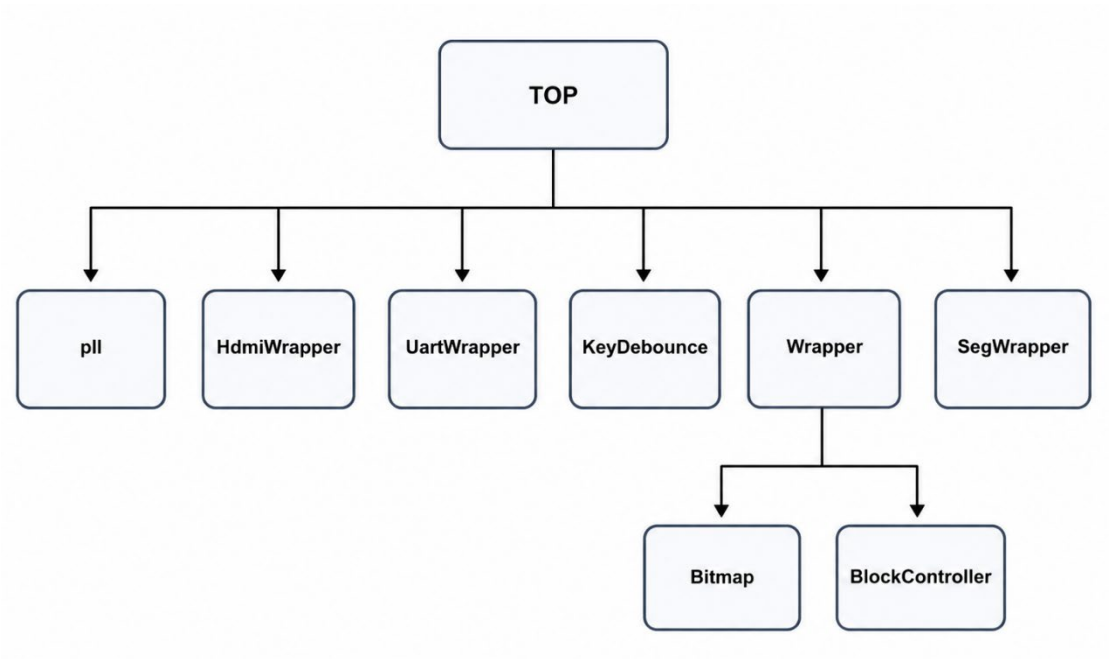
SME212 Digital Integrated Circuits Final Tetris Project Report

1. System Analysis:



(1) Top: responsible for connecting all submodules.

(2) KeyDebounce: responsible for processing key signals and returning debounced key information.

(3) Wrapper: the game-function implementation section, connecting the Bitmap and BlockController modules.

(4) Bitmap: responsible for storing and updating the current board information, handling bottom collision, line clearing and scoring, and determining whether each type of movement is valid.

(5) BlockController: responsible for updating, moving, and processing the current moving block.

(6) SegWrapper: responsible for sending the game score to the seven-segment display for score display.

(7) pll: a phase-locked loop that implements system-level management and offset control of the clock network, handling frequency multiplication, frequency division, phase shifting, and programmable duty cycle.

(8) HdmiWrapper: an interface controller that converts game bitmap data into HDMI video signals. It is mainly used to convert the pixel data (bitmap_data) generated by the game logic into TMDS differential signals compliant with the HDMI standard, so that the game image can be displayed on an HDMI monitor.

(9) UartWrapper: a module that transmits the game bitmap data through the UART interface. It is mainly used to encode the internal 32x16 bitmap data (bitmap_data) into

a byte stream in a specific format and send it through an asynchronous serial port to an external device, such as a PC or another controller. This module supports a customizable baud rate and uses a state machine to complete the transmission of the data packet.

2. Explanation of Code Logic for Each Module:

（1） Top:

As the central hub, it passes variables among the submodules. It also includes a running LED indicator, which can represent the current status and the pressed key.

```
module Top #(
    parameter CLK_FREQ = 32'd50_000_000
) (
    input           clk,
    input           rstn,
    input     [6:0]  keys,
    output          uart_tx,
    output reg [7:0]  led,
    output    [3:0]  seg_sel,
    output    [7:0]  seg_data,
    // HDMI RELATE SIGNALS
    input           clk_27M         ,
    output          tmds_clk_p      ,
    output          tmds_clk_n      ,
    output [2:0]     tmds_data_p     ,
    output [2:0]     tmds_data_n     ,
    output          iic_scl         ,
    inout           iic_sda         ,
    output          rstn_out
);

//============================================================
==============
// internal signals
//============================================================
==============

parameter AREA_ROW = 32;
parameter AREA_COL = 16;
```

```
parameter ROW_ADDR_W = 5;
parameter COL_ADDR_W = 4;

wire [ROW_ADDR_W-1:0] hdmi_bitmap_row;
wire [AREA_COL*2-1:0] hdmi_bitmap_data;

wire [ROW_ADDR_W-1:0] uart_bitmap_row;
wire [AREA_COL*2-1:0] uart_bitmap_data;


//============================================================
==============
// PLL, generate 148.5M & 10M clk
//============================================================
==============
wire pll_lock0;
wire pll_lock1;
wire pll_lock = pll_lock0 & pll_lock1;
wire clk_10m;
wire pixclk;
wire pixclk_5x;

pll_0 u_pll_0(
  .clkin1     (clk       ),//50MHz
  .clkout0    (clk_10m    ),//10MHz
  .pll_lock   (pll_lock0  )
);

pll_1 u_pll_1 (
  .clkin1     (clk_27M    ),// input 27M
  .pll_lock   (pll_lock1  ),// output
  .clkout0    (pixclk     ),// output 74.25M
  .clkout1    (pixclk_5x  ) // output 371.25M
);

//============================================================
==============
// HDMI Wrapper, display output
```

```
//=============================================================
==============
wire hdmi_init_over;

HdmiWrapper u_hdmi (
   .rstn(pll_lock)          , // pll lock
   .cfg_clk(clk_10m)        , // 10Mhz
   .iic_scl(iic_scl)        ,
   .iic_sda(iic_sda)        ,
   .init_over(hdmi_init_over) ,
   .rstn_out(rstn_out)       ,
                            // HDMI output
   .pix_clk(pixclk)         , // input
   .pix_clk_5x(pixclk_5x)    , // input
   .tmds_clk_p(tmds_clk_p)   , // output
   .tmds_clk_n(tmds_clk_n)   , // output
   .tmds_data_p(tmds_data_p)  , // output [2:0]
   .tmds_data_n(tmds_data_n)  , // output [2:0]
   .bitmap_row(hdmi_bitmap_row),
   .bitmap_data(hdmi_bitmap_data)
);



//=============================================================
==============
// Key debounce & filter
//=============================================================
==============

wire [6:0] keys_stable;

reg  [6:0] keys_d1 = 7'h7f; // keys_stable with 1 clock delay

wire pressed_left         = keys_d1[1] & (~keys_stable[1]);
wire pressed_right        = keys_d1[0] & (~keys_stable[0]);
wire pressed_down         = keys_d1[2] & (~keys_stable[2]);
wire pressed_up           = keys_d1[3] & (~keys_stable[3]);
wire pressed_switch       = keys_d1[4] & (~keys_stable[4]);
```

```
wire pressed_fall_down     = keys_d1[5] & (~keys_stable[5]);
wire pressed_reserverd     = keys_d1[6] & (~keys_stable[6]);

KeyDebounce #(
   .CLK_FREQ(CLK_FREQ),
   .KEY_CNT(7)
) u_key (
   .clk(clk),
   .keys(keys[6:0]),
   .keys_stable(keys_stable)
);

always @(posedge clk) begin
   keys_d1 <= keys_stable;
end



//=============================================================
==============
// Game wrapper, including core logic
//=============================================================
==============
wire game_over;
wire [9:0] game_score;

Wrapper #(
   .AREA_ROW(AREA_ROW),
   .AREA_COL(AREA_COL),
   .ROW_ADDR_W(ROW_ADDR_W),
   .COL_ADDR_W(COL_ADDR_W)
) u_wrapper (
   .clk(clk),
   .rstn(rstn),
   .pressed_left(pressed_left),     // key event
   .pressed_right(pressed_right),   //
   .pressed_down(pressed_down),     //
   .pressed_up(pressed_up),         //
   .pressed_switch(pressed_switch),
```

```
   .pressed_fall_down(pressed_fall_down),
   .pressed_reserverd(pressed_reserverd),
   .r1_row(hdmi_bitmap_row),  // output channel #1
   .r1_data(hdmi_bitmap_data),//
   .r2_row(uart_bitmap_row),  // output channel #2
   .r2_data(uart_bitmap_data),//
   .game_score(game_score),
   .game_over(game_over)
);


//=============================================================
==============
// uart wrapper, bitmap data send to PC
//=============================================================
==============

UartWrapper #(
   .AREA_ROW(AREA_ROW),
   .AREA_COL(AREA_COL),
   .ROW_ADDR_W(ROW_ADDR_W),
   .COL_ADDR_W(COL_ADDR_W),
   .CLK_FREQ(CLK_FREQ)
) u_uart (
   .clk(clk),
   .rstn(rstn),
   .uart_tx(uart_tx),
   .bitmap_row(uart_bitmap_row),
   .bitmap_data(uart_bitmap_data)
);


//=============================================================
==============
// seg display for game score
//=============================================================
==============
```

```
SegWrapper #(
   .CLK_FREQ(CLK_FREQ)
) u_seg (
   .clk(clk),
   .rstn(rstn),
   .game_score(game_score),
   .seg_sel(seg_sel),
   .seg_data(seg_data)
);



//=============================================================
==============
// LED as status
//=============================================================
==============

reg [25:0] clk_cnt = 26'd0;
wire game_running = (clk_cnt == (CLK_FREQ-1)) ? 1'b1 : 1'b0;

always @(posedge clk) begin
   if (~rstn) begin
      clk_cnt <= 26'd0;
   end
   else if (clk_cnt == (CLK_FREQ-1)) begin
      clk_cnt <= 26'd0;
   end
   else begin
      clk_cnt <= clk_cnt + 1'b1;
   end
end

always @(posedge clk) begin
   if (~rstn) begin
      led[7:0] <= 8'h01;
   end
   else if (~(&keys_stable)) begin
      led[7:0] <= {1'b0, ~keys_stable};
```

```
   end
   else if (game_running) begin
      if (game_over) begin
         led <= (led == 8'hff) ? 8'h00 : 8'hff;
      end
      else begin
         led[7:0] <= {led[6:0], led[7]};
      end
   end
end

endmodule
```

（2） Wrapper:

It connects the variable transfer between Bitmap and BlockController and implements the information interaction between the moving block and the board. In principle, score statistics could be implemented here, but I chose to implement them in Bitmap.

```
module Wrapper #(
   parameter AREA_ROW = 32,
   parameter AREA_COL = 16,
   parameter ROW_ADDR_W = 5,
   parameter COL_ADDR_W = 4
) (
   input                          clk,
   input                          rstn,
   input                          pressed_left,  // key event
   input                          pressed_right, //
   input                          pressed_up,    //
   input                          pressed_down,  //
   input                          pressed_switch,
   input                          pressed_fall_down,
   input                          pressed_reserverd,
   input    [ROW_ADDR_W-1:0]         r1_row,       // output
channel #1
   output   [AREA_COL*2-1:0]         r1_data,      //        :
data
   input    [ROW_ADDR_W-1:0]         r2_row,       // output
```

```
channel #1
   output   [AREA_COL*2-1:0]       r2_data,      //        :
data
   output                          game_over,
   output   [9:0]                   game_score
);



//=============================================================
==============
// wire and reg in the module
//=============================================================
==============

wire                           falling_update;

wire    [ROW_ADDR_W-1:0]       mv_blk_row;    // current moving
block
wire    [COL_ADDR_W-1:0]       mv_blk_col;    //           :
top-left position
wire    [15:0]                  mv_blk_data;   //           : block
4x4 bitmap
wire                           mv_down_enable;    //           : is
still on active
//wire    [ROW_ADDR_W-1:0]       tst_blk_row;    // testing
block
//wire    [COL_ADDR_W-1:0]       tst_blk_col;    //           :
top-left position
//wire    [15:0]                  tst_blk_data;   //           :
block 4x4 bitmap
//wire                           tst_blk_overl;  //



//=============================================================
==============
// game score
//=============================================================
==============
```

```
//reg [9:0] game_score_r = 0;

//assign game_score = game_score_r;

// TODO - add your logic

//I don't need that...



//================================================================
==============
// connnected sub-modules
//================================================================
==============

Bitmap #(
   .AREA_ROW(AREA_ROW),
   .AREA_COL(AREA_COL),
   .ROW_ADDR_W(ROW_ADDR_W),
   .COL_ADDR_W(COL_ADDR_W)
) u_bitmap (
   .clk(clk),
   .rstn(rstn),
   .falling_update(falling_update),
   .game_over(game_over),
   .mv_blk_row(mv_blk_row),
   .mv_blk_col(mv_blk_col),
   .mv_blk_data(mv_blk_data),
   .mv_down_enable(mv_down_enable),
   //.tst_blk_row(tst_blk_row),
   //.tst_blk_col(tst_blk_col),
   //.tst_blk_data(tst_blk_data),
   //.tst_blk_overl(tst_blk_overl),
   .can_move_down(can_move_down),
   .can_move_left(can_move_left),
   .can_move_right(can_move_right),
   .can_rotate(can_rotate),
   .can_move_up(can_move_up),
```

```
   .r1_row(r1_row),
   .r1_data(r1_data),
   .r2_row(r2_row),
   .r2_data(r2_data),
   // but I do need this...
   .game_score(game_score)
);


BlockController #(
   .AREA_ROW(AREA_ROW),
   .AREA_COL(AREA_COL),
   .ROW_ADDR_W(ROW_ADDR_W),
   .COL_ADDR_W(COL_ADDR_W)
) u_block_ctrl (
   .clk(clk),
   .rstn(rstn),
   .pressed_left(pressed_left),
   .pressed_right(pressed_right),
   .pressed_up(pressed_up),
   .pressed_down(pressed_down),
   .pressed_switch(pressed_switch),
   .pressed_fall_down(pressed_fall_down),
   .pressed_reserverd(pressed_reserverd),
   .falling_update(falling_update),
   .mv_blk_row(mv_blk_row),
   .mv_blk_col(mv_blk_col),
   .mv_blk_data(mv_blk_data),
   .mv_down_enable(mv_down_enable),
   //.tst_blk_row(tst_blk_row),
   //.tst_blk_col(tst_blk_col),
   //.tst_blk_data(tst_blk_data),
   //.tst_blk_overl(tst_blk_overl),
   .can_move_down(can_move_down),
   .can_move_left(can_move_left),
   .can_move_right(can_move_right),
   .can_rotate(can_rotate),
   .can_move_up(can_move_up)
```

```
);

endmodule
```

a) Bitmap:

The most important part of the TODO section is to handle bottom collision, line clearing, and scoring, as well as to determine the feasibility of each type of movement.

For the fixed-block section, the main tasks are to handle bottom collision, line clearing, and scoring.

i. Bottom collision means that the block can no longer move down, i.e., mv_down_enable == 0. If it is not in the top row, the moving block is converted into a fixed block at this time. If it is in the top row, the game ends.

ii. Line clearing means that when the row is 16'hffff, the rows above it are shifted downward by one row starting from that row.

iii. Scoring: when a line clear occurs and the top row is being processed, the score must be handled. Differential scoring is required because, when line clearing is processed, only one row is actually cleared at each moment, while the number of cleared rows accumulates.

```
//============================================================
==============
// fixed blocks
//============================================================
==============

//reg [AREA_ROW : 0] row_mov;// bitmap row-shift flag

generate
integer i,k;
always @(posedge clk) begin
   for (i = 0; i < AREA_ROW; i = i+1) begin
      if (i > 0) begin
         if (~rstn) begin
            // for debug
            if (i < AREA_ROW - 4)
               bitmap_h[i] <= 16'd0;
            else
```

```
            bitmap_h[i] <= 16'hfc_3f;

        score=0;
      end
      else begin
        // TODO

        // bottom collision
        if (mv_down_enable==0) begin
          for (k=0;k<AREA_COL;k=k+1) begin
            if(bitmap_l[i][k]==1)begin
              bitmap_h[i][k] <= 1;
            end
          end
        end
        // line clearing
        if (bitmap_h[i] == 16'hffff) begin
          for(k = i; k > 1; k = k-1) begin
            bitmap_h[k] <= bitmap_h[k-1];
          end
          eliminate_cnt=eliminate_cnt+1;
        end

      end
    end
    else begin // when i = 0
      if (~rstn) begin
        bitmap_h[i] <= 0;
      end
      else begin
        // TODO
        // bottom collision
        if (mv_down_enable==0) begin
          if (bitmap_h[0]!=0) begin
            game_over_cao<=1;
          end
        end
        // scoring
```

```
          if (eliminate_cnt==1) begin
             score = score + 1;
          end
          if (eliminate_cnt==2) begin
             score = score + 10-1;
          end
          if (eliminate_cnt==3) begin
             score = score + 66-10;
          end
          if (eliminate_cnt==4) begin
             score = score + 100-66;
          end
          eliminate_cnt=0;

        end
      end
   end
end

endgenerate
```

Movement-logic judgment:

The board after mv_blk hypothetically moves is simulated through bitmap_tmp. If the board becomes invalid after the move, this type of movement is not allowed. For left and upward movement, it is necessary to determine whether the coordinates go out of bounds and then determine whether overlap occurs. For right and downward movement, it is necessary to determine whether there is already a block in the rightmost or bottom row, in which case movement is not allowed, and then determine whether overlap occurs. For rotation, it is only necessary to determine whether overlap occurs.

```
//============================================================
==============
// movement judgment logic
//============================================================
==============

reg [AREA_COL - 1 : 0] bitmap_tmp [AREA_ROW  : 0];// temporary
```

```
bitmap

generate
integer s;
integer t;
integer tag;// current tag

always @(*) begin
   if(bitmap_l[AREA_ROW-1]!=0)begin
       can_move_down_r=0;
   end
   else begin
      bitmap_tmp[0]=16'h0000;

      // down
      for(s = 1; s < AREA_ROW+1; s = s+1)begin
         bitmap_tmp[s]=bitmap_l[s-1];
      end
      tag=0;
      for(s = 0; s < AREA_ROW; s = s+1)begin
         for(t = 0; t < AREA_COL; t = t+1)begin
            if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
               tag=1;
            end
         end
      end
      if(bitmap_tmp[AREA_ROW]!=0)begin// passes through the
bottom boundary
         tag=1;
      end
      can_move_down_r=(tag==0);

      // left
      tag=0;
      for(s = 0; s < AREA_ROW; s = s+1)begin
         bitmap_tmp[s]=bitmap_l[s]<<1;
         if(bitmap_l[s][0]!=0)begin
            tag=1;
```

```
      end
    end
    //if(mv_blk_col==0)
       //tag=1;
    for(s = 0; s < AREA_ROW; s = s+1)begin
      for(t = 0; t < AREA_COL; t = t+1)begin
        if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
          tag=1;
        end
      end
    end
    can_move_left_r=(tag==0);

    // right
    tag=0;
    for(s = 0; s < AREA_ROW; s = s+1)begin
      bitmap_tmp[s]=bitmap_l[s]>>1;
      if(bitmap_l[s][AREA_COL-1]!=0)begin
        tag=1;
      end
    end
    for(s = 0; s < AREA_ROW; s = s+1)begin
      for(t = 0; t < AREA_COL; t = t+1)begin
        if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
          tag=1;
        end
      end
    end
    can_move_right_r=(tag==0);

    // rotate
    for (s = 0; s < AREA_ROW; s = s+1) begin
      if (s == mv_blk_row) begin
        bitmap_tmp[s] = {12'b0,
mv_blk_data[12],mv_blk_data[8],mv_blk_data[4],mv_blk_data[0]}
<< mv_blk_col;
      end
      else if (s == mv_blk_row + 1) begin
```

```
          bitmap_tmp[s] = {12'b0,
mv_blk_data[13],mv_blk_data[9],mv_blk_data[5],mv_blk_data[1]}
<< mv_blk_col;
        end
        else if (s == mv_blk_row + 2) begin
          bitmap_tmp[s] = {12'b0,
mv_blk_data[14],mv_blk_data[10],mv_blk_data[6],mv_blk_data[2]}
<< mv_blk_col;
        end
        else if (s == mv_blk_row + 3) begin
          bitmap_tmp[s] = {12'b0,
mv_blk_data[15],mv_blk_data[11],mv_blk_data[7],mv_blk_data[3]}
<< mv_blk_col;
        end
        else begin
          bitmap_tmp[s] = 0;
        end
      end
      tag=0;
      for(s = 0; s < AREA_ROW; s = s+1)begin
        for(t = 0; t < AREA_COL; t = t+1)begin
          if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
            tag=1;
          end
        end
      end
      can_rotate_r=(tag==0);

      // up
      bitmap_tmp[AREA_ROW-1]=16'h0000;
      for(s = 0; s < AREA_ROW-1; s = s+1)begin
        bitmap_tmp[s]=bitmap_l[s+1];
      end
      tag=0;
      for(s = 0; s < AREA_ROW; s = s+1)begin
        for(t = 0; t < AREA_COL; t = t+1)begin
          if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
            tag=1;
```

```
          end
        end
     end
     if(bitmap_l[0]!=0)begin// would pass through the bottom
boundary
        tag=1;
     end
     //if(mv_blk_row==0)
         //tag=1;
     can_move_up_r=(tag==0);
   end
end
endgenerate


// connect to existing signals
assign mv_down_enable = can_move_down;
```

The following is the complete Bitmap code.

```
`define __DEBUG__

module Bitmap #(
   parameter AREA_ROW = 32,
   parameter AREA_COL = 16,
   parameter ROW_ADDR_W = 5,
   parameter COL_ADDR_W = 4
)
(
   input                          clk,
   input                          rstn,
   input                          falling_update,// block falling
signal
   output                         game_over,     // game over
   input    [ROW_ADDR_W-1:0]        mv_blk_row,    // current
moving block
   input    [COL_ADDR_W-1:0]        mv_blk_col,    //           :
top-left position
   input    [15:0]                  mv_blk_data,   //           :
```

```
block 4x4 bitmap
   output                        mv_down_enable,//           : is
still on active (enable to move down)
   //input    [ROW_ADDR_W-1:0]       tst_blk_row,   // testing
block
   //input    [COL_ADDR_W-1:0]       tst_blk_col,
//           : top-left position
   //input    [15:0]               tst_blk_data,  //           :
block 4x4 bitmap
   //output                       tst_blk_overl, //           :
is overlapping with current fixed blocks
   // movement judgment result signals
   output can_move_down,
   output can_move_left,
   output can_move_right,
   output can_rotate,
   output can_move_up,
   input    [ROW_ADDR_W-1:0]       r1_row,       // output
channel #1
   output   [AREA_COL*2-1:0]       r1_data,      //        :
data
   input    [ROW_ADDR_W-1:0]       r2_row,       // output
channel #1
   output   [AREA_COL*2-1:0]       r2_data,       //        :
data
   output   [9:0]                game_score
);


//=============================================================
==============
// bitmap content
//=============================================================
==============

reg [AREA_COL - 1 : 0] bitmap_h [AREA_ROW - 1 : 0];//fixed
blocks
reg [AREA_COL - 1 : 0] bitmap_l [AREA_ROW - 1 : 0];//active
```

```
blocks


wire [3:0] top4_row_overlap;
assign top4_row_overlap[0] = |(bitmap_l[0] & bitmap_h[0]);
assign top4_row_overlap[1] = |(bitmap_l[1] & bitmap_h[1]);
assign top4_row_overlap[2] = |(bitmap_l[2] & bitmap_h[2]);
assign top4_row_overlap[3] = |(bitmap_l[3] & bitmap_h[3]);



`ifdef __DEBUG__
wire [AREA_COL - 1 : 0] bitmap_h0 = bitmap_h[0];
wire [AREA_COL - 1 : 0] bitmap_h1 = bitmap_h[1];
wire [AREA_COL - 1 : 0] bitmap_h2 = bitmap_h[2];
wire [AREA_COL - 1 : 0] bitmap_h3 = bitmap_h[3];
wire [AREA_COL - 1 : 0] bitmap_h24 = bitmap_h[24];
wire [AREA_COL - 1 : 0] bitmap_h25 = bitmap_h[25];
wire [AREA_COL - 1 : 0] bitmap_h26 = bitmap_h[26];
wire [AREA_COL - 1 : 0] bitmap_h27 = bitmap_h[27];
wire [AREA_COL - 1 : 0] bitmap_h28 = bitmap_h[28];
wire [AREA_COL - 1 : 0] bitmap_h29 = bitmap_h[29];
wire [AREA_COL - 1 : 0] bitmap_h30 = bitmap_h[30];
wire [AREA_COL - 1 : 0] bitmap_h31 = bitmap_h[31];

wire [AREA_COL - 1 : 0] bitmap_l0 = bitmap_l[0];
wire [AREA_COL - 1 : 0] bitmap_l1 = bitmap_l[1];
wire [AREA_COL - 1 : 0] bitmap_l2 = bitmap_l[2];
wire [AREA_COL - 1 : 0] bitmap_l3 = bitmap_l[3];
wire [AREA_COL - 1 : 0] bitmap_l10 = bitmap_l[10];
wire [AREA_COL - 1 : 0] bitmap_l11 = bitmap_l[11];
wire [AREA_COL - 1 : 0] bitmap_l12 = bitmap_l[12];
wire [AREA_COL - 1 : 0] bitmap_l13 = bitmap_l[13];
wire [AREA_COL - 1 : 0] bitmap_l20 = bitmap_l[20];
wire [AREA_COL - 1 : 0] bitmap_l21 = bitmap_l[21];
wire [AREA_COL - 1 : 0] bitmap_l22 = bitmap_l[22];
wire [AREA_COL - 1 : 0] bitmap_l23 = bitmap_l[23];
wire [AREA_COL - 1 : 0] bitmap_l28 = bitmap_l[28];
```

```
wire [AREA_COL - 1 : 0] bitmap_l29 = bitmap_l[29];
wire [AREA_COL - 1 : 0] bitmap_l30 = bitmap_l[30];
wire [AREA_COL - 1 : 0] bitmap_l31 = bitmap_l[31];
`endif

reg [2:0] eliminate_cnt;  // cleared-line count
reg [9:0] score;
reg [1:0] game_over_cao;

assign game_score=score;
assign game_over = (|top4_row_overlap) || game_over_cao;

reg can_move_down_r;
reg can_move_left_r;
reg can_move_right_r;
reg can_rotate_r;
assign can_move_down=can_move_down_r;
assign can_move_left=can_move_left_r;
assign can_move_right=can_move_right_r;
assign can_rotate=can_rotate_r;

reg can_move_up_r;
assign can_move_up=can_move_up_r;

//================================================================
==============
// fixed blocks
//================================================================
==============

//reg [AREA_ROW : 0] row_mov;// bitmap row-shift flag

generate
integer i,k;
always @(posedge clk) begin
   for (i = 0; i < AREA_ROW; i = i+1) begin
     if (i > 0) begin
       if (~rstn) begin
```

```
        // for debug
        if (i < AREA_ROW - 4)
          bitmap_h[i] <= 16'd0;
        else
          bitmap_h[i] <= 16'hfc_3f;

        score=0;
      end
      else begin
        // TODO

        // bottom collision
        if (mv_down_enable==0) begin
          for (k=0;k<AREA_COL;k=k+1) begin
            if(bitmap_l[i][k]==1)begin
              bitmap_h[i][k] <= 1;
            end
          end
        end
        // line clearing
        if (bitmap_h[i] == 16'hffff) begin
          for(k = i; k > 1; k = k-1) begin
            bitmap_h[k] <= bitmap_h[k-1];
          end
          eliminate_cnt=eliminate_cnt+1;
        end

      end
    end
    else begin // when i = 0
      if (~rstn) begin
        bitmap_h[i] <= 0;
      end
      else begin
        // TODO
        // bottom collision
        if (mv_down_enable==0) begin
          if (bitmap_h[0]!=0) begin
```

```
              game_over_cao<=1;
            end
          end
          // scoring
          if (eliminate_cnt==1) begin
            score = score + 1;
          end
          if (eliminate_cnt==2) begin
            score = score + 10-1;
          end
          if (eliminate_cnt==3) begin
            score = score + 66-10;
          end
          if (eliminate_cnt==4) begin
            score = score + 100-66;
          end
          eliminate_cnt=0;

        end
      end
   end
end

endgenerate


//================================================================
==============
// moving block, DONT EDIT THIS PART
//================================================================
==============

generate
genvar j;
for (j = 0; j < AREA_ROW; j = j+1) begin
   always @(*) begin
      if (~rstn) begin
        bitmap_l[j] = 0;
```

```
     end
     else begin
        if (j == mv_blk_row) begin
          bitmap_l[j] = {12'b0, mv_blk_data[3:0]} <<
mv_blk_col;
        end
        else if (j == mv_blk_row + 1) begin
          bitmap_l[j] = {12'b0, mv_blk_data[7:4]} <<
mv_blk_col;
        end
        else if (j == mv_blk_row + 2) begin
          bitmap_l[j] = {12'b0, mv_blk_data[11:8]} <<
mv_blk_col;
        end
        else if (j == mv_blk_row + 3) begin
          bitmap_l[j] = {12'b0, mv_blk_data[15:12]} <<
mv_blk_col;
        end
        else begin
          bitmap_l[j] = 0;
        end
     end
   end
end
endgenerate


//============================================================
==============
// output channel #1, DONT EDIT THIS PART
//============================================================
==============

wire [AREA_COL-1:0] r1_data_h = bitmap_h[r1_row];
wire [AREA_COL-1:0] r1_data_l = bitmap_l[r1_row];

assign r1_data = {r1_data_h, r1_data_l};
```

```
//=============================================================
==============
// output channel #2, DONT EDIT THIS PART
//=============================================================
==============

reg [2*AREA_COL-1:0] r2_data_r;
assign r2_data = r2_data_r;

always @(*) begin
   r2_data_r <= {bitmap_h[r2_row], bitmap_l[r2_row]};
end


//=============================================================
==============
// TODO - add your codes below
//=============================================================
==============

//=============================================================
==============
// movement judgment logic
//=============================================================
==============

reg [AREA_COL - 1 : 0] bitmap_tmp [AREA_ROW  : 0];// temporary
bitmap

generate
integer s;
integer t;
integer tag;// current tag

always @(*) begin
   if(bitmap_l[AREA_ROW-1]!=0)begin
      can_move_down_r=0;
```

```
  end
  else begin
    bitmap_tmp[0]=16'h0000;

    // down
    for(s = 1; s < AREA_ROW+1; s = s+1)begin
      bitmap_tmp[s]=bitmap_l[s-1];
    end
    tag=0;
    for(s = 0; s < AREA_ROW; s = s+1)begin
      for(t = 0; t < AREA_COL; t = t+1)begin
        if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
          tag=1;
        end
      end
    end
    if(bitmap_tmp[AREA_ROW]!=0)begin// passes through the
bottom boundary
      tag=1;
    end
    can_move_down_r=(tag==0);

    // left
    tag=0;
    for(s = 0; s < AREA_ROW; s = s+1)begin
      bitmap_tmp[s]=bitmap_l[s]<<1;
      if(bitmap_l[s][0]!=0)begin
        tag=1;
      end
    end
    //if(mv_blk_col==0)
       //tag=1;
    for(s = 0; s < AREA_ROW; s = s+1)begin
      for(t = 0; t < AREA_COL; t = t+1)begin
        if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
          tag=1;
        end
      end
```

```
    end
    can_move_left_r=(tag==0);

    // right
    tag=0;
    for(s = 0; s < AREA_ROW; s = s+1)begin
      bitmap_tmp[s]=bitmap_l[s]>>1;
      if(bitmap_l[s][AREA_COL-1]!=0)begin
        tag=1;
      end
    end
    for(s = 0; s < AREA_ROW; s = s+1)begin
      for(t = 0; t < AREA_COL; t = t+1)begin
        if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
          tag=1;
        end
      end
    end
    can_move_right_r=(tag==0);

    // rotate
    for (s = 0; s < AREA_ROW; s = s+1) begin
      if (s == mv_blk_row) begin
        bitmap_tmp[s] = {12'b0,
mv_blk_data[12],mv_blk_data[8],mv_blk_data[4],mv_blk_data[0]}
<< mv_blk_col;
      end
      else if (s == mv_blk_row + 1) begin
        bitmap_tmp[s] = {12'b0,
mv_blk_data[13],mv_blk_data[9],mv_blk_data[5],mv_blk_data[1]}
<< mv_blk_col;
      end
      else if (s == mv_blk_row + 2) begin
        bitmap_tmp[s] = {12'b0,
mv_blk_data[14],mv_blk_data[10],mv_blk_data[6],mv_blk_data[2]}
<< mv_blk_col;
      end
      else if (s == mv_blk_row + 3) begin
```

```
          bitmap_tmp[s] = {12'b0,
mv_blk_data[15],mv_blk_data[11],mv_blk_data[7],mv_blk_data[3]}
<< mv_blk_col;
        end
        else begin
          bitmap_tmp[s] = 0;
        end
      end
      tag=0;
      for(s = 0; s < AREA_ROW; s = s+1)begin
        for(t = 0; t < AREA_COL; t = t+1)begin
          if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
            tag=1;
          end
        end
      end
      can_rotate_r=(tag==0);

      // up
      bitmap_tmp[AREA_ROW-1]=16'h0000;
      for(s = 0; s < AREA_ROW-1; s = s+1)begin
        bitmap_tmp[s]=bitmap_l[s+1];
      end
      tag=0;
      for(s = 0; s < AREA_ROW; s = s+1)begin
        for(t = 0; t < AREA_COL; t = t+1)begin
          if(bitmap_tmp[s][t]==1&bitmap_h[s][t]==1)begin
            tag=1;
          end
        end
      end
      if(bitmap_l[0]!=0)begin// would pass through the bottom
boundary
        tag=1;
      end
      //if(mv_blk_row==0)
         //tag=1;
      can_move_up_r=(tag==0);
```

```
   end
end
endgenerate


// connect to existing signals
assign mv_down_enable = can_move_down;

endmodule
```

b) BlockController:

It processes each type of key operation and updates mv_blk according to the operation. The more difficult cases are leftward or upward movement for blocks that do not fully fill the 4x4 range. Our strategy is to prioritize movement within the 4x4 range rather than prioritizing movement of the entire 4x4 block, thereby avoiding coordinates going out of bounds. We need to enumerate the movement patterns of blocks that do not fully fill the 4x4 range while their coordinates remain unchanged. In addition, when the bottom-collision signal is 1, we need to generate a new moving block.
Bonus: the reserved key is treated as an upward-movement key. The moving block can move upward.

```
module BlockController #(
   parameter AREA_ROW = 32,
   parameter AREA_COL = 16,
   parameter ROW_ADDR_W = 5,
   parameter COL_ADDR_W = 4
) (
   input                       clk,
   input                       rstn,
   input                       pressed_left,  // key event
   input                       pressed_right, //
   input                       pressed_up,    //
   input                       pressed_down,  //
   input                       pressed_switch,
   input                       pressed_fall_down,
   input                       pressed_reserverd,
   output                      falling_update,// block falling
```

```
signal
   output   [ROW_ADDR_W-1:0]       mv_blk_row,    // current
moving block
   output   [COL_ADDR_W-1:0]       mv_blk_col,    //           :
top-left position
   output   [15:0]                 mv_blk_data,   //           :
block 4x4 bitmap
   input                           mv_down_enable,//           :
feedback: is still on active (enable to move down)
   //output   [ROW_ADDR_W-1:0]       tst_blk_row,   // testing
block
   //output   [COL_ADDR_W-1:0]       tst_blk_col,
//           : top-left position
   //output   [15:0]                 tst_blk_data,  //           :
block 4x4 bitmap
   //input                           tst_blk_overl  //           :
feedback: is overlapping with current fixed blocks
   // movement judgment result signals
   input can_move_down,
   input can_move_left,
   input can_move_right,
   input can_rotate,
   input can_move_up
);

//=============================================================
==============
// define basic block, 4 x 4
//=============================================================
==============

// square block
reg [15:0] BLK_0 = {
   4'b0_0_0_0,
   4'b0_1_1_0,
   4'b0_1_1_0,
   4'b0_0_0_0
};
```

```
reg [15:0] BLK_0_s = {
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b0_1_1_0,
   4'b0_1_1_0
};

reg [15:0] BLK_0_z = {
   4'b0_0_0_0,
   4'b0_0_1_1,
   4'b0_0_1_1,
   4'b0_0_0_0
};

reg [15:0] BLK_0_sz = {
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b0_0_1_1,
   4'b0_0_1_1
};

// long bar
reg [15:0] BLK_1 = {
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b1_1_1_1
};

reg [15:0] BLK_1_z3 = {
   4'b0_0_0_1,
   4'b0_0_0_1,
   4'b0_0_0_1,
   4'b0_0_0_1
};

reg [15:0] BLK_1_z2 = {
```

```
   4'b0_0_1_0,
   4'b0_0_1_0,
   4'b0_0_1_0,
   4'b0_0_1_0
};

reg [15:0] BLK_1_z1 = {
   4'b0_1_0_0,
   4'b0_1_0_0,
   4'b0_1_0_0,
   4'b0_1_0_0
};

reg [15:0] BLK_1_z0 = {
   4'b1_0_0_0,
   4'b1_0_0_0,
   4'b1_0_0_0,
   4'b1_0_0_0
};

reg [15:0] BLK_1_s3 = {
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b1_1_1_1
};

reg [15:0] BLK_1_s2 = {
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b1_1_1_1,
   4'b0_0_0_0
};

reg [15:0] BLK_1_s1 = {
   4'b0_0_0_0,
   4'b1_1_1_1,
   4'b0_0_0_0,
```

```
   4'b0_0_0_0

};

reg [15:0] BLK_1_s0 = {
   4'b1_1_1_1,
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b0_0_0_0
};

// long bar
reg [15:0] BLK_2 = {
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b1_1_1_1,
   4'b1_1_1_1
};

reg [15:0] BLK_2_z2 = {
   4'b0_0_1_1,
   4'b0_0_1_1,
   4'b0_0_1_1,
   4'b0_0_1_1
};

reg [15:0] BLK_2_z1 = {
   4'b0_1_1_0,
   4'b0_1_1_0,
   4'b0_1_1_0,
   4'b0_1_1_0
};

reg [15:0] BLK_2_z0 = {
   4'b1_1_0_0,
   4'b1_1_0_0,
   4'b1_1_0_0,
   4'b1_1_0_0
```

```
};

reg [15:0] BLK_2_s2 = {
   4'b0_0_0_0,
   4'b0_0_0_0,
   4'b1_1_1_1,
   4'b1_1_1_1
};

reg [15:0] BLK_2_s1 = {
   4'b0_0_0_0,
   4'b1_1_1_1,
   4'b1_1_1_1,
   4'b0_0_0_0
};

reg [15:0] BLK_2_s0 = {
   4'b1_1_1_1,
   4'b1_1_1_1,
   4'b0_0_0_0,
   4'b0_0_0_0
};

// square block
reg [15:0] BLK_3 = {
   4'b1_1_1_1,
   4'b1_1_1_1,
   4'b1_1_1_1,
   4'b1_1_1_1
};

// hollow square
reg [15:0] BLK_4 = {
   4'b1_1_1_1,
   4'b1_0_0_1,
   4'b1_0_0_1,
   4'b1_1_1_1
};
```

```
//=============================================================
==============
// generate next block, 4 x 4
//=============================================================
==============

reg [2:0] nxt_blk_idx = 3'd4;
reg [15:0] nxt_blk_data;


always @(posedge clk) begin
    if (~rstn) begin
        nxt_blk_idx <= 3'd4;
    end else begin
        nxt_blk_idx <= {nxt_blk_idx[1:0], nxt_blk_idx[2] ^
nxt_blk_idx[1]};
    end
end

always @(*) begin
   case(nxt_blk_idx)
     3'd0: nxt_blk_data = BLK_0;
     3'd1: nxt_blk_data = BLK_1;
     3'd2: nxt_blk_data = BLK_2;
     3'd3: nxt_blk_data = BLK_3;
     3'd4: nxt_blk_data = BLK_4;
     3'd5: nxt_blk_data = BLK_0;
     3'd6: nxt_blk_data = BLK_1;
     3'd7: nxt_blk_data = BLK_2;
     default: nxt_blk_data = BLK_0;
   endcase
end

//=============================================================
==============
// test block relate codes, TODO
//=============================================================
```

```
==============

//reg [ROW_ADDR_W-1:0] tst_blk_row_r;
//reg [COL_ADDR_W-1:0] tst_blk_col_r;
//reg [15:0] tst_blk_data_r;

//assign tst_blk_row = tst_blk_row_r;
//assign tst_blk_col = tst_blk_col_r;
//assign tst_blk_data = tst_blk_data_r;

//reg tst_blk_tocheck = 1'b0;

//nothing to do...
//already done in bitmap...
//================================================================
==============
// update current block, based on: falling update, pressed
keys
//================================================================
==============

reg falling_update_r = 1'b0; // falling request signal
reg [ROW_ADDR_W-1:0] mv_blk_row_r;
reg [COL_ADDR_W-1:0] mv_blk_col_r;
reg [15:0] mv_blk_data_r;

assign falling_update = falling_update_r;
assign mv_blk_row = mv_blk_row_r;
assign mv_blk_col = mv_blk_col_r;
assign mv_blk_data = mv_blk_data_r;


always @(posedge clk) begin
   if (~rstn) begin
      // reset falling request
      falling_update_r <= 1'b0;
      // reset current moving block
      mv_blk_row_r <= 0;
```

```
      mv_blk_col_r <= (AREA_COL >> 1) - 2;
      mv_blk_data_r <= nxt_blk_data;
    end
    // 1) when current block falling down
    else if (falling_update) begin
      // case 1a) is collide (not active)
      if (~mv_down_enable) begin
        mv_blk_row_r <= 0;
        mv_blk_col_r <= (AREA_COL >> 1) - 2;
        mv_blk_data_r <= nxt_blk_data;
        falling_update_r <= 1'b0;
      end
      else if (mv_blk_row_r == {ROW_ADDR_W{1'b1}}) begin
        falling_update_r <= 1'b0;
      end
      // case 1b) is still active
      else begin
        mv_blk_row_r <= mv_blk_row_r + 1;
        // Note - need to consider if fall to bottom, and stop
        falling_update_r <= (mv_blk_row_r ==
{ROW_ADDR_W{1'b1}}) ? 1'b0: 1'b1;
      end
    end
    // 2) when has pressed keys - down to bottom
    else if (pressed_fall_down) begin
      falling_update_r <= 1'b1;
    end
    // 3) when has pressed keys - UP (rotate)
    else if (pressed_up) begin
      // TODO

if(mv_blk_data_r!=BLK_0_s&&mv_blk_data_r!=BLK_0_z&&mv_blk_data
_r!=BLK_0_sz)
        mv_blk_data_r <= {
          mv_blk_data[12], mv_blk_data[8], mv_blk_data[4],
mv_blk_data[0],
          mv_blk_data[13], mv_blk_data[9], mv_blk_data[5],
mv_blk_data[1],
```

```
          mv_blk_data[14], mv_blk_data[10], mv_blk_data[6],
mv_blk_data[2],
          mv_blk_data[15], mv_blk_data[11], mv_blk_data[7],
mv_blk_data[3]
        };
  end
  // 4) when has pressed keys - DOWN (1 step)
  else if (pressed_down) begin
    // TODO
    if(can_move_down) begin
      mv_blk_row_r <= mv_blk_row_r + 1;
      mv_blk_col_r <= mv_blk_col_r;
      mv_blk_data_r <= mv_blk_data_r;
    end
  end
  // 5) when has pressed keys - LEFT
  else if (pressed_left) begin
    // TODO
    if (mv_blk_data_r==BLK_2_z0)
      mv_blk_data_r <= BLK_2_z1;
    else if (mv_blk_data_r == BLK_2_z1)
      mv_blk_data_r <= BLK_2_z2;
    else if (mv_blk_data_r == BLK_1_z0)
      mv_blk_data_r <= BLK_1_z1;
    else if (mv_blk_data_r == BLK_1_z1)
      mv_blk_data_r <= BLK_1_z2;
    else if(mv_blk_data_r == BLK_1_z2)
      mv_blk_data_r <= BLK_1_z3;
    else if (mv_blk_data_r == BLK_0)
      mv_blk_data_r <= BLK_0_z;
    else if (mv_blk_data_r == BLK_0_s)
      mv_blk_data_r <= BLK_0_sz;
    else if (can_move_left) begin
      mv_blk_row_r <= mv_blk_row_r;
      mv_blk_col_r <= mv_blk_col_r - 1;
      mv_blk_data_r <= mv_blk_data_r;
    end
  end
```

```
// 6) when has pressed keys - RIGHT
else if (pressed_right) begin
   // TODO
   if (can_move_right) begin
      mv_blk_row_r <= mv_blk_row_r;
      mv_blk_col_r <= mv_blk_col_r + 1;
      mv_blk_data_r <= mv_blk_data_r;
   end
end
// 7) when has pressed keys - active black switch
else if (pressed_switch) begin
   mv_blk_data_r <= nxt_blk_data;
end
// 8) when has pressed keys - reserverd
else if (pressed_reserverd) begin
   if (mv_blk_data_r==BLK_2_s0)
      mv_blk_data_r <= BLK_2_s1;
   else if (mv_blk_data_r == BLK_2_s1)
      mv_blk_data_r <= BLK_2_s2;
   else if (mv_blk_data_r == BLK_1_s0)
      mv_blk_data_r <= BLK_1_z1;
   else if (mv_blk_data_r == BLK_1_s1)
      mv_blk_data_r <= BLK_1_s2;
   else if(mv_blk_data_r == BLK_1_s2)
      mv_blk_data_r <= BLK_1_s3;
   else if (mv_blk_data_r == BLK_0)
      mv_blk_data_r <= BLK_0_s;
   else if (mv_blk_data_r == BLK_0_z)
      mv_blk_data_r <= BLK_0_sz;
   else if (can_move_up) begin
      mv_blk_row_r <= mv_blk_row_r - 1;
      mv_blk_col_r <= mv_blk_col_r;
      mv_blk_data_r <= mv_blk_data_r;
   end
end
//I worked hard and stayed up all night just to add this...
else if (~mv_down_enable) begin
   // reset falling request
```

```
      falling_update_r <= 1'b0;
      // reset current moving block
      mv_blk_row_r <= 0;
      mv_blk_col_r <= (AREA_COL >> 1) - 2;
      mv_blk_data_r <= nxt_blk_data;
   end
end

endmodule
```

（3） SegWrapper:

game_score is converted into sel_num for each digit. The principle is to extract the specific value of each digit of a four-digit decimal number. Then, the value of each digit is used to control whether each segment of the seven-segment display is on or off.

```
`define SEG_SEL_NULL  4'b0000
`define SEG_SEL_0     4'b0001
`define SEG_SEL_1     4'b0010
`define SEG_SEL_2     4'b0100
`define SEG_SEL_3     4'b1000

// `define SEG_FLASH_DUR 26'd49_999

module SegWrapper #(
   parameter CLK_FREQ = 50_000_000,
   parameter SEG_FLASH_DUR = 49_999
) (
   input                        clk,
   input                        rstn,
   input      [9:0]              game_score,
   output     reg [3:0]              seg_sel,
   output     reg [7:0]              seg_data
);

// TODO - NEED TO remove following codes, replaced by your
codes

// counter, triggers the scan_next signal
```

```
parameter SCAN_FREQ = 200;     //scan frequency
parameter SCAN_CLK_CNT = CLK_FREQ /(SCAN_FREQ * 4) - 1;

reg [31:0] clk_cnt = 32'b0;
assign scan_next = (clk_cnt == SCAN_CLK_CNT);

always @(posedge clk or negedge rstn) begin
    // reset
    if (rstn == 1'b0) begin
        clk_cnt <= 32'b0;
    end
    else begin
        // clk counter
        if (clk_cnt == SCAN_CLK_CNT) begin
            clk_cnt <= 32'b0;
        end
        else begin
            clk_cnt <= clk_cnt + 32'b1;
        end
    end
end


// state machine for seg_sel
reg [3:0] next_seg_sel = `SEG_SEL_NULL;

// state machine for seg_sel, 1) state transition, sequential
logic
always @(posedge clk or negedge rstn) begin
    if (rstn == 1'b0) begin
        seg_sel <= `SEG_SEL_NULL;
    end
    else begin
        // TODO - update seg_sel
        seg_sel<=next_seg_sel;
    end
end
```

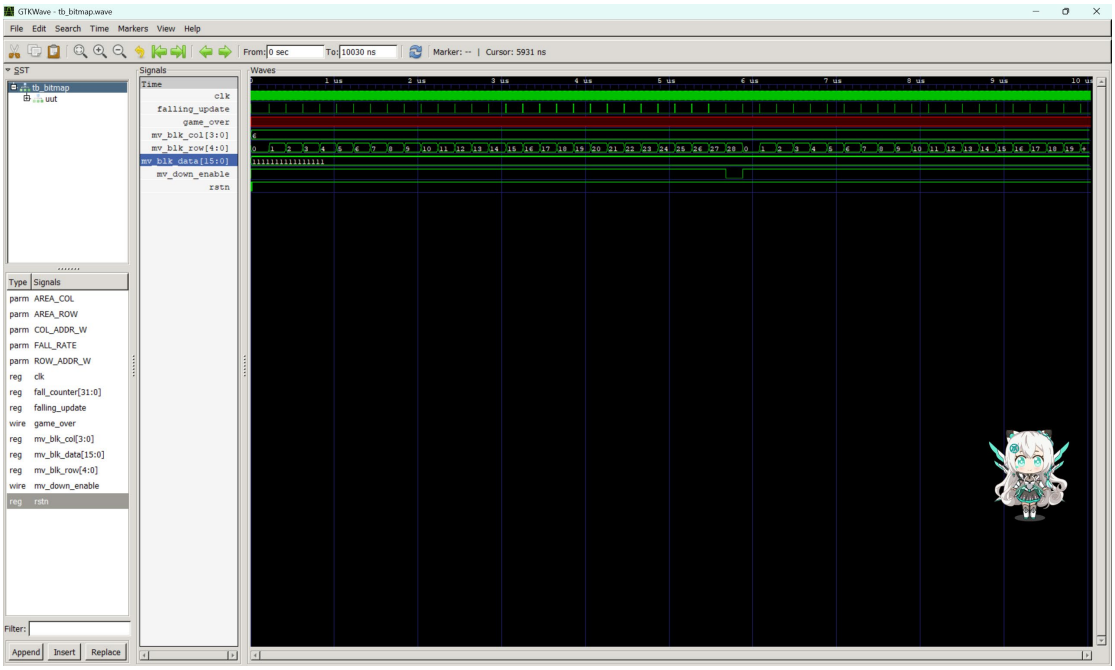
```
// state machine for seg_sel, 2) state switching,
combinational logic
always @(*) begin
    if (scan_next) begin
        // TODO - update next_seg_sel, based on seg_sel, and
inner signal scan_next
        if(seg_sel==4'b0001)begin
            next_seg_sel<=4'b0010;
        end
        else if(seg_sel==4'b0010)begin
            next_seg_sel<=4'b0100;
        end
        else if(seg_sel==4'b0100)begin
            next_seg_sel<=4'b1000;
        end
        else if(seg_sel==4'b1000)begin
            next_seg_sel<=4'b0001;
        end
        else begin
            next_seg_sel<=4'b0001;
        end
    end
    else begin
        next_seg_sel <= seg_sel;
    end
end

// state machine for seg_sel, 3) output sel_num & seg_data,
sequential logic
reg [3:0] sel_num = 4'b0;

always @(posedge clk or negedge rstn)
begin
    if (rstn == 1'b0) begin
        sel_num = 8'b0;
    end
    else begin
        // TODO - udpate sel_num, based on seg_sel
```

```
    case(seg_sel)
        4'b1000:sel_num = game_score%10;
        4'b0100:sel_num = (game_score-game_score/100*100)/10;
        4'b0010:sel_num = (game_score-game_score/1000*1000)/100;
       default:sel_num = game_score/1000;
    endcase
    end
end

// combination logic, output: seg_data, based on sel_num
always @(*)begin
    case(sel_num)
        4'd0: seg_data = 8'b1100_0000;
        4'd1: seg_data = 8'b1111_1001;
        4'd2: seg_data = 8'b1010_0100;
        4'd3: seg_data = 8'b1011_0000;
        4'd4: seg_data = 8'b1001_1001;
        4'd5: seg_data = 8'b1001_0010;
        4'd6: seg_data = 8'b1000_0010;
        4'd7: seg_data = 8'b1111_1000;
        4'd8: seg_data = 8'b1000_0000;
        4'd9: seg_data = 8'b1001_0000;
        4'ha: seg_data = 8'b1000_1000;
        4'hb: seg_data = 8'b1000_0011;
        4'hc: seg_data = 8'b1100_0110;
        4'hd: seg_data = 8'b1010_0001;
        4'he: seg_data = 8'b1000_0110;
        4'hf: seg_data = 8'b1000_1110;
       default:seg_data = 8'b1100_0000;
    endcase
end

endmodule
```

3. Module Testbench:

（1） Bitmap:

The result meets expectations.




```
`timescale 1ns / 1ps

module tb_bitmap ();

//================================================================
==============
// Game wrapper, including core logic
//================================================================
==============
parameter AREA_ROW = 32;
parameter AREA_COL = 16;
parameter ROW_ADDR_W = 5;
parameter COL_ADDR_W = 4;

reg clk = 1'b0;
reg rstn = 1'b1;

wire game_over;
reg falling_update = 1'b0; // falling update signal

reg    [ROW_ADDR_W-1:0]       mv_blk_row = 0;   // current
moving block
reg    [COL_ADDR_W-1:0]       mv_blk_col = 6;   //           :
top-left position
```

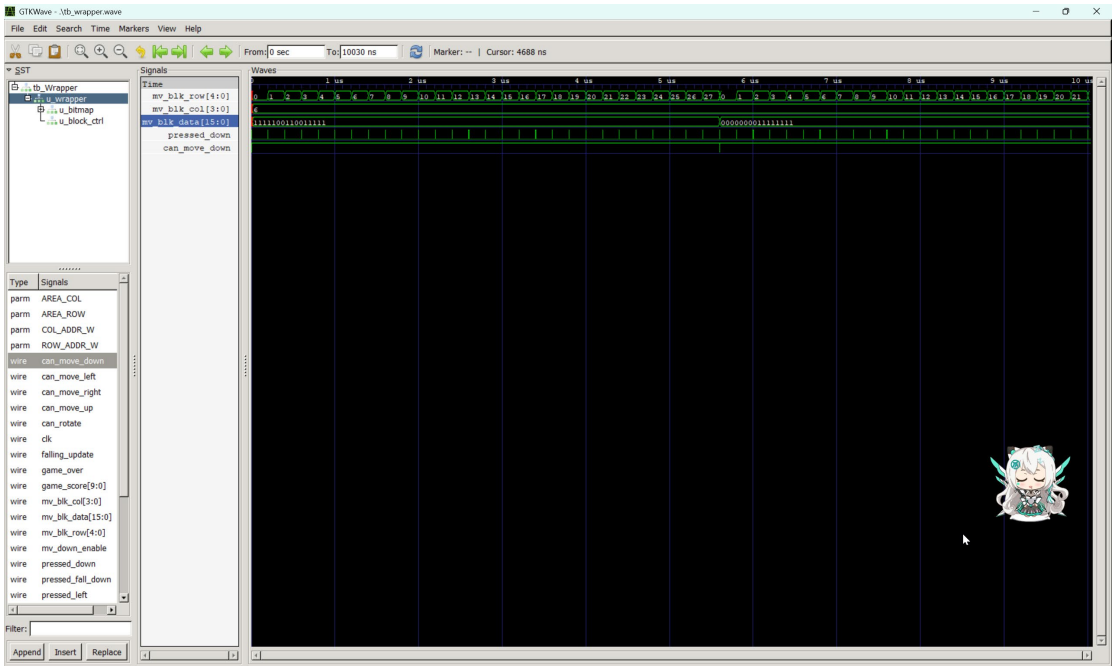
```
wire mv_down_enable;
reg [15:0] mv_blk_data = {
   4'b1_1_1_1,
   4'b1_1_1_1,
   4'b1_1_1_1,
   4'b1_1_1_1
};

Bitmap uut (
   .clk(clk),
   .rstn(rstn),
   .falling_update(falling_update),
   .game_over(game_over),
   .mv_blk_row(mv_blk_row),
   .mv_blk_col(mv_blk_col),
   .mv_blk_data(mv_blk_data),
   .mv_down_enable(mv_down_enable)
);

initial begin

   rstn = 1;
   #10;
   rstn = 0;
   #10;
   rstn = 1;
   #10;

   #10000;
   $finish;
end

always #1 clk = ~clk;

// new falling_update generation logic - use a counter to
implement periodic falling
reg [31:0] fall_counter = 0;
```

```
parameter FALL_RATE = 100; // adjust this value to control the
falling speed

always @(posedge clk or negedge rstn) begin
    if (~rstn) begin
        falling_update <= 1'b0;
        fall_counter <= 0;
        mv_blk_row <= 0;
    end
    else begin
        fall_counter <= fall_counter + 1;

        if (fall_counter >= FALL_RATE) begin
            falling_update <= 1'b1;  // periodically trigger
falling
            fall_counter <= 0;
        end
        else begin
            falling_update <= 1'b0;
        end

        // moving-block control logic
        if (falling_update) begin
            mv_blk_row <= mv_down_enable ? (mv_blk_row + 1) : 0;
        end

        // game-over handling
        if (game_over) begin
            falling_update <= 1'b0;
        end
    end
end

// output waveform file
initial begin
    $dumpfile("tb_bitmap.wave");
    $dumpvars;
    $display("save to tb_bitmap.wave");
```

```
end

endmodule
```

（2） Wrapper:

The result meets expectations.




```
`timescale 1ns / 1ps

module tb_Wrapper ();

//================================================================
==============
// Game wrapper, including core logic
//================================================================
==============
parameter AREA_ROW = 32;
parameter AREA_COL = 16;
parameter ROW_ADDR_W = 5;
parameter COL_ADDR_W = 4;

wire [ROW_ADDR_W-1:0] hdmi_bitmap_row;
wire [COL_ADDR_W-1:0] hdmi_bitmap_col;
wire [AREA_COL*2-1:0] hdmi_bitmap_data;
```

```
wire [ROW_ADDR_W-1:0] uart_bitmap_row;
wire [COL_ADDR_W-1:0] uart_bitmap_col;
wire [AREA_COL*2-1:0] uart_bitmap_data;

reg clk = 1'b0;
reg clk2 = 1'b0;
reg rstn = 1'b1;

reg pressed_left = 1'b1;
reg pressed_left_d1 = 1'b1;
wire pressed_left_trigger = (~pressed_left_d1) & pressed_left;

reg pressed_right = 1'b1;
reg pressed_right_d1 = 1'b1;
wire pressed_right_trigger = (~pressed_right_d1) &
pressed_right;

reg pressed_down = 1'b1;
reg pressed_down_d1 = 1'b1;
wire pressed_down_trigger = (~pressed_down_d1) & pressed_down;

reg pressed_up = 1'b1;
reg pressed_up_d1 = 1'b1;
wire pressed_up_trigger = (~pressed_up_d1) & pressed_up;

reg [ROW_ADDR_W+COL_ADDR_W-1:0] hdmi_bitmap_pos = 0;
reg [ROW_ADDR_W+COL_ADDR_W-1:0] uart_bitmap_pos = 0;

assign hdmi_bitmap_row = hdmi_bitmap_pos[8:4];
assign hdmi_bitmap_col = hdmi_bitmap_pos[3:0];

assign uart_bitmap_row = uart_bitmap_pos[8:4];
assign uart_bitmap_col = uart_bitmap_pos[3:0];

Wrapper #(
   .AREA_ROW(AREA_ROW),
   .AREA_COL(AREA_COL),
```

```
    .ROW_ADDR_W(ROW_ADDR_W),
    .COL_ADDR_W(COL_ADDR_W)
) u_wrapper (
    .clk(clk),
    .rstn(rstn),
    .pressed_left(pressed_left_trigger),     // key event
    .pressed_right(pressed_right_trigger),   //
    .pressed_down(pressed_down_trigger),     //
    .pressed_up(pressed_up_trigger),         //
    .r1_row(hdmi_bitmap_row),  // output channel #1
    .r1_data(hdmi_bitmap_data),//
    .r2_row(uart_bitmap_row),  // output channel #2
    .r2_data(uart_bitmap_data) //
);

initial begin

    rstn = 1;
    #10;
    rstn = 0;
    #10;
    rstn = 1;
    #10;

    #10000;
    $finish;
end

always @(posedge clk2 or negedge rstn) begin
    if (~rstn) begin
        hdmi_bitmap_pos <= 0;
        uart_bitmap_pos <= 0;
    end
    else begin
        hdmi_bitmap_pos <= hdmi_bitmap_pos + AREA_COL;
        uart_bitmap_pos <= uart_bitmap_pos + AREA_COL;
    end
end
```

```
always #1 clk = ~clk;

always #8 clk2 = ~clk2;

// always #100 pressed_left = ~pressed_left;
// always #100 pressed_right = ~pressed_right;
always #100 pressed_down = ~pressed_down;
// always #100 pressed_up = ~pressed_up;

always @(posedge clk) begin
   pressed_left_d1 <= pressed_left;
   pressed_right_d1 <= pressed_right;
   pressed_down_d1 <= pressed_down;
   pressed_up_d1 <= pressed_up;
end

// output waveform file
initial begin
    $dumpfile("tb_wrapper.wave");
    $dumpvars;
    $display("save to tb_wrapper.wave");
end

endmodule
```

4. Hardware synthesis reports, clock reports, reflection, and conclusions

（1） Hardware Resource Utilization:

```
Total LUTs: 4598 of 17536 (26.22%)
       LUTs as dram: 0 of 4440 (0.00%)
       LUTs as logic: 4598
Total Registers: 1048 of 26304 (3.98%)
Total Latches: 3

DRM18K:
Total DRM18K = 1.0 of 48 (2.08%)

APMs:
Total APMs = 0.00 of 30 (0.00%)

Total I/O ports = 42 of 240 (17.50%)
```

Module Inst Name	LUT	FF	Distributed RAM	APM	DRM	IO	USCM
Top	4601	1048	0	0	1	42	0
u_hdmi	439	359	0	0	1	8	0
ms72xx_ctl	182	208	0	0	1	0	0
pattern_vg	2	6	0	0	0	0	0
rgb2tmds	136	83	0	0	0	8	0
sync_vg	75	48	0	0	0	0	0
u_key	32	27	0	0	0	0	0
u_pll_0	0	0	0	0	0	0	0
u_pll_1	0	0	0	0	0	0	0
u_seg	263	25	0	0	0	0	0
u_uart	86	58	0	0	0	0	0
u_uart_tx	29	20	0	0	0	0	0
u_wrapper	3729	538	0	0	0	0	0
u_bitmap	3585	509	0	0	0	0	0
u_block_ctrl	144	29	0	0	0	0	0

（2） Clock Report:

	Check	Number of Issues
1	no_clock	4
2	virtual_clock	1
3	unexpandable_clocks	0
4	no_input_delay	9
5	no_output_delay	32
6	partial_input_delay	0
7	partial_output_delay	0
8	io_min_max_delay_consistency	0
9	input_delay_assigned_to_clock	0
10	loops	0
11	latch_loops	0
12	latchs	3

Clock Name	Type	Period	Frequency	Rise	Fall	Clock Load	Non-clock Load	Source	Targets
clk	Declared	20.000	50.000MHz	0.000	10.000	689	1		{ clk }
clk|u_pll_0/u_pll_e1/CLKOUT0_Inferred	Generated	100.000	10.000MHz	0.000	50.000	133	0	clk	{ u_pll_0/u_pll_e1/CLKOUT0 }
Top|clk_27M	Declared	1000.000	1.000MHz	0.000	500.000	0	0		{ clk_27M }
pll_1|u_pll_1/u_pll_e1/CLKOUT1	Declared	1000.000	1.000MHz	0.000	500.000	60	0		{ u_pll_1/u_pll_e1/CLKOUT1 }
pll_1|u_pll_1/u_pll_e1/CLKOUT0	Declared	1000.000	1.000MHz	0.000	500.000	93	0		{ u_pll_1/u_pll_e1/CLKOUT0 }

（3） Summary and Reflection:

This project was relatively difficult. Unlike previous programming-related courses, the introduction of clock signals made many situations more complex, requiring strong logical reasoning and frequent inference. The seemingly simple clock signal is actually very tricky. The continuous switching of the clock caused the difference between blocking and non-blocking assignments; everything changed.

Some of the difficult parts were unforgettable, such as the handling of line clearing, the handling of leftward and upward movement at boundaries, and the writing of the logic for refreshing a new moving block when bottom collision occurs. Every moment of painstaking thought and fluent typing suddenly feels stirring at this moment.

After two all-nighters, I still successfully completed the demo in the first batch.

On the day I submitted the report, I was running a fever due to a viral infection. But after writing this far, I feel as if I have completed a great achievement, and everything becomes less terrible.