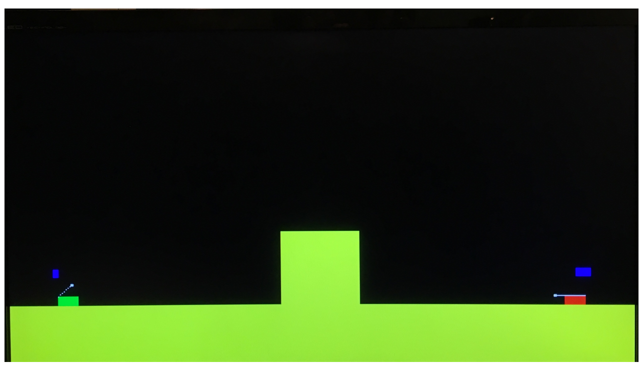
ELEX 7660: Digital System Design

Design Project

VGA Based Arcade Game (ZART)

Terry Calderbank
Kolyn Gray



12 April, 2018

Contents

1 Introduction

The goal of this project was to implement our new-found knowledge of finite state machine design in order to make a simple arcade game displayed over the VGA protocol. In the project we recreated an old and incredibly obscure arcade game called ZART (Zee-Artillery). As an MS-DOS based game for Windows and PalmOS, ZART was considered retro when it was released by a small shareware company called PLBM games in 1999[1]. While extremely dated looking and soon to be faced with all kinds of compatibility issues with the new versions of windows, ZART was unique in that it only required each player to use a single keyboard key to play. The addictive game play, based around iteratively honing in an artillery round on an opponent's tank, also featured collapsible sand terrain as well as randomly generated wind effects.

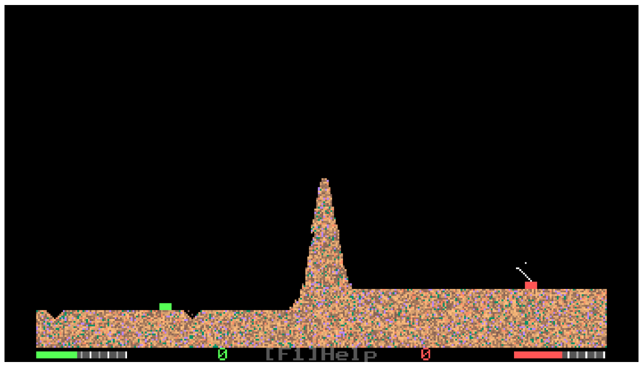


Figure 1: Gameplay of Zart by PLBM Games (1999)

1.1 Motivation

We chose this project because it allowed us to apply everything we have learned about state machines in a complex, but not overwhelming manner. We also feel that implementing somewhat realistic two-dimensional kinematic projectile motion in System Verilog was an interesting challenge. This project required us to learn a lot about VGA display signals as well as the additional hardware required to display additional colour ranges. While VGA is becoming obsolete, we felt that it was a good stepping stone towards understanding more advanced protocols like HDMI and DVI.

2 Project Overview

As seen in Figure 2, the arcade game system is made up of various blocks that all come together to form a system. The system inputs are the push buttons used by the player, and the output is 5 signals going out to the VGA monitor.

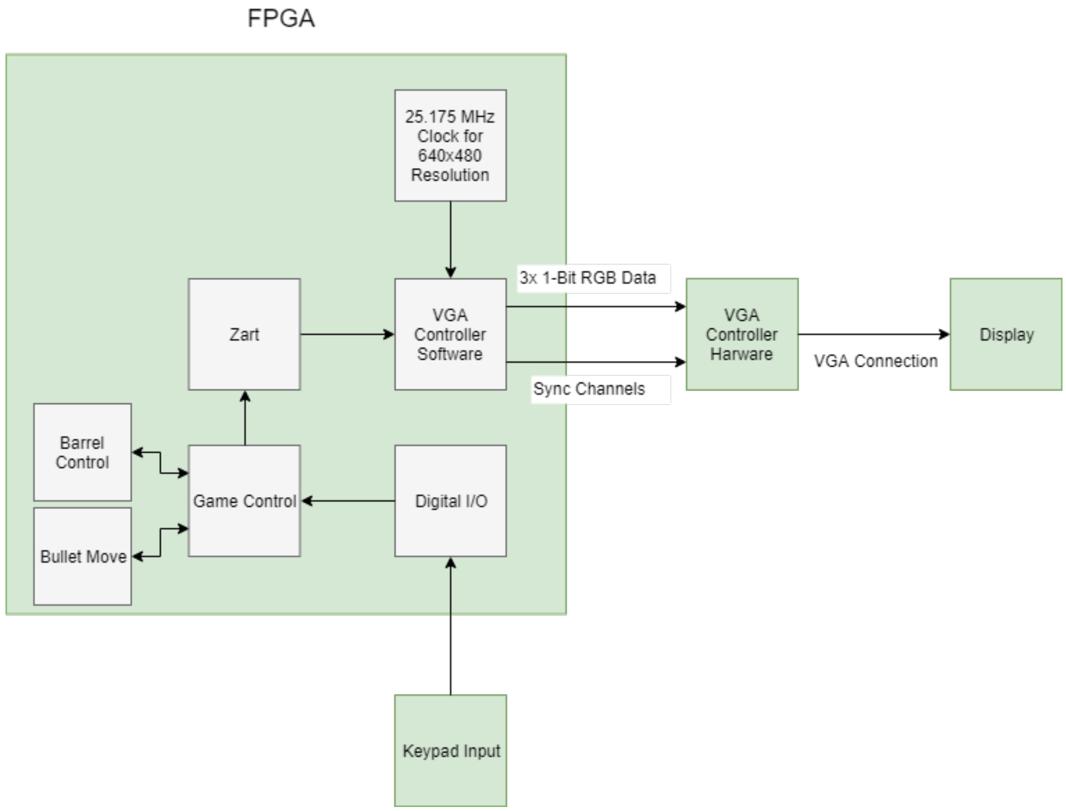


Figure 2: Project block diagram

The game used push buttons to change the state of each tank. One set was to move the player to the next stage of shooting and the other was to select the parameter. The flow of these operation is depicted in Figure 3.

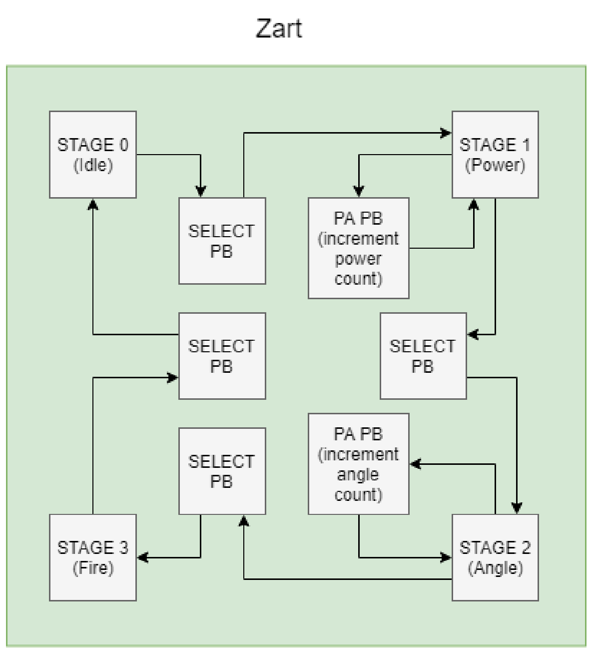


Figure 3: Zart Game State Diagram

We felt that the game would look more refined in the bullets moved pixel by pixel and not jump across the screen. So we made two counters counting from a 2kHz clock, each having max values that changed depending on how fast the bullet was supposed to move. The Y direction counter would update the max count when the ball moved. It would update to ever increasing values to simulate the bullet slowing down in the y direction while rising and speed up in the y direction as the bullet fell. Theoretically for the bullet to completely stop, the counter would need to count to infinity. To get around this we did trials to see at what counter values the ball appeared to have adequately stopped. The code would test for the Y counter max to exceed this value and then the direction would change. This process is visualized in Figure 4.

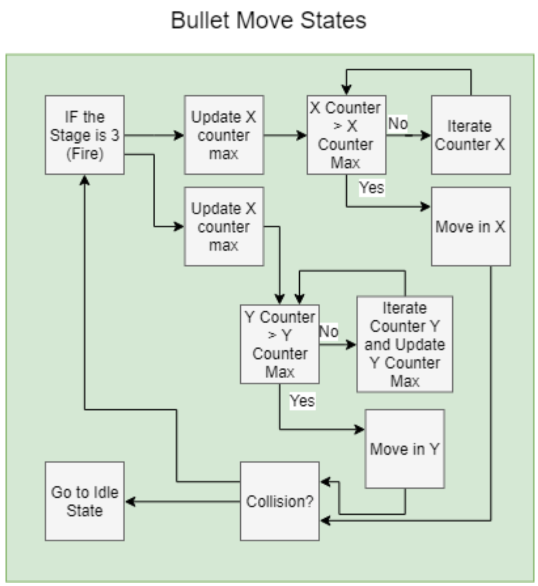


Figure 4: Bullet Moving State Diagram

3 Display Hardware

The hardware required for this project was somewhat minimalist. For basic implementation of 8 colour display, all that was required was a VGA female connector, and $3 \times 270\Omega$ resistors[2]. The purpose of the resistors was to create a voltage divider with the 75Ω monitor termination in order to deliver 0-0.7V from our 3.3V outputs on the DE0 Nano FPGA.

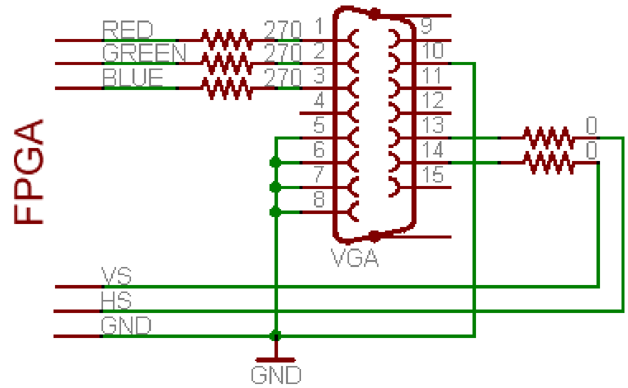


Figure 5: VGA Connector diagram showing minimal connections required

4 VGA Software

We would like to start this section by acknowledging Jean Nicolle's work at www.fpga4fun.com. Without his explanation of the VGA protocol, this project would have been far more difficult.

4.1 Signal Timing

In order to display content onto a VGA monitor proper timing information must be sent over the five signal wires as seen in Figure 7. Choosing this timing depends on your desired display resolution. In an effort to keep things simple we chose a resolution of 640x480. With this information, you can determine the signal information required by looking at a parameter chart like Figure 6.

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31
640x480, 72Hz	31.500	640	24	40	128	480	9	3	28
640x480, 75Hz	31.500	640	16	96	48	480	11	2	32
640x480, 85Hz	36.000	640	32	48	112	480	1	3	25
800x600, 56Hz	38.100	800	32	128	128	600	1	4	14
800x600, 60Hz	40.000	800	40	128	88	600	1	4	23
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23
800x600, 75Hz	49.500	800	16	80	160	600	1	2	21
800x600, 85Hz	56.250	800	32	64	152	600	1	3	27
1024x768, 60Hz	65.000	1024	24	136	160	768	3	6	29
1024x768, 70Hz	75.000	1024	24	136	144	768	3	6	29
1024x768, 75Hz	78.750	1024	16	96	176	768	1	3	28
1024x768, 85Hz	94.500	1024	48	96	208	768	1	3	36

Figure 6: This table lists timing values for several popular resolutions[2].

The selected resolution requires a pixel clock of approximately 25.175 MHz. According to our reference, this frequency does not have to be exact due to the prevalence of 'multisync' monitors that can adjust themselves to frequency differences. The front and back porch values did not have to be set manually as long as we held our sync pulses low long enough.

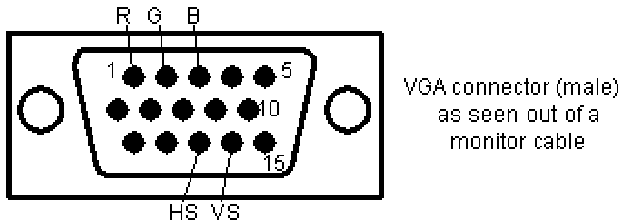


Figure 7: VGA Signal Channels[2]

Once the pixel clock has been selected, two slower pulses can be generated to control drawing on the screen. The pixel clock traces across the screen and when it reaches the end, an active low h_sync pulse triggers telling the device to go to the next line. This happens at about 32kHz. Once all the lines have been drawn, an active low v_sync pulse triggers telling the monitor to go back to the top and start again. The timing of this pulse is often referred to as refresh rate, and in our case it was 60Hz. See Figure 11.

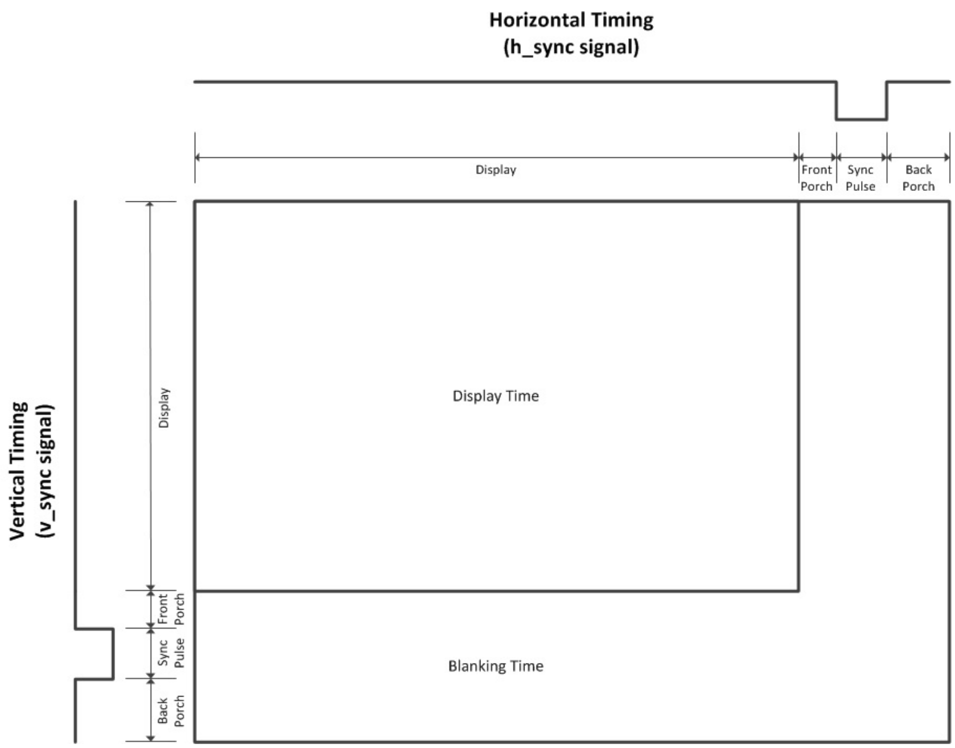


Figure 8: Signal timing for VGA Signal[3]

4.2 Drawing with VGA

Drawing to VGA was achieved using 3 colour channels. This gave us 3 bits to work with, resulting in 8 available colours. Rather than using up the resources required to implement a 640x480 3 colour-channel array, we opted to use conditional statements in combination with the X and Y counters to draw objects to each of the 3 channels. In this way, an object could be stored as a number range rather than a lengthy array of values. For example, to draw a simple red rectangle, the RGB output code would look something like the following:

$R = (CounterX >= leftbound)\&\&(CounterX <= rightbound)\&\&(CounterY >= topbound)\&\&(CounterY <= bottombound);$

$G = 0;$

$B = 0;$

Using the same idea an equation for a line could be used as a test statement for drawing lines. The only problem with this method would be lines with changing slopes. An example of defining conditions for changing slopes can be seen in the program listing in section 6.8.

4.3 Example Graphics

The following pictures are of various in game graphics generated using conditional statements. See the vga.sv program listing for the code required to draw a BLOCK, POINT, LINEPOS, and LINENEG.

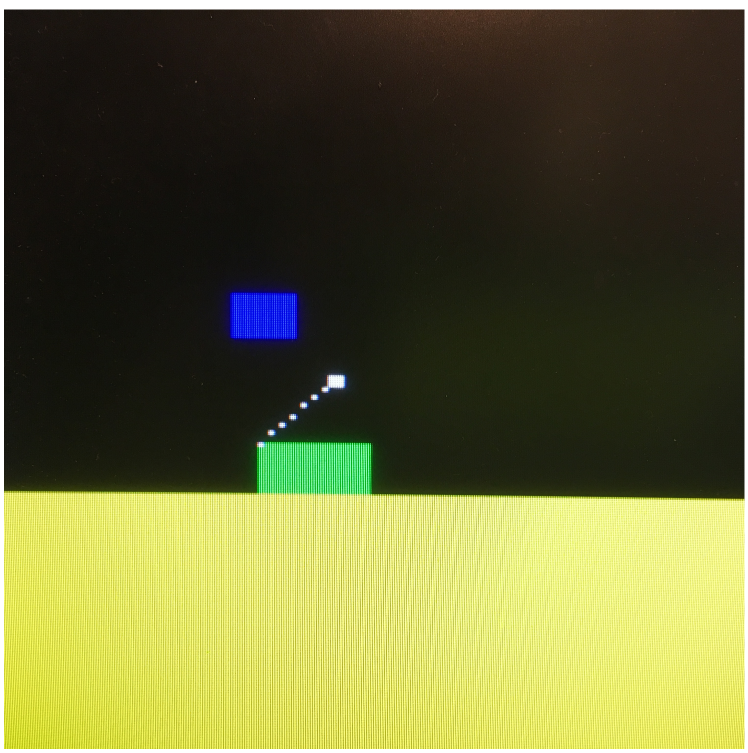


Figure 9: Tank Image Depicting using LINEPOS, and BLOCK close up

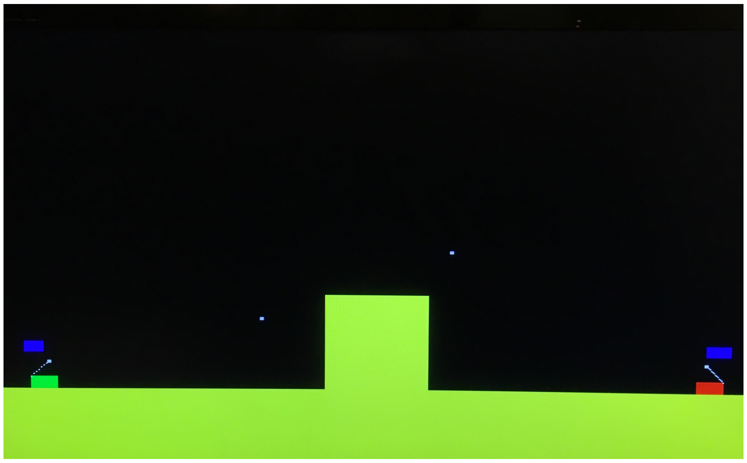


Figure 10: Image Depicting using LINEPOS, LINENEG, POINT, and BLOCK

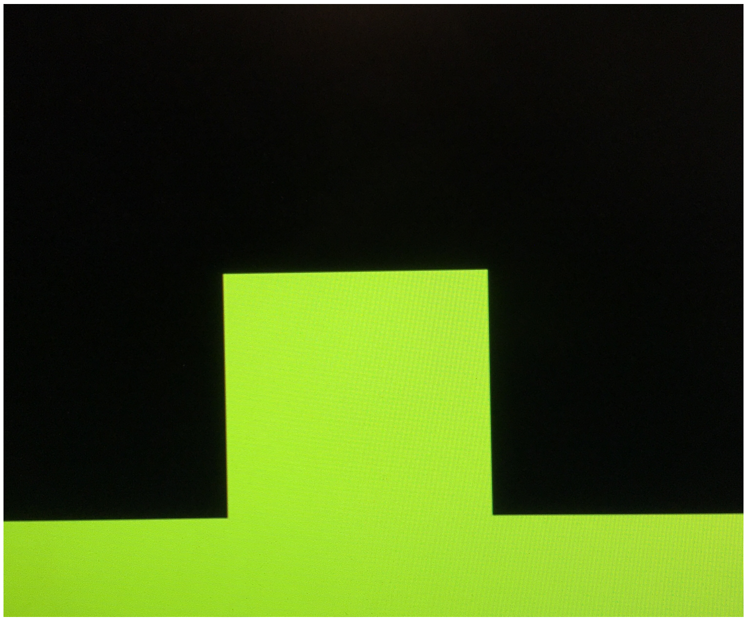


Figure 11: Image for the hill Depicting the usage of BLOCK

5 Suggestions for Future Work

Some additional items we would have likes to implement given more time were the following:

5.1 Randomly Generated Wind

This could be implemented by creating a variable that gets randomized in magnitude and left or right direction at the beginning of the round. The variable would then get added or subtracted from a player's bullet speed during the firing state. This would manifest itself as a constant accelerating force that would influence the trajectories of the projectiles.

5.2 Moving Tanks

This could easily be accomplished by adding more player state variables to include an overall tank X position. The position could be changed using additional buttons. This would increase competitiveness of gameplay.

5.3 Destroyable Terrain

This would involve some work implementing the FPGA's larger memory to store each and every pixel of the terrain in a large array. If an intersection between a bullet and the terrain was detected, those specific pixels could be removed from the array.

5.4 Higher Resolution Colour

The initial planning of this project involved using a 10 bit video DAC to implement higher resolution colour. Due to timing constraints, this was not achieved. An easier way of increasing colour resolution in the future would be to use a resistor network to create a 3 bit DAC. This would be easily done by adding an extra two outputs per colour channel and then summing the 3 together using a buffer. More information can be found online by searching 3 or 4 bit resistor DAC.

References

[1] DOS Games Archive, "Zart - zee artillery game," accessed 2018-04-12. [Online]. Available: https://archive.org/details/ZARTZeeArtilleryGame_1020

[2] J. P. Nicolle, "Pong Game," accessed 2018-04-12. [Online]. Available: http://www.fpga4fun.com/PongGame.html

[3] S. Larson, "VGA Controller (VHDL)," accessed 2018-04-12. [Online]. Available: https://eewiki.net/pages/viewpage.action?pageId=15925278

6 Program Listings

6.1 zart.sv

```
module zart(input logic clk_50MHz,
            output logic clk,
            (* altera_attribute = "-name WEAK_PULL_UP_RESISTOR ON" *)
            input logic [3:0] kpr,
            output logic [3:0] kpc,
            output logic [7:0] LED,
            output logic vga_h_sync, vga_v_sync, vga_R, vga_G, vga_B
            );

   logic clk_4Hz;
   logic kphit; // flag for keypad hit
   logic [15:0] num; //keypad register

   // shifted values ( (0,0) top left )
   logic [11:0] pos_x_i_p1;
   logic [11:0] pos_y_i_p1;
   logic [11:0] pos_x_i_p2;
   logic [11:0] pos_y_i_p2;
   logic [11:0] pos_x_p1;
   logic [11:0] pos_y_p1;
   logic [11:0] pos_x_p2;
   logic [11:0] pos_y_p2;

   // unshifted values ( (0,0) bottom left)
   logic [11:0] pos_y_i_p1_tmp;
   logic [11:0] pos_y_i_p2_tmp;
   logic [11:0] pos_y_p1_tmp;
   logic [11:0] pos_y_p2_tmp;

   logic [11:0] shift = 480; // the shift value

   // shift the values by the shift value
   always_comb begin
      pos_y_i_p1 = shift - pos_y_i_p1_tmp;
      pos_y_i_p2 = shift - pos_y_i_p2_tmp;
      pos_y_p1 = shift - pos_y_p1_tmp;
      pos_y_p2 = shift - pos_y_p2_tmp;
   end



   clockgen2 clockgen(clk_50MHz, clk_25MHz, clk_4kHz);
   clockdiv clockdiv_0(clk_4kHz,clk_4Hz);
   colseq colseq_0(kpr, kpc, clk_4kHz);
   kpdecode kpdecode_0(kpr, kpc, kphit, num);

   Game_Control GAME_0( num[3],
                        num[13],
                        num[0],
                        num[6],
                        pos_x_i_p1,
                        pos_y_i_p1_tmp,
                        pos_x_i_p2,
                        pos_y_i_p2_tmp,
                        pos_x_p1,
                        pos_y_p1_tmp,
                        pos_x_p2,
                        pos_y_p2_tmp);

   vga vga( pos_x_p1,
            pos_y_p1,
            pos_x_p2,
            pos_y_p2,
            pos_x_i_p1,
            pos_y_i_p1,
            pos_x_i_p2,
            pos_y_i_p2,
            clk_50MHz,
            vga_h_sync,
            vga_v_sync,
            vga_R,
            vga_G,
            vga_B);

   // turn on leds on fpga when a key on the keypad is hit.
```

```
   assign LED = { num[13],
                  num[6],
                  num[3],
                  num[0],
                  4'b0000};

   assign clk = clk_4Hz;
endmodule
```

6.2 Game_Control.sv

```
module Game_Control( input logic pb_select_p1, // player 1 stage select button
                     input logic pb_select_p2, // player 2 stage select button
                     input logic pb_pa_p1, // player 1 power or angle select button
                     input logic pb_pa_p2, // player 2 power or angle select button
                     output logic [11:0] pos_x_i_p1, // player 1 barrel x postion
                     output logic [11:0] pos_y_i_p1, // player 1 barrel y postion
                     output logic [11:0] pos_x_i_p2, // player 2 barrel x postion
                     output logic [11:0] pos_y_i_p2, // player 2 barrel y postion
                     output logic [11:0] pos_x_p1, // player 1 ball x postion
                     output logic [11:0] pos_y_p1, // player 1 ball y postion
                     output logic [11:0] pos_x_p2,  // player 2 ball x postion
                     output logic [11:0] pos_y_p2);  // player 2 ball y postion

   // temporary values for all the modules
   logic [3:0] stage_p1 = 0;
   logic [3:0] stage_p2 = 0;
   logic [3:0] stage_p1_n = 0;
   logic [3:0] stage_p2_n = 0;

   logic [11:0] angle_x_p1;
   logic [11:0] angle_y_p1;
   logic [11:0] angle_x_p2;
   logic [11:0] angle_y_p2;

   logic [11:0] angle_div;
   logic [11:0] angle_div2;

   logic [3:0] power_p1;
   logic [3:0] power_p2;

   // flags to detect any out of bounds balls
   logic collision_p1;
   logic collision_p2;

   // flags to detect wins or self deaths
   logic [1:0] winner_p1;
   logic [1:0] winner_p2;

   // barrel and bullet control modules for player 1
   // 1 = because this tank shoots right
   // (50,93) = position of the tank for player 1
   BarrelControl #(1,50,93) BC_p1( pb_pa_p1,
                           pb_select_p1,
                           stage_p1,
                           pos_x_i_p1,
                           pos_y_i_p1,
                           angle_x_p1,
                           angle_y_p1,
                           angle_div,
                           power_p1);
   // 1  = becuase the bullets move to the right
   BulletMove #( 1 ) BM_p1( clk_2kHz,
                     power_p1,
                     angle_x_p1,
                     angle_y_p1,
                     angle_div,
                     stage_p1,
                     pos_x_i_p1,
                     pos_y_i_p1,
                     pos_x_p1,
                     pos_y_p1,
                     collision_p1,
                     winner_p1);

   // barrel and bullet control modules for player 2
   // -1 = because this tank shoots left
   // (589,93) = position of the tank for player 2
   BarrelControl #(-1,589,93) BC_p2 ( pb_pa_p2,
                            pb_select_p2,
                            stage_p2,
                            pos_x_i_p2,
                            pos_y_i_p2,
                            angle_x_p2,
                            angle_y_p2,
                            angle_div2,
                            power_p2);
   // 0  = becuase the bullets move to the left
   BulletMove #( 0 ) BM_p2( clk_2kHz,
```

```
                        power_p2,
                        angle_x_p2,
                        angle_y_p2,
                        angle_div2,
                        stage_p2,
                        pos_x_i_p2,
                        pos_y_i_p2,
                        pos_x_p2,
                        pos_y_p2,
                        collision_p2,
                        winner_p2);

   // changes the stage of player 1
   // the stage can change when the player push the pb_select or
   //    the bullet goes out of bounds or
   //    the player hits themselves or their enemy
   always @(pb_select_p1 or collision_p1 or winner_p1 or winner_p2) begin
      if(collision_p1 == 1) begin
         stage_p1_n = 0;
      end else begin
         if(winner_p1 == 1 || winner_p1 == 2 || winner_p2 == 1 || winner_p2 == 2) begin
            stage_p1_n = 0;
         end else begin
            if(pb_select_p1 == 1) begin
               if(stage_p1 >= 4) begin
                  stage_p1_n = 0;
               end else begin
                  stage_p1_n = stage_p1 + 1;
               end
            end
         end
      end
   end

   // changes the stage of player 2
   // the stage can change when the player push the pb_select or
   //    the bullet goes out of bounds or
   //    the player hits themselves or their enemy
   always @(pb_select_p2 or collision_p2 or winner_p2 or winner_p2) begin
      if(collision_p2 == 1) begin
         stage_p2_n = 0;
      end else begin
         if(winner_p1 == 1 || winner_p1 == 2 || winner_p2 == 1 || winner_p2 == 2) begin
            stage_p2_n = 0;
         end else begin
            if(pb_select_p2 == 1) begin
               if(stage_p2 >= 4) begin
                  stage_p2_n = 0;
               end else begin
                  stage_p2_n = stage_p2 + 1;
               end
            end
         end
      end
   end

   always_comb begin
      stage_p1 = stage_p1_n;
      stage_p2 = stage_p2_n;
   end

endmodule
```

6.3 BarrelControl.sv

```
module BarrelControl #(DIR=1,x_i=240,y_i=320) // dir = 1 for right, dir = 0 for left
                                   // x_i = the x postion of the tank
                                   // y_i = the y postion of the tank
               (input logic pb_pa, // push button to choose the next power or angle
               input logic pb_select, // psuh button to lock in power or angle or fire
               input logic [3:0] stage, // current stage: idle, power select,
                                 // angle select, fire
               output logic [11:0] pos_x_i, // initial x position of the ball,
                                    // and the barrel end
               output logic [11:0] pos_y_i, // initial y position of the ball,
                                     // and the barrel end
               output logic [11:0] angle_x, // scaled initial angle for the
                                    // ball in the x dir
               output logic [11:0] angle_y, // scaled initial angle for the
                                    // ball in the y dir
               output logic [11:0] angle_div, // scaling factor for the angles
               output logic [3:0] power); // selected power, 1 - 10

   logic [7:0] barrel_length = 20; // barrel length in pixels
   logic [3:0] power_n = 0; // counter for the pb_pa push button to select power
   logic [11:0] angle_n = 0; // counter for the pb_pa push button to select angle

   initial begin
      angle_x = 1000*DIR;
      angle_y = 0;
      power = 0;
      angle_div = 1000;
   end

   // gives the calculates the position of the barrel, which is also
   //    the initial position of the ball
   always_comb begin
      if(DIR == 1 || DIR==-1) begin
         pos_x_i = x_i + (barrel_length*angle_x)/angle_div;
         pos_y_i = y_i + (barrel_length*angle_y)/angle_div;
      end else begin
         pos_x_i = 640/2;
         pos_y_i = 480/2;
      end
   end

   // selects the angle or power
   // if the stage is 1 (power select) then the counter is outputed as the power
   // if the stage is 2 (angle select) then a scaled angle for each direction is given
   always @(posedge pb_select) begin
      unique case (stage)
         1: power <= power_n;
         2: begin
            unique case(angle_n)
               1: begin //0 deg
                  angle_x <= 1000*DIR;
                  angle_y <= 0;
               end
               2: begin // 9 deg
                  angle_x <= 988*DIR;
                  angle_y <= 156;
               end
               3: begin // 18 deg
                  angle_x <= 951*DIR;
                  angle_y <= 309;
               end
               4: begin // 27 deg
                  angle_x <= 891*DIR;
                  angle_y <= 454;
               end
               5: begin // 36 deg
                  angle_x <= 809*DIR;
                  angle_y <= 588;
               end
               6: begin // 45 deg
                  angle_x <= 707*DIR;
                  angle_y <= 707;
               end
               7: begin // 54 deg
                  angle_x <= 588*DIR;
                  angle_y <= 809;
               end
               8: begin // 63 deg
```

```
                    angle_x <= 454*DIR;
                    angle_y <= 891;
                end
                9: begin // 72 deg
                    angle_x <= 309*DIR;
                    angle_y <= 951;
                end
                10: begin // 81 deg
                    angle_x <= 156*DIR;
                    angle_y <= 988;
                end
                default: begin
                    angle_x <= 1000*DIR;
                    angle_y <= 0;
                end
            endcase
        end
        default: begin
            angle_x <= 1000*DIR;
            angle_y <= 0;
            power <= 0;
        end
    endcase
end

// depending on the stage pushing the pb_pa push button
//    either increases the power or angle counter
always @(posedge pb_pa) begin
    unique case (stage)
        1: begin
            if(power_n > 10) begin
                power_n <= 10;
            end else begin
                power_n <= power_n + 1;
            end
        end
        2: begin
            if(angle_n > 10) begin
                angle_n <= 10;
            end else begin
                angle_n <= angle_n + 1;
            end
        end
        default: begin
            power_n <= 0;
            angle_n <= 0;
        end
    endcase

end

endmodule
```

6.4 BulletMove.sv

```
module BulletMove    #( DIR=1 ) // 1 = ball traveling to the right, 0 = left
                (input logic clk_2kHz,
                input logic [3:0] power, // scaled 1 - 10 of how fast the ball travels
                input logic [15:0] angle_x, // the balls scaled initial
                                   //    velocity in the x dir
                input logic [15:0] angle_y,  // the balls scaled initial
                                   //    velocity in the y dir
                input logic [15:0] angle_div, // velocity scaling factor
                input logic [3:0] stage, // current stage of this player
                input logic [11:0] pos_x_i, // inital position of the ball in x dir
                input logic [11:0] pos_y_i, // inital position of the ball in y dir
                output logic [11:0] pos_x, // current position of the ball in x dir
                output logic [11:0] pos_y, // current position of the ball in y dir
                output logic collision, // flag for a collision
                output logic [1:0] winner); // flag for win or lose

   logic [19:0] CMf = 10; // the square of counter max for the bullet speed
   logic [19:0] ans = 0; // answer to a sqrt operation for the counter max

   assign CMf = 4000000 / (4000000/BPPS_PERIOD_y_i/BPPS_PERIOD_y_i - 2*40*(pos_y-pos_y_i));

   sqrt sqrt_0(CMf, ans);

   int BPPS_PERIOD_x = 6; // counter max for the x direction
   int BPPS_PERIOD_y = 6; // counter max for the y direction
   int BPPS_PERIOD_y_i = 6; // initial counter max for the y direction
   logic [15:0] BPPS_count_x = 1; // counter to count up to the time to move in x
   logic [15:0] BPPS_count_y = 1; // counter to count up to the time to move in y
   logic [15:0] BPPS_count_x_next = 1; // temp for BPPS_count_x
   logic [15:0] BPPS_count_y_next = 1; // temp for BPPS_count_y

   logic B_move_x = 0; // a flag to move the ball in the x direction
   logic B_move_y = 0; // a flag to move the ball in the y direction

   logic y_dir = 1; // changes to -1 when the ball begins to fall

   // this is always block is to detect when the ball has moved out of bounds or
   //    hit a player
   always @(pos_y or pos_x) begin
      if(stage == 3) begin // stage 3 is the fired ball pahse
         if(pos_y > 479 || pos_y < 80 ) begin
            collision <= 1;
         end else begin
            if(pos_x > 639 || pos_x < 1) begin
               collision <= 1;
            end else begin
               if(pos_x > 280 && pos_x < 360 && pos_y < 180) begin
                  collision <= 1;
               end else begin
                  if( DIR ) begin
                     if(pos_x > 50 && pos_x < 70 && pos_y > 80 && pos_y < 93) begin
                        winner <= 2;
                     end else begin
                        if(pos_x > 569 && pos_x < 589 && pos_y > 80 && pos_y < 93)
                        begin
                           winner <= 1;
                        end else begin
                           collision <= 0;
                           winner <= 0;
                        end
                     end
                  end else begin
                     if(pos_x > 50 && pos_x < 70 && pos_y > 80 && pos_y < 93) begin
                        winner <= 1;
                     end else begin
                        if(pos_x > 569 && pos_x < 589 && pos_y > 80 && pos_y < 93)
                        begin
                           winner <= 2;
                        end else begin
                           collision <= 0;
                           winner <= 0;
                        end
                     end
                  end
               end
            end
         end
      end else begin
```

```
            collision <= 0;
            winner <= 0;
         end
      end


      // signals a move then the direction counters achieve the right count
      always @(*) begin
         if(BPPS_count_x >= BPPS_PERIOD_x) begin
            B_move_x = ~B_move_x;
            BPPS_count_x_next = 1;
         end else begin
            BPPS_count_x_next = BPPS_count_y + 1;
         end

         if(BPPS_count_y >= BPPS_PERIOD_y) begin
            B_move_y = ~B_move_y;
            BPPS_count_y_next = 1;
         end else begin
            BPPS_count_y_next = BPPS_count_y + 1;
         end
      end

      // calculates the period of the counters depending on the
      //    angles and power of each direction
      always @(posedge clk_2kHz) begin
         if(stage == 2) begin
            BPPS_PERIOD_x <= ((4*2000*10*angle_div)/(905*angle_x*power));
            BPPS_PERIOD_y_i <= ((4*2000*10*angle_div)/(905*angle_y*power));
            BPPS_count_x <= 1;
            BPPS_count_y <= 1;
         end else begin
            if(stage == 3) begin
               BPPS_count_x <= BPPS_count_x_next;
               BPPS_count_y <= BPPS_count_y_next;
               if(ans > 150) begin
                  BPPS_PERIOD_y <= 150;
                  y_dir = 0;
               end else begin
                  BPPS_PERIOD_y <= ans;
                  y_dir = y_dir;
               end

            end else begin
               y_dir = 1;
               BPPS_count_x <= 1;
               BPPS_count_y <= 1;
            end
         end
      end

      // moves the ball in the x dir when the counter finishes
      // if the player push pbselect the ball will move back to the tank barrel
      // DIR:
      // 1 = moves bullet right
      // 0 = moves bullet left
      always @(posedge B_move_x) begin

         if(stage == 3) begin
            if(DIR) begin
               pos_x <= pos_x + 1;
            end else begin
               pos_x <= pos_x - 1;
            end
         end else begin
            pos_x <= pos_x_i;
         end
      end

      // moves the ball in the y dir when the counter finishes
      // if the player push pbselect the ball will move back to the tank barrel
      always @(posedge B_move_y) begin

         if(stage == 3) begin
            if(y_dir) begin
               pos_y <= pos_y + 1;
            end else begin
               pos_y <= pos_y - 1;
            end
         end else begin
```

```
            pos_y <= pos_y_i;
        end
    end

endmodule
```

6.5 sqrt.sv

```
module sqrt(input logic [19:0] num, output logic [19:0] ans);

   logic [19:0] min = 0;
   logic [19:0] max = 0;
   logic [19:0] mid = 0;
   logic [19:0] guess = 0;
   logic [19:0] count = 0;

   // uses a successive approximation to find the sqrt or a number
   always @(*) begin
      min = 0;
      max = num;
      count = 0;

      while((max - min) > 1 && count < 50) begin
         mid = (max + min) / 2;
         guess = (mid*mid);
         if(guess <= num) begin
            min = mid;
         end
         if(guess >= num) begin
            max = mid;
         end
         count = count + 1;
      end

      ans = min;
   end

endmodule
```

6.6 colseq.sv

```
// colseq.sv - ELEX 7660
// Kolyn.Gray 2018-01-22

module colseq (input logic [3:0] kpr, output logic [3:0] kpc, input logic clk);

   // if no rows are low cycle through making the columns low
   always_ff@(posedge clk) begin
      if(kpr == 4'b1111)
      begin
         unique case (kpc)
            4'b0111: kpc = 4'b1011;
            4'b1011: kpc = 4'b1101;
            4'b1101: kpc = 4'b1110;
            4'b1110: kpc = 4'b0111;
            default: kpc = 4'b1011;
         endcase
      end
   end

endmodule
```

6.7 kpdecode.sv

```
// kpdecode.sv - ELEX 7660
// Kolyn.Gray 2018-01-22

module kpdecode ( input logic [3:0] kpr,
              input logic [3:0] kpc,
              output logic kphit,
              output logic [15:0] num);

   // allows for 4 buttons to be pressed in the same column at once
   // the idea being depending on the rows that are low
   // and using the current columns that are low you can determine what buttons are pressed
   always_comb begin
   if(kpr != 4'b1111 && kpc != 4'b1111)
   begin
         kphit <= 1;
         unique case (kpr)
            4'b0000:

               unique case (kpc)
                  4'b0111: num = 16'b0010_0000_0100_1001;
                  4'b1011: num = 16'b0100_0000_1001_0010;
                  4'b1101: num = 16'b1000_0001_0010_0100;
                  4'b1110: num = 16'b0001_1110_0000_0000;
                  default: num = 16'hx;
               endcase
            4'b0001:
               unique case (kpc)
                  4'b0111: num = 16'b0000_0000_0100_1001;
                  4'b1011: num = 16'b0000_0000_1001_0010;
                  4'b1101: num = 16'b0000_0001_0010_0100;
                  4'b1110: num = 16'b0000_1110_0000_0000;
                  default: num = 16'hx;
               endcase
            4'b0010:
               unique case (kpc)
                  4'b0111: num = 16'b0010_0000_0000_1001;
                  4'b1011: num = 16'b0100_0000_0001_0010;
                  4'b1101: num = 16'b1000_0000_0010_0100;
                  4'b1110: num = 16'b0001_0110_0000_0000;
                  default: num = 16'hx;
               endcase
            4'b0011:
               unique case (kpc)
                  4'b0111: num = 16'b0000_0000_0000_1001;
                  4'b1011: num = 16'b0000_0000_0001_0010;
                  4'b1101: num = 16'b0000_0000_0010_0100;
                  4'b1110: num = 16'b0000_0110_0000_0000;
                  default: num = 16'hx;
               endcase
            4'b0100:
               unique case (kpc)
                  4'b0111: num = 16'b0010_0000_0100_0001;
                  4'b1011: num = 16'b0100_0000_1000_0010;
                  4'b1101: num = 16'b1000_0001_0000_0100;
                  4'b1110: num = 16'b0001_1010_0000_0000;
                  default: num = 16'hx;
               endcase
            4'b0101:
               unique case (kpc)
                  4'b0000: num = 16'b0000_1011_1100_0111;
                  4'b0001: num = 16'b0000_0001_1100_0111;
                  4'b0010: num = 16'b0000_1010_1100_0011;
                  4'b0011: num = 16'b0000_0000_1100_0011;
                  4'b0100: num = 16'b0000_1011_0100_0101;
                  4'b0101: num = 16'b0000_0001_0100_0101;
                  4'b0110: num = 16'b0000_1010_0100_0001;
                  4'b0111: num = 16'b0000_0000_0100_0001;
                  4'b1000: num = 16'b0000_1011_1000_0110;
                  4'b1001: num = 16'b0000_0001_1000_0110;
                  4'b1010: num = 16'b0000_1010_1000_0010;
                  4'b1011: num = 16'b0000_0000_1000_0010;
                  4'b1100: num = 16'b0000_1011_0000_0100;
                  4'b1101: num = 16'b0000_0001_0000_0100;
                  4'b1110: num = 16'b0000_1010_0000_0000;
                  4'b1111: num = 16'b0000_0000_0000_0000;
                  default: num = 16'hx;
               endcase
            4'b0110:
```

```
                unique case (kpc)
                    4'b0111: num = 16'b0010_0000_0000_0001;
                    4'b1011: num = 16'b0100_0000_0000_0010;
                    4'b1101: num = 16'b1000_0000_0000_0100;
                    4'b1110: num = 16'b0001_0010_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b0111:
                unique case (kpc)
                    4'b0000: num = 16'b0000_0010_0000_0111;
                    4'b0001: num = 16'b0000_0000_0000_0111;
                    4'b0010: num = 16'b0000_0010_0000_0011;
                    4'b0011: num = 16'b0000_0000_0000_0011;
                    4'b0100: num = 16'b0000_0010_0000_0101;
                    4'b0101: num = 16'b0000_0000_0000_0101;
                    4'b0110: num = 16'b0000_0010_0000_0001;
                    4'b0111: num = 16'b0000_0000_0000_0001;
                    4'b1000: num = 16'b0000_0010_0000_0110;
                    4'b1001: num = 16'b0000_0000_0000_0110;
                    4'b1010: num = 16'b0000_0010_0000_0010;
                    4'b1011: num = 16'b0000_0000_0000_0010;
                    4'b1100: num = 16'b0000_0010_0000_0100;
                    4'b1101: num = 16'b0000_0000_0000_0100;
                    4'b1110: num = 16'b0000_0010_0000_0000;
                    4'b1111: num = 16'b0000_0000_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b1000:
                unique case (kpc)
                    4'b0111: num = 16'b0010_0000_0100_1000;
                    4'b1011: num = 16'b0100_0000_1001_0000;
                    4'b1101: num = 16'b1000_0001_0010_0000;
                    4'b1110: num = 16'b0001_1100_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b1001:
                unique case (kpc)
                    4'b0111: num = 16'b0000_0000_0100_1000;
                    4'b1011: num = 16'b0000_0000_1001_0000;
                    4'b1101: num = 16'b0000_0001_0010_0000;
                    4'b1110: num = 16'b0000_1100_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b1010:
                unique case (kpc)
                    4'b0111: num = 16'b0010_0000_0000_1000;
                    4'b1011: num = 16'b0100_0000_0001_0000;
                    4'b1101: num = 16'b1000_0000_0010_0000;
                    4'b1110: num = 16'b0001_0100_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b1011:
                unique case (kpc)
                    4'b0111: num = 16'b0000_0000_0000_1000;
                    4'b1011: num = 16'b0000_0000_0001_0000;
                    4'b1101: num = 16'b0000_0000_0010_0000;
                    4'b1110: num = 16'b0000_0100_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b1100:
                unique case (kpc)
                    4'b0111: num = 16'b0010_0000_0100_0000;
                    4'b1011: num = 16'b0100_0000_1000_0000;
                    4'b1101: num = 16'b1000_0001_0000_0000;
                    4'b1110: num = 16'b0001_1000_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b1101:
                unique case (kpc)
                    4'b0000: num = 16'b0000_1001_1100_0000;
                    4'b0001: num = 16'b0000_0001_1100_0000;
                    4'b0010: num = 16'b0000_1000_1100_0000;
                    4'b0011: num = 16'b0000_0000_1100_0000;
                    4'b0100: num = 16'b0000_1001_0100_0000;
                    4'b0101: num = 16'b0000_0001_0100_0000;
                    4'b0110: num = 16'b0000_1000_0100_0000;
                    4'b0111: num = 16'b0000_0000_0100_0000;
                    4'b1000: num = 16'b0000_1001_1000_0000;
                    4'b1001: num = 16'b0000_0001_1000_0000;
                    4'b1010: num = 16'b0000_1000_1000_0000;
```

```
                    4'b1011: num = 16'b0000_0000_1000_0000;
                    4'b1100: num = 16'b0000_1001_0000_0000;
                    4'b1101: num = 16'b0000_0001_0000_0000;
                    4'b1110: num = 16'b0000_1000_0000_0000;
                    4'b1111: num = 16'b0000_0000_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b1110:
                unique case (kpc)
                    4'b0111: num = 16'b0010_0000_0000_0000;
                    4'b1011: num = 16'b0100_0000_0000_0000;
                    4'b1101: num = 16'b1000_0000_0000_0000;
                    4'b1110: num = 16'b0001_0000_0000_0000;
                    default: num = 16'hx;
                endcase
            4'b1111: num = 16'b0000_0000_0000_0000;
            default: num = 16'hx;
        endcase
    end
    else
    begin
        kphit <= 0;
        num = 16'hx;
    end
end

endmodule
```

6.8 vga.sv

```
//bitwise operation to define a rectangle during the sweep
‘define BLOCK(a) (((CounterX >= ‘‘a‘‘.l )&&(CounterX <= ‘‘a‘‘.r ))\
            &&((CounterY >= ‘‘a‘‘.t )&&(CounterY <= ‘‘a‘‘.b )))

//bitwise operation to draw a rectangular point with a "radius" of 2*thickness
‘define POINT(x,y,thickness) ((CounterX >= x - thickness )&&(CounterX <= x + thickness ))\
                     &&((CounterY >= y - thickness )&&(CounterY <= y + thickness ))

//line drawing algorithm for positive slopes
‘define LINEPOS(line,thickness) (\
      (CounterY ==(‘‘line‘‘.y2 + (‘‘line‘‘.my*(CounterX -‘‘line‘‘.x2))/‘‘line‘‘.mx))\
      && ((CounterX >= ‘‘line‘‘.x1) && (CounterX <=‘‘line‘‘.x2))\
      && (CounterY >= ‘‘line‘‘.y2) && (CounterY <=‘‘line‘‘.y1))

//line drawing algorithm for negative slopes
‘define LINENEG(line,thickness) (\
      (CounterY ==(‘‘line‘‘.y1 + (‘‘line‘‘.my*(CounterX -‘‘line‘‘.x1))/‘‘line‘‘.mx))\
      && ((CounterX >= ‘‘line‘‘.x1) && (CounterX <=‘‘line‘‘.x2))\
      && (CounterY >= ‘‘line‘‘.y1) && (CounterY <=‘‘line‘‘.y2))

module vga(bulletx,bullety,barrel1x,barrel1y,barrel2x,barrel2y,
            clk, vga_h_sync, vga_v_sync, vga_R, vga_G, vga_B);

   //Position variables
   input logic[9:0] bulletx;
   input logic[8:0] bullety;
   input logic[9:0] barrel1x;
   input logic[8:0] barrel1y;
   input logic[9:0] barrel2x;
   input logic[8:0] barrel2y;
   input logic clk;
   output vga_h_sync, vga_v_sync, vga_R, vga_G, vga_B;

   //modified sync generator from fpga4fun.com pong code
   hvsync_generator syncgen(  .CLOCK_50(clk),
                        .vga_h_sync(vga_h_sync),
                        .vga_v_sync(vga_v_sync),
                        .inDisplayArea(inDisplayArea),
                        .CounterX(CounterX),
                        .CounterY(CounterY));


   ///////////////////////////
   logic inDisplayArea; //allows writing to only 640x480 area as seen below
   logic [9:0] CounterX; //x write position
   logic [8:0] CounterY; //y write position

   //rectangles go left right top bottom
   typedef struct {
      logic[9:0] l;
      logic[9:0] r;
      logic[8:0] t;
      logic[8:0] b;
      logic[2:0] ob_col; //3 bit RGB
   } rect;


   //lines work better if pt1 is to the left of pt2
   typedef struct {
      logic[9:0] x1;
      logic[9:0] x2;
      logic[8:0] y1;
      logic[8:0] y2;
      logic[9:0] mx;//dx
      logic[8:0] my;//dy
      logic[2:0] line_col; //3 bit RGB
   } line_seg;

   //3 channel output variables
   logic R, G, B;

   //MAIN TERRAIN
   rect land = ’{0,639,400,479,3’b110};
   logic land_draw;

   //Middle obstacle
   rect barrier = ’{280,360,300,400,3’b110};
   logic barrier_draw;
```

```

    //TANKS
    rect tank1 = '{50,70,387,400,3'b010};
    logic tank1_draw;

    rect tank2 = '{569,589,387,400,3'b100};
    logic tank2_draw;

    //BARREL
    line_seg barrel = '{50,70,387,375,0,0,3'b111};
    logic barrel_draw;

    line_seg barrel2 = '{569,589,375,387,0,0,3'b111};
    logic barrel2_draw;

    //barrel position assignments
    assign barrel.y2 = barrel1y;
    assign barrel.x2 = barrel1x;

    assign barrel2.y1 = barrel2y;
    assign barrel2.x1 = barrel2x;

    //bullet on off drawing logic
    logic bullet = 0;
    logic bullet2 = 0;

    //calculate line slopes, check if divide by zero
    always_comb
    begin
       if(barrel.x2-barrel.x1)
          barrel.mx = barrel.x2-barrel.x1;
       barrel.my = barrel.y2-barrel.y1;

       if(barrel2.x2-barrel2.x1)
          barrel2.mx = barrel2.x2-barrel2.x1;
       barrel2.my = barrel2.y2-barrel2.y1;
    end

    //draw it all
    always_comb
    begin

       //conditions under which each object is drawn using defines above
       land_draw =    `BLOCK(land);
       barrel_draw =  `POINT(barrel.x1,barrel.y1,0) ||
                   `POINT(barrel.x2,barrel.y2,1) ||
                   `LINEPOS(barrel,1);

       barrel2_draw =    `POINT(barrel2.x2,barrel2.y2,0) ||
                   `POINT(barrel2.x1,barrel2.y1,1) ||
                   `LINENEG(barrel2,1);

       tank1_draw =   `BLOCK(tank1);
       tank2_draw =   `BLOCK(tank2);
       barrier_draw =    `BLOCK(barrier);
       bullet =       `POINT(p1bulletx,p2bullety,3);
       bullet2 =      `POINT(p2bulletx,p2bullety,3);


       //Output channels before checking for display area
       R = bullet || (land_draw & land.ob_col[2]) || (barrier_draw & barrier.ob_col[2]) ||
          (barrel2_draw & barrel2.line_col[2]) ||(barrel_draw & barrel.line_col[2]) ||
          (tank1_draw & tank1.ob_col[2]) || (tank2_draw & tank2.ob_col[2]);

       G = bullet || (land_draw & land.ob_col[1]) || (barrier_draw & barrier.ob_col[1]) ||
          (barrel2_draw & barrel2.line_col[2]) ||(barrel_draw & barrel.line_col[1]) ||
          (tank1_draw & tank1.ob_col[1]) || (tank2_draw & tank2.ob_col[1]);

       B = bullet || (land_draw & land.ob_col[0]) || (barrier_draw & barrier.ob_col[0]) ||
          (barrel2_draw & barrel2.line_col[2]) ||(barrel_draw & barrel.line_col[0]) ||
          (tank1_draw & tank1.ob_col[0]) || (tank2_draw & tank2.ob_col[0]);

    end

    //final output variables
    logic vga_R, vga_G, vga_B;

    always @(posedge clk)
    begin
      vga_R <= R & inDisplayArea;
```

```
      vga_G <= G & inDisplayArea;
      vga_B <= B & inDisplayArea;
    end


endmodule
```

6.9 hvsync_generator.sv

```
module hvsync_generator(CLOCK_50,
                    vga_h_sync,
                    vga_v_sync,
                    inDisplayArea,
                    CounterX,
                    CounterY);

   input CLOCK_50;
   output vga_h_sync, vga_v_sync;
   output inDisplayArea;
   output [9:0] CounterX;
   output [8:0] CounterY;

   //Counter ranges sized according to screen size
   /////////////////////
   logic [9:0] CounterX;
   logic [8:0] CounterY;
   logic CounterXmaxed;

   logic clk;
   logic clk2;
   //25.175 MHz clock source
   clock25 clockgen(CLOCK_50, clk);

   //counter x maxed flag variable
   always_comb
   begin
      CounterXmaxed = (CounterX==813);
   end


   always @(posedge clk)
   if(CounterXmaxed)
      CounterX <= 0;//reset counter at max
   else
      CounterX <= CounterX + 1;//otherwise increment counter

   always @(posedge clk) begin
      //This variable is allowed to overflow instead of resetting
      if(CounterXmaxed) CounterY <= CounterY + 1;
   end


   logic vga_HS, vga_VS;
   always @(posedge clk)
   begin
   // change this value to move the display horizontally
      vga_HS <= (CounterX[9:4]==47);
      // change this value to move the display vertically
      vga_VS <= (CounterY==500);
   end

   logic inDisplayArea;
   always @(posedge clk)
   if(inDisplayArea==0)
      inDisplayArea <= (CounterXmaxed) && (CounterY<480);
   else
      inDisplayArea <= !(CounterX==639);

   assign vga_h_sync = ~vga_HS;
   assign vga_v_sync = ~vga_VS;

endmodule
```