值皆相同。

```
1102    reg  [31:0] th_duration_count;
1103
1104    always @(posedge clk_100mhz or negedge rst_n) begin
1105      if (~rst_n) begin
1106        th_duration_count <= 'b0;
1107      end else if (state == STATE_GG) begin
1108        th_duration_count <= 32'b0;
1109      end else if (th_duration_count < 10_000_000) begin
1110        th_duration_count <= th_duration_count + 32'b1;
1111      end else begin
1112        th_duration_count <= 32'b0;
1113      end
1114    end
1115
1116    reg  [3:0] th_note_index;
1117
1118    always @(posedge clk_100mhz or negedge rst_n) begin
1119      if (~rst_n) begin
1120        th_note_index <= 'b0;
1121      end else if (state == STATE_GG && key_valid) begin
1122        if (key_down[KEY_ENTER] || right_number_pressed) th_note_index <= 16'b0;
1123      end else if (th_duration_count == 10_000_000 && th_note_index <= 16'd11) begin
1124        th_note_index <= th_note_index + 16'b1;
1125      end else begin
1126        th_note_index <= th_note_index;
1127      end
1128    end
```

同理，我們需要建構一些用來解讀樂譜的硬體。此部分與上一首曲目的結構類似，使用 th_note_index 來追蹤當下撥放到哪一個音符，而 th_duration_count 則用來計時，由於這 12 個音符的時值皆相同，此處我們設定為每當上數到 10_000_000 時就歸零。換句話說，每個音符僅撥放持續 0.1 秒鐘。須特別留意的是：由於我希望此音效在開始遊戲時撥放一次，不循環撥放。因此我讓 th_note_index 在過了第 12 個音符之後，維持停滯不前；直到開始新的另一局遊戲，才重設指向第一個音符。

```
1155    wire [15:0] dtmf_low_freq[0:9];
1168    wire [15:0] dtmf_high_freq[0:9];
```

這些是我們在使用電話時，按下各個數字所對應到的「雙音多頻」。這些我們時常聽到的所謂「按鍵聲」其實是電話系統中電話機與交換機之間的信令，用於發送被撥打的號碼。與剛剛的那些樂譜不一樣的地方是，這些資料只記錄了 0~9 所對應的頻率(一個數字對應到兩個頻率)，不紀錄音符時值，因為每個音要被播放的時機在於各個數字被按下的那段時間。也沒有循環播放的問題。

```
1193  reg [31:0] freq_count_left_lim;
1194
1195  always @* begin
1196    freq_count_left_lim = 32'hFFFF_FFFF; // default
1197    if (16'b0 < th_note_index && th_note_index <= 16'd11)
1198      freq_count_left_lim = 50_000_000 / th_left_freq_list[th_note_index];
1199    if (state == STATE_GG)
1200      freq_count_left_lim = 50_000_000 / freq_list[note_index];
1201    if (right_number_pressed)
1202      freq_count_left_lim = 50_000_000 / dtmf_low_freq[last_change_bcd];
1203  end
1204
1205  reg [31:0] freq_count_right_lim;
1206
1207  always @* begin
1208    freq_count_right_lim = 32'hFFFF_FFFF; // default
1209    if (16'b0 < th_note_index && th_note_index <= 16'd11)
1210      freq_count_right_lim = 50_000_000 / th_right_freq_list[th_note_index];
1211    if (right_number_pressed)
1212      freq_count_right_lim = 50_000_000 / dtmf_high_freq[last_change_bcd];
1213  end
```

有了上述三種音效(分別為：紀錄於深度為 1440 的慶祝音樂 RAM、12 個開始遊戲音符、數字鍵雙音多頻)，我們必須決定何時該播放哪一個音效，也就是說，要決定在那些情況之下將哪些頻率等訊號傳送到方波產生模組。慶祝音樂要在 state == STATE_GG 時播放，由於此慶祝音樂僅占用一個聲道，我隨意挑了一個聲道來使用(我挑左聲道)。而開始遊戲音效要在 state == GG 且玩家按下 Enter 鍵的那一刻開始播放，而這些條件已經在控制 th_note_index 的硬體裡描述好了，所以我直接使用 th_note_index 的數值來決定是否要傳送此頻率信號。

```
1223  reg [31:0] freq_count_left;
1235  reg [31:0] freq_count_right;
1255  reg phase_left;
1267  reg phase_right;
1294  reg [3:0] volume;
1307  wire [15:0] audio_left;
1308  wire [15:0] audio_right;
```

將 freq_count_left_lim 和 freq_count_right_lim 各自作為兩個上數計數器 freq_count_left 和 freq_count_right 的上限。如此一來，這兩個計數器的頻率就會剛好是那些音效的音高。而 phase_left 和 phase_right 則表示左右聲道各自當下的音壓在方波波長的前半部還是後半部。如果當下處在方波的前半部，audio_left 和 audio_right 就各自賦值 16'h7FFF(二補數最大值)；如果當下處在方波的後半部，audio_left 和 audio_right 就各自賦值 16'h8000(二補數最小值)。需留意，我們在賦值前先向右 shift 了 volume，所以 volume 越小，振幅會越大、音量會越大；而 volume 越大，振幅則越小、音量也越小。此段硬體描述較單純，故不列出，詳見.v 檔案。

```
1335    assign audio_mclk = clk_freerun[1]; // 25MHz (~24.5760MHz)
1336    assign audio_lrck = clk_freerun[8]; // 25MHz/128 (~192kHz)
1337    assign audio_sck  = clk_freerun[3]; // 25MHz/128*32 (25MHz/4)
1338
1339    reg audio_lrck_delayed;
```

接下來就只是將波型的二補數訊號傳送到 parallel to serial convertor 即可。此部分跟 LAB08 的原理一模一樣，詳見.v 檔。

5. Conclusion

在這次的 project 中，我們用到了這學期所學的所有東西。包括了簡單的上下數 counter、計時器、LED 燈、七段顯示器、鍵盤輸入、音效、建構 FSM，以及最重要的——設計一個設想周全的功能。除了用上之前所學，我們還學習到了如何使用多維陣列來儲存以及讀取歷史紀錄、如何建立一首曲子並在該播放時放出來以及 for 迴圈簡化繁雜重複的程式。而我們獨有的特色「AI 電腦玩家」更是讓我們驚豔，使用簡單的判斷與亂數就能在平均五回合內猜出正確的數字，是這次學習到的另一個大重點！