1. Introduction and Background discussion on displays/VGA

A VGA monitor can be thought of as a grid of pixels where each pixel is visited and assigned a color independently. The VGA connector has three pins for RGB color signals, and two pins for vertical and horizontal synchronization signals. To be able to draw on the monitor, we need to control the signals to those pins with respect to different regions of the screen depending on what we want to draw. The vga_out.v module will drive those signals to the VGA connector appropriately, and provide the developer with the coordinates of the pixel being drawn at the current clock cycle. Using those coordinates, we are able to implement a drawing logic that controls how to color a given pixel to eventually create drawn objects.

1.1 VGA Signaling Protocol: vga_out.v

The VGA signaling protocol depends on two counters: hcount, and vcount. The horizontal counter counts up to 1679 and wraps around to zero. Every time it wraps around, the vertical counter is incremented, up until it counts to 827 which then also wraps around to zero, as described in the following code:

```
always @ (posedge clk) begin
    if (hcount == 11'd1679)   begin
        hcount <= 11'd0;
        if (vcount == 10'd827)
            vcount <= 10'd0;
        else
            vcount <= vcount + 10'd1;
    end
    else
        hcount <= hcount +1;
end
```

Those two counters will control the behaviors of the vga_out.v module. The counters set the outputs hsync and vsync which are used to connect to the corresponding hsync and vsync pins of the VGA connector that provide the timing information necessary. Hsync is set to zero when hcount is between 0 and 135 inclusive (and 1 otherwise), and vsync is set to 1 when vcount is between 0 and 2 inclusive (and 0 otherwise). We can see the following behavior described in the following code:

```
assign h_sync = (hcount <= 11'd135) ? 1'b0 : 1'b1;
assign v_sync = (vcount <= 10'd2)   ? 1'b1 : 1'b0;
```

The counters also control the RGB output of vga_out.v which connect to the RGB pins of the VGA controller. The RGB outputs accept color inputs only when the pixel in question is inside the visible region of the display. We know that the pixel is inside the visible region when hcount is between 336 and 1615 (inclusive), and vcount is between 27 and 826 (inclusive). If the pixel is not inside the visible region, then the RGB outputs are assigned to 0. The following code achieves the described behavior:

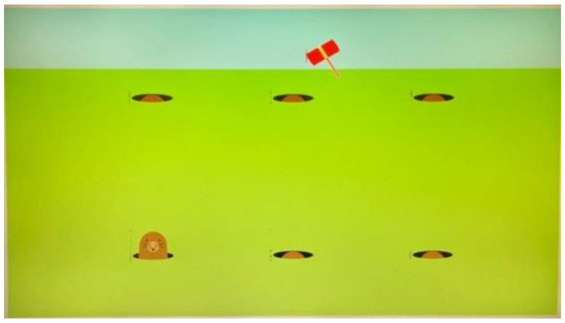
```
//if it is in the valid region, use the inputted color. Otherwise, black.
    assign pix_r = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >=
10'd27 && vcount <= 10'd826) ? r_in : 4'd0;
    assign pix_b = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >=
10'd27 && vcount <= 10'd826) ? b_in : 4'd0;
    assign pix_g = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >=
10'd27 && vcount <= 10'd826) ? g_in : 4'd0;
```

where r_in, b_in, and g_in are module wire inputs in vga_out.v that are assigned colors based on the drawing logic implemented in drawcon_v.

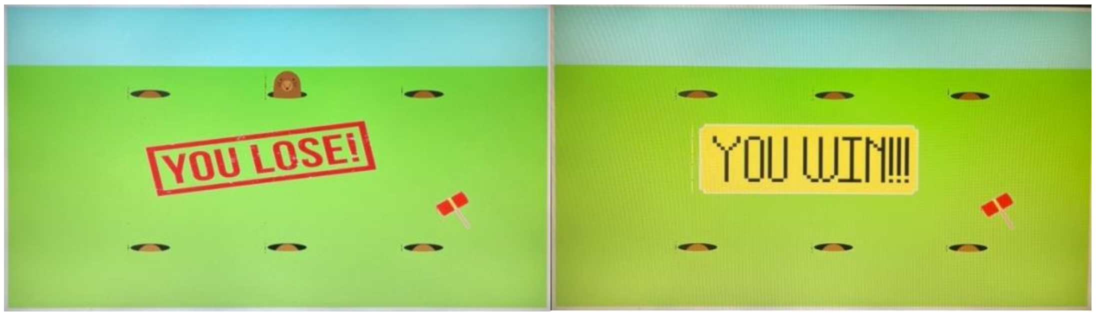
Lastly, the vga_out.v module informs the developer which visible pixel (curr_x, curr_y) is currently being drawn in that clock cycle. Since traditional displays start from the top left corner and goes down the row, then left from the second row and down the second row and so on, we want the coordinates of the top left corner to be (0,0) and the last pixel to be (1279, 799) which is at the bottom right corner. We can use an offset counter to assign the coordinates of the current pixel being drawn with respect to the horizontal and vertical counters. By using the same ranges used above, if hcount and vcount are inside those ranges, then we assigne curr_x to hcount – 336 and curr_y to vcount – 27.

```
assign curr_x = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >= 10'd27
&& vcount <= 10'd826) ? hcount - 11'd336 : 11'd0;

assign curr_y = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >= 10'd27
&& vcount <= 10'd826) ? vcount - 10'd27 : 10'd0;
```

2. **<u>Game design:</u> game_top.v**



The game design follows that of a real Whack-a-mole game with few modifications. In my design, I made six positions where the moles can pop out, and I wanted them to pop out in an unpredictable manner. Although players of a real whack-a-mole game you find in arcades play to achieve a high score, in my design it's a computer vs. player implementation. To win the game, you must hit 16 moles once they pop up from their holes, and you lose the game if you miss-hit the moles (hitting anywhere a mole isn't outside its hole) 16 times. The score keeping of the game is maintained over the seven segment displays on the Nexys A7 board on the first and fifth display from the right, for the player and computer respectively. When the game ends, it should display the win or lose message and the game freezes.



2.1 1 second clock and LFSR

I made a total of six 1-bit registers in game_top.v for each mole location named top-left, top-center, top-right, bottom-left, bottom-center, and bottom-right. We will call those registers *positional registers* for the rest of the report. We will use those positional registers in a way such that when one of them is set high, it signals for the mole in that corresponding location to pop up. The drawing implementation for that part is discussed in the next section. To be able to select “randomly” among the positional registers which one to set high we use a linear-feedback shift register. The algorithm implemented starts with initializing a 3-bit register named temp to 101. It XOR’s the first and second LSB bits, right shifts the result of the XOR, and inserts the result to the MSB bit of temp. The algorithm then takes the value of temp, if it is between 0 and 5 it will drive one of the positional registers high. Otherwise, nothing happens. This algorithm will be running in an always block that runs on approximately a 1 second clock. One cycle it runs the linear-feedback shift register algorithm, the next it resets all the positional registers to zero, forcing all moles to go back in the hole. The following code creates the 1 second clock, and uses it for the algorithm just described.

```
/////////////////////////// Moles "random selection" mechanism ////////////////////////////

    reg [25:0] clk3_counter = 26'd0;
    reg clk_3; // one second clock
    reg rst_2 = 1'd0;

    // clk for generating when moles pop up.
    always @ (posedge clk_out1)
    begin
        if (clk3_counter == 26'd50_000_000) begin // almost a second.
            clk_3 <= ~ clk_3;
            clk3_counter <= 26'd0;
        end
        else
```

```
            clk3_counter <= clk3_counter + 26'd1;

    end

    reg [2:0] temp = 3'b101;
    reg t;

    // implementing a linear feedback shift register for randomness
    always @ (posedge clk_3)
    begin
        if (rst_2 == 1'd0) begin // resets eveyrything for a second every other cycle.
            top_left <= 1'b0;
            top_center <= 1'b0;
            top_right <= 1'b0;
            bottom_left <= 1'b0;
            bottom_center <= 1'b0;
            bottom_right <= 1'b0;

        end
        else begin
            t <= temp[0] ^ temp[1];
            temp <= temp >> 1;
            temp[2] <= t;

            case (temp)
                3'd0: top_left <= 1'b1;
                3'd1: top_center <= 1'b1;
                3'd2: top_right <= 1'b1;
                3'd3: bottom_left <= 1'b1;
                3'd4: bottom_center <= 1'b1;
                3'd5: bottom_right <= 1'b1;
            endcase

        end

        rst_2 <= ~rst_2;
    end
```

2.2 Controlling hammer & hitting:

The hammer is restricted to hover over the six positions of the moles, slightly to the right and top of them. The buttons on the board are used to control the hammer's movements. The declared registers blkpos_x and blkpos_y in game_top.v resemble the current x and y coordinates of the hammer. To know whether the player hit the mole or committed a miss-hit, in an always block we check: if the center button is clicked, and the hammer is hovering over the mole, and the positional register of that mole is set high, then we increment the player's score (dig0) on the seven-segment display. Otherwise, increment the computer's score (dig4). The following code displays the implementation for two positions out of the six. The rest follow the same pattern.

```
///////////// converting clk_out1 to 20 Hz --> clk_2. enough to capture button inputs /////////////

    always @ (posedge clk_out1)
    begin
        if (clk2_counter == 22'd4173000) begin
            clk_2 <= ~ clk_2;
            clk2_counter <= 22'd0;
        end
        else
            clk2_counter <= clk2_counter + 22'd1;

    end


/////////////////////////// Clicking, Moving, & Score Keeping /////////////////////////////

    // six positions:
    // top right, top center, top left
    // down right, down center, down left, passed to drawcon as input where it gets set.
    always @ (posedge clk_2)
    begin

    if (!win && !lose) begin   // doesn't go on after game ends.
        if (center) begin  // if center is clicked

            if (top_left) begin  // meaning, if top left mole is up
                if (blkpos_x == 11'd368 && blkpos_y == 11'd136) begin   // and
hammer is hovering on it
                    dig0 <= dig0 + 4'd1;  // increase score
                end
                else
                    dig4 <= dig4 + 4'd1;  // miss. increase computer score (opponent)
            end

            else if (top_center) begin
                if (blkpos_x == 11'd689 && blkpos_y == 11'd136) begin
                    dig0 <= dig0 + 4'd1;
                end
                else
                    dig4 <= dig4 + 4'd1;
            end
```

We can observe from the code that we check for the clicking only when neither registers win and lose are driven high, meaning the game is still not finished. When either computer or the player get a score of 16, the lose or win registers get set high and the player has no further control over the game.

2.3 Moving the hammer:

The hammer position is initialized to start over the bottom right mole. To move the hammer, we check if either of the direction buttons are clicked and whether it is allowed to move in that direction. Existing in the same always block as the code above, this following code is used to move the hammer:

```
        else  begin  // if center is not clicked...

            // screen width = 1279. height = 799
            // up = 536, down = 136, right = 1010, left = 368,

            if (up && blkpos_y >= 11'd400)  // if up is clicked, and hammer is in lower half
of screen
                blkpos_y <= blkpos_y - 11'd400;

            if (down && blkpos_y <= 11'd400)
                blkpos_y <= blkpos_y + 11'd400;

            if (right && blkpos_x < 11'd1010)   // if right is clicked and hammer is not
on the right side, can go right
                blkpos_x <= blkpos_x + 11'd321;


            if (left && blkpos_x > 11'd415)
                blkpos_x <= blkpos_x - 11'd321;

        end
```

Now that the logic of the game is implemented, we pass the signals we controlled in game_top.v to drawcon.c to represent them in drawn objects on the screen.

3. Drawing logic: drawcon.v

drawcon.v is the module responsible for all the drawing logic and implementation. Each clock cycle, this module outputs the RGB colors for a pixel based on the drawing logic implemented. The core design of this module uses two registers for each RGB color, one for the background color and one for the color used to draw an object. For each RGB output r, g, or b, if the object color register is set to any value, use that value for the output. Otherwise, use the background color. This is the backbone of this module and it is described in the following code.

```
    assign r = (blk_r != 4'd0) ? blk_r : bg_r;
    assign g = (blk_g != 4'd0) ? blk_g : bg_g;
    assign b = (blk_b != 4'd0) ? blk_b : bg_b;
```

3.1 Drawing the background:

Drawing the background is fairly simple. I just separated the background into two regions: sky and a grass field. I chose the sky and grass boundary to start at row 160 of the visible region of the screen. The code below achieves this outcome.

```
///////////////////////////// Drawing the Background /////////////////////////////////////

        else begin  // if draw_xy is not on a mole's location then it's the background.
            if (draw_y > 160) begin  // boundry that seperates sky and land
                bg_r = 4'h9;  // light green grass
                bg_g = 4'hE;
                bg_b = 4'h5;
            end
            else begin
                bg_r = 4'hB;  // light blue skies.
                bg_g = 4'hF;
                bg_b = 4'hF;
            end
        end
```

3.2 Drawing the mole sprites:

In place of the six mole positions, we have two sprites in use. One for when the mole is still under the hole, and one where it pops out.

Both sprites are of dimensions w: 100 and h: 80 in pixels. In an always block, for each of the six positions, we want to check when draw_x and draw_y are inside those regions of those dimensions. If they are, then we check if the positional register of that position is set high. If it is, then we use the sprite of the mole outside its hole. Otherwise, we use the sprite of the mole inside its hole. The following code shows an example for the top left position:

Memory block instantiations:

```
mole_under mole_under_0 (.clka(clk), .addra(address_mole_under),
.douta(mem_mole_under));
    mole_up mole_up_0 (.clka(clk), .addra(address_mole_up),
.douta(mem_mole_up));
```

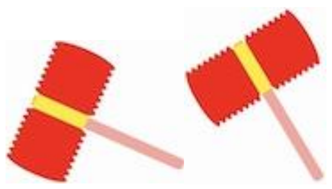
Drawing logic:

```
////////////////////////////// Drawing the Moles ///////////////////////////////////////////

            // Either inside the hole, or out. Depending on whether the signals top_left, top_right, ...., are
set.
            else if(draw_x > 11'd287 && draw_x <= 11'd387 && draw_y > 11'd168 &&
draw_y <= 11'd248) begin
                if (!top_left) begin
                    address_mole_under = (draw_y - 11'd169)*11'd100 + (draw_x -
11'd288);
                    bg_r = mem_mole_under[11:8];
                    bg_g = mem_mole_under[7:4];
                    bg_b = mem_mole_under[3:0];
                end
                else begin
                    address_mole_up = (draw_y - 11'd169)*11'd100 + (draw_x -
11'd288);
                    bg_r = mem_mole_up[11:8];
                    bg_g = mem_mole_up[7:4];
                    bg_b = mem_mole_up[3:0];
                end

            end
```

Those steps are repeated six times for each position where the moles exist.

<u>3.3 Drawing the hammers:</u>

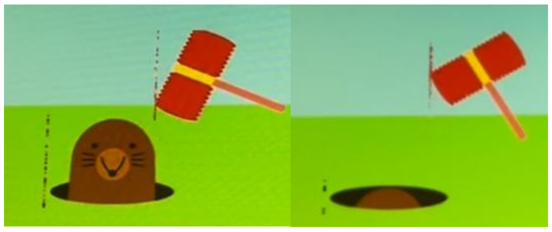
The hammer has two states, which are standby and hitting.



The background of those images is white, so in the python script that generates the coe file, I included code that replaces all white color in an image with the same shade of green used for the grass. However, when the hammer is hovering over the top row of moles, the hammer sprite needs to reflect both background colors of the grass, and the sky. We need to figure out which rows of the two images separates the two background colors. To do that, I used the following formula:

$$row = (draw_y - \left(blkpos_y - \frac{sprite\ width}{2} + 1\right))$$

We know from drawing the game's background that draw_y in this case is 160. Then, in the script we include code that paints the background blue when iterating above that row and green otherwise. Now, we have two extra sprites for the hammer that are only used when the hammer hovers over the top row of moles.



We use the same logic of detecting whether draw_x and draw_y are on the hammer's location as the code block above, this time however using the given blkpos_x and blkpos_y inputted from game_top.v. The code responsible for selecting between the four sprites is shown below:

Memory block instantiation:

```
    hammer_up hammer_up_0 (.clka(clk), .addra(address_hammer_up),
.douta(mem_hammer_up));
    hammer_up1 hammer_up_1 (.clka(clk), .addra(address_hammer_up1),
.douta(mem_hammer_up1)); // blue background included

    hammer_down hammer_down_0 (.clka(clk), .addra(address_hammer_down),
.douta(mem_hammer_down));
    hammer_down1 hammer_down_1 (.clka(clk), .addra(address_hammer_down1),
.douta(mem_hammer_down1)); // blue background included
```

Drawing logic:

```
/////////////////////////////// Drawing the Hammer ////////////////////////////////////////

    always @ *
    begin

        // draw hammer in neutral state
        // if center is not clicked, and draw_xy are on hammer's location
        if (!center && draw_x > blkpos_x && draw_x <= blkpos_x + 11'd80 &&
draw_y > blkpos_y - 11'd50 && draw_y <= blkpos_y + 11'd50) begin
            if (blkpos_y > 11'd400) begin
                address_hammer_up = (draw_y - (blkpos_y - 11'd51))*11'd80 +
(draw_x - blkpos_x +11'd1);
                blk_r = mem_hammer_up[11:8];
                blk_g = mem_hammer_up[7:4];
                blk_b = mem_hammer_up[3:0];
            end
            else  begin
                address_hammer_up1 = (draw_y - (blkpos_y - 11'd51))*11'd80 +
(draw_x - blkpos_x +11'd1);
                blk_r = mem_hammer_up1[11:8];
                blk_g = mem_hammer_up1[7:4];
                blk_b = mem_hammer_up1[3:0];
            end
        end

        // draw hammer on whacking state
        else if (center && draw_x > blkpos_x && draw_x <= blkpos_x + 11'd100
&& draw_y > blkpos_y - 11'd40 && draw_y <= blkpos_y + 11'd40) begin
            if (blkpos_y > 11'd400) begin
                address_hammer_down = (draw_y - (blkpos_y - 11'd41))*11'd100
+ (draw_x - blkpos_x +11'd1);
                blk_r = mem_hammer_down[11:8];
                blk_g = mem_hammer_down[7:4];
                blk_b = mem_hammer_down[3:0];
            end
            else  begin
                address_hammer_down1 = (draw_y - (blkpos_y -
11'd41))*11'd100 + (draw_x - blkpos_x +11'd1);
                blk_r = mem_hammer_down1[11:8];
                blk_g = mem_hammer_down1[7:4];
                blk_b = mem_hammer_down1[3:0];
            end
        end
        else  begin
            blk_r = 4'b0000;    // when zero, then background color is used.
            blk_g = 4'b0000;
            blk_b = 4'b0000;
        end

    end

endmodule
```

4. Signals workflow

Game_top.v is the module that instantiates all other modules.

```
    reg top_left, top_center, top_right, bottom_left, bottom_center,
bottom_right;  // mole locations. When set high, mole is popped up from respective hole.

    reg win = 1'b0;                          // if set high, game stops, user won
    reg lose = 1'b0;                         // if set high, game stops, Computer wins

    reg [21:0] clk2_counter = 22'd0;
    reg clk_2 = 1'b0;
    reg [10:0] blkpos_x = 11'd1010;       // starting hammer positions
    reg [10:0] blkpos_y = 11'd536;

    wire [10:0] draw_x;
    wire [10:0] draw_y;

    wire [3:0] r, g, b;

    reg [3:0] dig0 = 4'd0;                // has user score
    reg [3:0] dig4 = 4'd0;                // has computer score


    // module responsible for the seven segments on the board
    multidigit m3 (.dig7(4'd0), .dig6(4'd0), .dig5(4'd0), .dig4(dig4), .dig3(4'd0),
.dig2(4'd0), .dig1(4'd0), .dig0(dig0),
    .clk(clk), .rst(rst),
    .a(a1), .b(b1), .c(c1), .d(d1), .e(e1), .f(f1), .g(g1),
    .an(an));

    // drawcon does the drawing
    drawcon m2 (.clk(clk_out1), .win(win),
.lose(lose),.center(center),.blkpos_x(blkpos_x), .blkpos_y(blkpos_y)
,.draw_x(draw_x),
    .draw_y(draw_y), .r(r), .g(g), .b(b), .top_left(top_left),
.top_center(top_center), .top_right(top_right), .bottom_left(bottom_left),
    .bottom_center(bottom_center), .bottom_right(bottom_right));

    //vga_out controls the signals going into the vga screen. Also tells us what is currently being
drawn.
    vga_out m1 (.clk(clk_out1), .r_in(r), .b_in(b), .g_in(g), .pix_r(pix_r),
.pix_b(pix_b), .pix_g(pix_g), .h_sync(h_sync), .v_sync(v_sync),
    .curr_x(draw_x), .curr_y(draw_y));
```

From a workflow perspective, it is also good to mention that in addition to controlling all the game logic signals, game_top.v also declares all the color output wires, passes them to drawcon.v where they get set according to the drawing logic, then passes those values to vga_out.v where displaying them on the monitor follows the discussed VGA protocol.

5. Testing

Before testing anything on the screen, creating a testbench for vga_out.v was useful. It confirmed that things are working as the specification detailed. However, after starting to experiment with drawing objects on the monitor, the monitor was sufficient enough on detailing which parts of the implementation were faulty and which parts were working as designed. Since we are implementing a visual game, what shows on the screen eventually is more important than monitoring how individual signals are changing through a testbench. Using testbenches could come in handy if a debugging process required a closer look at the values of the signals being used. I personally didn't need to use a testbench as I didn't run through complex bugs.

6. Reflection

I really appreciate how detailed and sufficient the steps are in the gitlab project specification. They were clear and made it easy for me to later understand how I will implement my project and where different designs logic should be located. They also explained how VGA's work pretty well. The TAs were also really helpful (in the labs and also via email or physical meetups), and I feel there are plenty resources given to help with progress.

It would've helped to include small example code on how sprites are implemented beyond just the instantiation of the memory block. It would also be worth to mention that address zero corresponds to the first pixel of the sprite. There aren't many examples online on the particular way we initiated memory blocks, and some extra instructions could spare the students some time resulting in the same learning outcome. Other than that, the instructions on sprites were also really good.

Regarding the problem I have with the missing pixels when drawing the sprites, I have been told that they are related the clock's speed, but since the VGA requires a certain clock speed then there isn't much that I can do about it. What I couldn't understand is why other student didn't face this problem. Is it because of the particular sprite sizes I was using? I am not sure.

7. Appendix

Vga_out.v:

```
module vga_out(
    input clk,
    input [3:0] r_in, b_in, g_in,
    output [3:0] pix_r, pix_b, pix_g,
    output h_sync, v_sync,
    output [10:0] curr_x,
    output [9:0] curr_y
    );

    reg [10:0] hcount = 11'd0;
    reg [9:0] vcount = 10'd0;

    assign h_sync = (hcount <= 11'd135) ? 1'b0 : 1'b1;
    assign v_sync = (vcount <= 10'd2)   ? 1'b1 : 1'b0;

    //if it is in the valid region, use the inputted color. Otherwise, black.
    assign pix_r = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >=
10'd27 && vcount <= 10'd826) ? r_in : 4'd0;
    assign pix_b = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >=
10'd27 && vcount <= 10'd826) ? b_in : 4'd0;
```

```
    assign pix_g = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >=
10'd27 && vcount <= 10'd826) ? g_in : 4'd0;

    // assign the current position with respect to the visibile region
    assign curr_x = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >=
10'd27 && vcount <= 10'd826) ? hcount - 11'd336 : 11'd0;
    assign curr_y = (hcount >= 11'd336 && hcount <= 11'd1615 && vcount >=
10'd27 && vcount <= 10'd826) ? vcount - 10'd27 : 10'd0;



    always @ (posedge clk) begin

        if (hcount == 11'd1679)  begin
            hcount <= 11'd0;
            if (vcount == 10'd827)
                vcount <= 10'd0;
            else
                vcount <= vcount + 10'd1;
        end
        else
            hcount <= hcount +1;

    end
endmodule
```

drawcon.v:

```
module drawcon(
    input clk, center, win, lose,
    input [10:0] blkpos_x, draw_x,
    input [10:0] blkpos_y, draw_y,
    output [3:0] r, g, b,
    input top_left, top_center, top_right, bottom_left, bottom_center,
bottom_right
    );


    reg [3:0] bg_r, bg_g, bg_b, blk_r, blk_g, blk_b;

    assign r = (blk_r != 4'd0) ? blk_r : bg_r;
    assign g = (blk_g != 4'd0) ? blk_g : bg_g;
    assign b = (blk_b != 4'd0) ? blk_b : bg_b;

    // declerations for wires/regs used for memory blocks.
    reg [13:0] address_mole_under;
    reg [13:0] address_mole_up;
    reg [13:0] address_hammer_up;
    reg [13:0] address_hammer_up1;
    reg [13:0] address_hammer_down;
    reg [13:0] address_hammer_down1;
    reg [16:0] address_win;
    reg [16:0] address_lose;
    wire [11:0] mem_mole_under;
    wire [11:0] mem_mole_up;
    wire [11:0] mem_hammer_up;
    wire [11:0] mem_hammer_up1;
    wire [11:0] mem_hammer_down;
    wire [11:0] mem_hammer_down1;
    wire [11:0] mem_win;
    wire [11:0] mem_lose;

    // memory blocks
    mole_under mole_under_0 (.clka(clk), .addra(address_mole_under),
.douta(mem_mole_under));
    mole_up mole_up_0 (.clka(clk), .addra(address_mole_up),
.douta(mem_mole_up));
    hammer_up hammer_up_0 (.clka(clk), .addra(address_hammer_up),
.douta(mem_hammer_up));
    hammer_up1 hammer_up_1 (.clka(clk), .addra(address_hammer_up1),
.douta(mem_hammer_up1)); // blue background included
    hammer_down hammer_down_0 (.clka(clk), .addra(address_hammer_down),
.douta(mem_hammer_down));
    hammer_down1 hammer_down_1 (.clka(clk), .addra(address_hammer_down1),
.douta(mem_hammer_down1)); // blue background included
    win win_0 (.clka(clk), .addra(address_win), .douta(mem_win));
    lose lose_0 (.clka(clk), .addra(address_lose), .douta(mem_lose));

    // this block handles the moles and background drawings, and the "you win" and "you lose"
drawings
    always @ *
    begin

        // draw the boarders
        if ((draw_x > 11'd0 && draw_x < 11'd10) || (draw_x < 11'd1279 &&
draw_x > 11'd1269) || (draw_y > 11'd0 && draw_y < 11'd10) || (draw_y < 11'd799
&& draw_y > 11'd789)) begin
            bg_r = 4'b1111; // white boardess
            bg_g = 4'b1111;
            bg_b = 4'b1111;
        end

//////////////////////////// Drawing the Moles ////////////////////////////////////////

        // Either inside the hole, or out. Depending on whether the signals top_left, top_right, ...., are
set.
        else if(draw_x > 11'd287 && draw_x <= 11'd387 && draw_y > 11'd168 &&
draw_y <= 11'd248) begin
            if (!top_left) begin
```

```
                address_mole_under = (draw_y - 11'd169)*11'd100 + (draw_x -
11'd288);
                bg_r = mem_mole_under[11:8];
                bg_g = mem_mole_under[7:4];
                bg_b = mem_mole_under[3:0];
            end
            else begin
                address_mole_up = (draw_y - 11'd169)*11'd100 + (draw_x -
11'd288);
                bg_r = mem_mole_up[11:8];
                bg_g = mem_mole_up[7:4];
                bg_b = mem_mole_up[3:0];
            end

        end

        else if(draw_x > 11'd608 && draw_x <= 11'd708 && draw_y > 11'd168 &&
draw_y <= 11'd248) begin
            if (!top_center) begin
                address_mole_under = (draw_y- 11'd169)*11'd100 + (draw_x -
11'd609);
                bg_r = mem_mole_under[11:8];
                bg_g = mem_mole_under[7:4];
                bg_b = mem_mole_under[3:0];
            end
            else begin
                address_mole_up = (draw_y- 11'd169)*11'd100 + (draw_x -
11'd609);
                bg_r = mem_mole_up[11:8];
                bg_g = mem_mole_up[7:4];
                bg_b = mem_mole_up[3:0];
            end
        end
        else if(draw_x > 11'd928 && draw_x <= 11'd1028 && draw_y > 11'd168 &&
draw_y <= 11'd248) begin // upper right box
            if (!top_right) begin
                address_mole_under = (draw_y - 11'd169)*11'd100 + (draw_x -
11'd929);
                bg_r = mem_mole_under[11:8];
                bg_g = mem_mole_under[7:4];
                bg_b = mem_mole_under[3:0];
            end
            else begin
                address_mole_up = (draw_y - 11'd169)*11'd100 + (draw_x -
11'd929);
                bg_r = mem_mole_up[11:8];
                bg_g = mem_mole_up[7:4];
                bg_b = mem_mole_up[3:0];
            end
        end
        else if(draw_x > 11'd287 && draw_x <= 11'd387 && draw_y > 11'd568 &&
draw_y <= 11'd648) begin // lower left box
            if (!bottom_left) begin
                address_mole_under = (draw_y - 11'd569)*11'd100 + (draw_x -
11'd288);
                bg_r = mem_mole_under[11:8];
                bg_g = mem_mole_under[7:4];
                bg_b = mem_mole_under[3:0];
            end
            else begin
                address_mole_up = (draw_y - 11'd569)*11'd100 + (draw_x -
11'd288);
                bg_r = mem_mole_up[11:8];
                bg_g = mem_mole_up[7:4];
                bg_b = mem_mole_up[3:0];
            end
        end
        else if(draw_x > 11'd608 && draw_x <= 11'd708 && draw_y > 11'd568 &&
draw_y <= 11'd648) begin // lower center box
            if (!bottom_center) begin
                address_mole_under = (draw_y - 11'd569)*11'd100 + (draw_x -
11'd609);
                bg_r = mem_mole_under[11:8];
                bg_g = mem_mole_under[7:4];
                bg_b = mem_mole_under[3:0];
            end
            else begin
                address_mole_up = (draw_y - 11'd569)*11'd100 + (draw_x -
11'd609);
                bg_r = mem_mole_up[11:8];
                bg_g = mem_mole_up[7:4];
                bg_b = mem_mole_up[3:0];
            end
        end
        else if(draw_x > 11'd928 && draw_x <= 11'd1028 && draw_y > 11'd568 &&
draw_y <= 11'd648) begin // lower right box
            if (!bottom_right) begin
                address_mole_under = (draw_y - 11'd569)*11'd100 + (draw_x -
11'd929);
                bg_r = mem_mole_under[11:8];
                bg_g = mem_mole_under[7:4];
                bg_b = mem_mole_under[3:0];
            end
            else begin
                address_mole_up = (draw_y - 11'd569)*11'd100 + (draw_x -
11'd929);
                bg_r = mem_mole_up[11:8];
                bg_g = mem_mole_up[7:4];
                bg_b = mem_mole_up[3:0];
            end
        end


////////////////////////// Drawing the Win/Lose Messages /////////////////////////////////

        else if(draw_x > 11'd325 && draw_x <= 11'd875 && draw_y > 11'd295 &&
draw_y <= 11'd505) begin
            if (win) begin
                address_win = (draw_y - 11'd296)*11'd550 + (draw_x -
11'd326);
                bg_r = mem_win[11:8];
                bg_g = mem_win[7:4];
                bg_b = mem_win[3:0];
            end
            else if (lose) begin
                address_lose = (draw_y - 11'd296)*11'd550 + (draw_x -
11'd326);
                bg_r = mem_lose[11:8];
                bg_g = mem_lose[7:4];
                bg_b = mem_lose[3:0];
            end
        end

////////////////////////////// Drawing the Background ////////////////////////////////////

        else begin // if draw_xy is not on a mole's location then it's the background.
            if (draw_y > 160) begin // boundry that seperates sky and land
                bg_r = 4'h9; // light green grass
                bg_g = 4'hE;
                bg_b = 4'h5;
            end
            else begin
                bg_r = 4'hB; // light blue skies.
                bg_g = 4'hF;
                bg_b = 4'hF;
            end
        end

    end

////////////////////////////// Drawing the Hammer //////////////////////////////////////

    always @ *
    begin

        // draw hammer in neutral state
        // if center is not clicked, and draw_xy are on hammer's location
        if (!center && draw_x > blkpos_x && draw_x <= blkpos_x + 11'd80 &&
draw_y > blkpos_y - 11'd50 && draw_y <= blkpos_y + 11'd50) begin
            if (blkpos_y > 11'd400) begin
```

```
                address_hammer_up = (draw_y - (blkpos_y - 11'd51))*11'd80 +
(draw_x - blkpos_x +11'd1);
                blk_r = mem_hammer_up[11:8];
                blk_g = mem_hammer_up[7:4];
                blk_b = mem_hammer_up[3:0];
            end
            else begin
                address_hammer_up1 = (draw_y - (blkpos_y - 11'd51))*11'd80 +
(draw_x - blkpos_x +11'd1);
                blk_r = mem_hammer_up1[11:8];
                blk_g = mem_hammer_up1[7:4];
                blk_b = mem_hammer_up1[3:0];
            end
        end

        // draw hammer on whacking state
        else if (center && draw_x > blkpos_x && draw_x <= blkpos_x + 11'd100
&& draw_y > blkpos_y - 11'd40 && draw_y <= blkpos_y + 11'd40) begin
            if (blkpos_y > 11'd400) begin
                address_hammer_down = (draw_y - (blkpos_y - 11'd41))*11'd100
+ (draw_x - blkpos_x +11'd1);
                blk_r = mem_hammer_down[11:8];
                blk_g = mem_hammer_down[7:4];
                blk_b = mem_hammer_down[3:0];
            end
            else begin
                address_hammer_down1 = (draw_y - (blkpos_y -
11'd41))*11'd100 + (draw_x - blkpos_x +11'd1);
                blk_r = mem_hammer_down1[11:8];
                blk_g = mem_hammer_down1[7:4];
                blk_b = mem_hammer_down1[3:0];
            end
        end
        else begin
            blk_r = 4'b0000;    // when zero, then background color is used.
            blk_g = 4'b0000;
            blk_b = 4'b0000;
        end

    end

endmodule
```

game_top.v:

```
module game_top(
    input clk, rst,
    input up, down, left, right, center,
    input [3:0] r_in, b_in, g_in,
    output [3:0] pix_r, pix_b, pix_g,
    output h_sync, v_sync, a1, b1, c1, d1, e1, f1, g1,
    output [7:0] an
    );

    wire clk_out1;

    reg top_left, top_center, top_right, bottom_left, bottom_center,
bottom_right;  // mole locations. When set high, mole is popped up from respective hole.

    reg win = 1'b0;                          // if set high, game stops, user won
    reg lose = 1'b0;                         // if set high, game stops, Computer wins

    reg [21:0] clk2_counter = 22'd0;
    reg clk_2 = 1'b0;

    reg [10:0] blkpos_x = 11'd1010;         // starting hammer positions
    reg [10:0] blkpos_y = 11'd536;

    wire [10:0] draw_x;
    wire [10:0] draw_y;

    wire [3:0] r, g, b;

    reg [3:0] dig0 = 4'd0;                   // has user score
    reg [3:0] dig4 = 4'd0;                   // has computer score


    // module responsible for the seven segments on the board
    multidigit m3 (.dig7(4'd0), .dig6(4'd0), .dig5(4'd0), .dig4(dig4), .dig3(4'd0),
.dig2(4'd0), .dig1(4'd0), .dig0(dig0),
    .clk(clk), .rst(rst),
    .a(a1), .b(b1), .c(c1), .d(d1), .e(e1), .f(f1), .g(g1),
    .an(an));

    // drawcon does the drawing
    drawcon m2 (.clk(clk_out1), .win(win),
.lose(lose),.center(center),.blkpos_x(blkpos_x), .blkpos_y(blkpos_y)
,.draw_x(draw_x),
    .draw_y(draw_y), .r(r), .g(g), .b(b), .top_left(top_left),
.top_center(top_center), .top_right(top_right), .bottom_left(bottom_left),
    .bottom_center(bottom_center), .bottom_right(bottom_right));

    //vga_out controls the signals going into the vga screen. Also tells us what is currently being
drawn.
    vga_out m1 (.clk(clk_out1), .r_in(r), .b_in(b), .g_in(g), .pix_r(pix_r),
.pix_b(pix_b), .pix_g(pix_g), .h_sync(h_sync), .v_sync(v_sync),
    .curr_x(draw_x), .curr_y(draw_y));
```

```
///////////// converting clk_out1 to 20 Hz --> clk_2. enough to capture button inputs /////////////

    always @ (posedge clk_out1)
    begin
        if (clk2_counter == 22'd4173000) begin
            clk_2 <= ~ clk_2;
            clk2_counter <= 22'd0;
        end
        else
            clk2_counter <= clk2_counter + 22'd1;

    end


/////////////////////////// Clicking, Moving, & Score Keeping //////////////////////////////

    // six positions:
    // top right, top center, top left
    // down right, down center, down left, passed to drawcon as input where it gets set.
    always @ (posedge clk_2)
    begin

    if (!win && !lose) begin    // doesn't go on after game ends.
        if (center) begin  // if center is clicked

            if (top_left) begin  // meaning, if top left mole is up
                if (blkpos_x == 11'd368 && blkpos_y == 11'd136) begin   // and
hammer is hovering on it
                    dig0 <= dig0 + 4'd1;  // increase score
                end
                else
                    dig4 <= dig4 + 4'd1;  // miss. increase computer score (opponent)
            end

            else if (top_center) begin
                if (blkpos_x == 11'd689 && blkpos_y == 11'd136) begin
                    dig0 <= dig0 + 4'd1;
                end
                else
                    dig4 <= dig4 + 4'd1;
            end

            else if (top_right) begin
                if (blkpos_x == 11'd1010 && blkpos_y == 11'd136) begin
                    dig0 <= dig0 + 4'd1;
                end
                else
                    dig4 <= dig4 + 4'd1;
            end

            else if (bottom_left) begin
```

```
                if (blkpos_x == 11'd368 && blkpos_y == 11'd536) begin
                    dig0 <= dig0 + 4'd1;
                end
                else
                    dig4 <= dig4 + 4'd1;
            end


            else if (bottom_center) begin
                if (blkpos_x == 11'd689 && blkpos_y == 11'd536) begin
                    dig0 <= dig0 + 4'd1;
                end
                else
                    dig4 <= dig4 + 4'd1;
            end


            else if (bottom_right) begin
                if (blkpos_x == 11'd1010 && blkpos_y == 11'd536) begin
                    dig0 <= dig0 + 4'd1;
                end
                else
                    dig4 <= dig4 + 4'd1;
            end

            else
                dig4 <= dig4 + 4'd1;


            if (dig0 == 4'b1111) begin // win
                win = 1'b1;
            end
            else if (dig4 == 4'b1111) begin // lose
                lose = 1'b1;
            end

        end
        else begin // if center is not clicked...

            // screen width = 1279. height = 799
            // up = 536, down = 136, right = 1010, left = 368,

            if (up && blkpos_y >= 11'd400) // if up is clicked, and hammer is in lower half
of screen
                blkpos_y <= blkpos_y - 11'd400;

            if (down && blkpos_y <= 11'd400)
                blkpos_y <= blkpos_y + 11'd400;

            if (right && blkpos_x < 11'd1010)  // if right is clicked and hammer is not
on the right side, can go right
```

```
                blkpos_x <= blkpos_x + 11'd321;


            if (left && blkpos_x > 11'd415)
                blkpos_x <= blkpos_x - 11'd321;

        end
    end
end


/////////////////////////// Moles "random selection" mechanism ////////////////////////////

    reg [25:0] clk3_counter = 26'd0;
    reg clk_3; // one second clock
    reg rst_2 = 1'd0;

    // clk for generating when moles pop up.
    always @ (posedge clk_out1)
    begin
        if (clk3_counter == 26'd50_000_000) begin // almost a second.
            clk_3 <= ~ clk_3;
            clk3_counter <= 26'd0;
        end
        else
            clk3_counter <= clk3_counter + 26'd1;

    end

    reg [2:0] temp = 3'b101;
    reg t;

    // implementing a linear feedback shift register for randomness
    always @ (posedge clk_3)
    begin
        if (rst_2 == 1'd0) begin // resets eveyrything for a second every other cycle.
            top_left <= 1'b0;
            top_center <= 1'b0;
            top_right <= 1'b0;
            bottom_left <= 1'b0;
            bottom_center <= 1'b0;
            bottom_right <= 1'b0;

        end
        else begin
            t <= temp[0] ^ temp[1];
            temp <= temp >> 1;
            temp[2] <= t;

            case (temp)
                3'd0: top_left <= 1'b1;
                3'd1: top_center <= 1'b1;
                3'd2: top_right <= 1'b1;
                3'd3: bottom_left <= 1'b1;
                3'd4: bottom_center <= 1'b1;
                3'd5: bottom_right <= 1'b1;
            endcase

        end

        rst_2 <= ~rst_2;
    end
```

```
// injury, or severe property or environmental damage
// (individually and collectively, "Critical
// Applications"). Customer assumes the sole risk and
// liability of any use of Xilinx products in Critical
// Applications, subject only to applicable laws and
// regulations governing limitations on product liability.
//
// THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
// PART OF THIS FILE AT ALL TIMES.
//
//----------------------------------------------------------------------------
// User entered comments
//----------------------------------------------------------------------------
// None
//
//----------------------------------------------------------------------------
//  Output     Output      Phase    Duty Cycle   Pk-to-Pk     Phase
//   Clock     Freq (MHz)  (degrees)    (%)     Jitter (ps)  Error (ps)
//----------------------------------------------------------------------------
// clk_out1__83.45588______0.000______50.0______299.155____297.220
//
//----------------------------------------------------------------------------
// Input Clock   Freq (MHz)    Input Jitter (UI)
//----------------------------------------------------------------------------
// __primary_________100.000____________0.010

// The following must be inserted into your Verilog file for this
// core to be instantiated. Change the instance name and port connections
// (in parentheses) to your own signal names.

//----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG

  clk_wiz_0 instance_1
   (
    // Clock out ports
    .clk_out1(clk_out1),          // output clk_out1
   // Clock in ports
    .clk_in1(clk));          // input clk_in1



// INST_TAG_END ------ End INSTANTIATION Template ---------
endmodule
```