Slave GPU for Microcontrollers

Osbaldo Vera

R11364327

Texas Tech University

ECE 4333-302

Brian Nutter, Ph.D., P.E.

December 3, 2019

Abstract

This paper describes a slave GPU system designed for microcontrollers. The design focuses on a sprite controller implementation on a Artix-7 FPGA board, the Basys 3. This paper goes over the memory requirements for the frame buffers, sprite memory and background memory. Also covered is a section on the 340 x 240 p VGA controller and the drawing module as well as user controls.

Table of Contents

List of Figures

1. Introduction

The Internet of things is the addition of connecting and communicating between two controlled systems. It is incredibly common to see smart phones communicate with objects that are connected to the same Wi-Fi. Most of these objects, like household items, have a very limited graphical user interface, partly due to the higher cost of micro-controllers with built in GPU's. This forces the developers to use much needed processing power to display information. Ultimately, the goal of this project is to provide a simple slave GPU to be used by these microcontrollers to display information on a 320x200 p screen. For demonstration purposes this particular device will load up a simple game of Tanks.

Given the nature of this project, there is very little hardware. One Microcontroller (MSP430G2553) with two 2-axis joysticks with built in buttons, one display monitor, and lastly the FPGA Board (Artix-7). The FPGA will house all the necessary components of the GPU, including the VGA controller, Memory, and the Drawing module as shown in Figure 1.

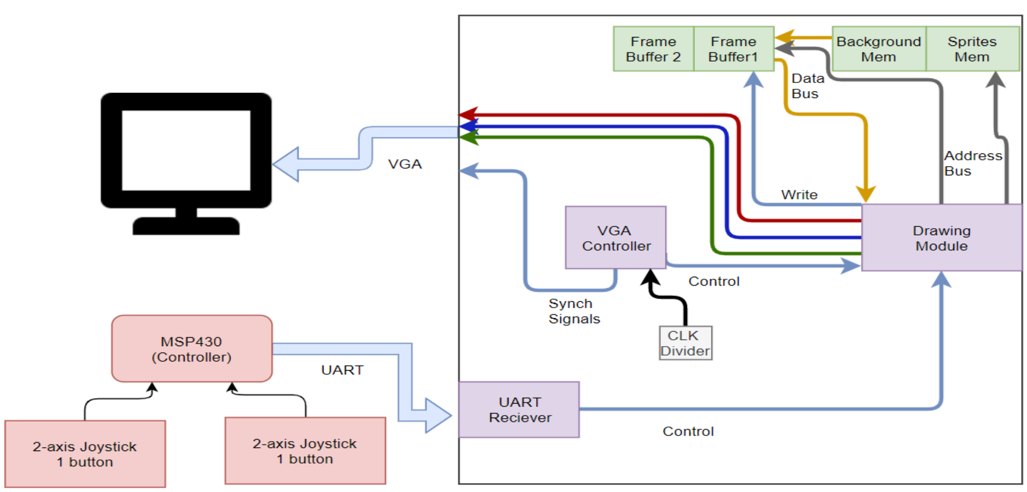


Figure 1: Full System Diagram

2.VGA Controller

The Video Graphics Array (VGA) interface uses a 15 pin D connector for communication, as shown in Figure 2. Also shown are the pins that will be isolated for this project, analog signals Red, Green, and Blue, along with active low signals Horizontal Synchronization (HS) and Vertical Synchronization (VS). The Artix-7 board includes the necessary 4-bit digital to analog converters on each analog signal. As shown in the figure below, it uses 4 bits in a voltage divider configuration to accomplish this. This gives one the flexibility to use 4-bits for each color for a total of 12-bits for each pixel.

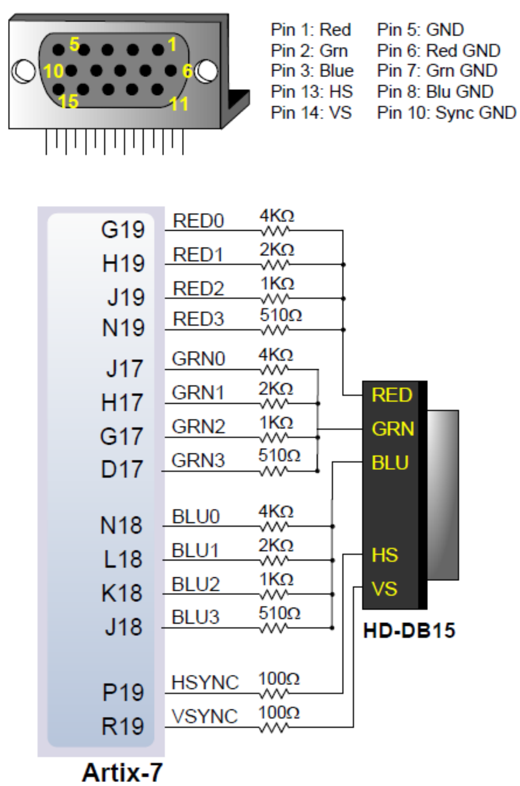


Figure 2: VGA Connector and Artix -7 Output

Using the 5 signals mentioned above, one is able to display information onto a screen given that the timing is correct. Each resolution will have its specified timing between the horizontal rows and when the screen resets. Unfortunately, displays in this current age very rarely support 320x200 p display. The most common resolution is twice the size, 640x480 p, using the timing from this resolution to create a 320x200 p VGA Controller. According to VESA standards, a 640x480 p operating at 60 Hz will have a pixel frequency of 25 MHz, this is the time it takes to change from one pixel to the next on the physical screen. In most screens this will go from left to right until it reaches the end of the screen on the first row. After which one must give it time to reset onto the next row, in total this equates to 160 pixel cycles. This section is split up into 3 regions, the front porch, horizontal synchronization and the back porch. Similarly, at the end of the screen, it must reset itself all the way to the top, which takes a total of 44-pixel cycles.[1] A two-dimensional Timing block used to fully understand the VGA is shown in Figure below.

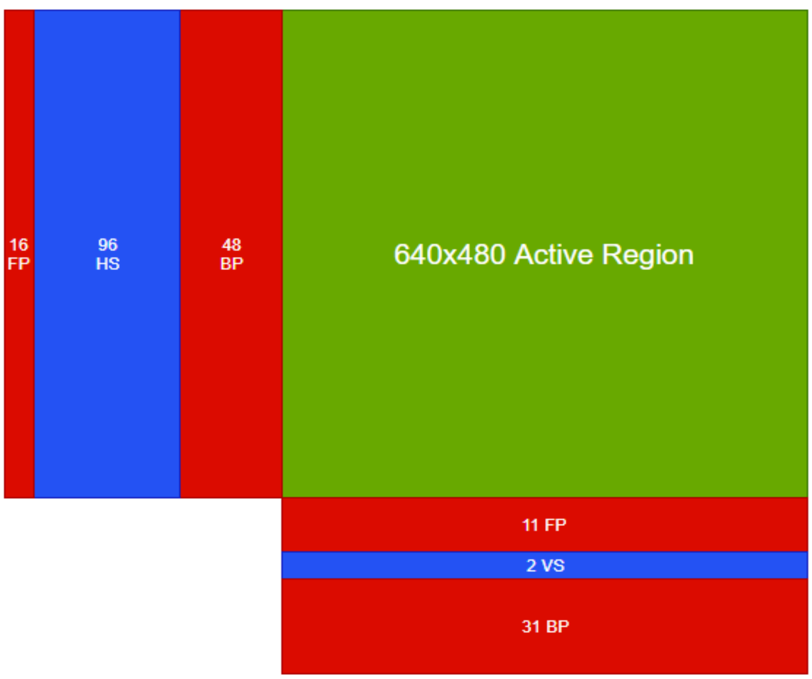


Figure 3: VESA 680x480 p VGA Timing

Given the basic understanding of VGA timings, the implementation of the VGA controller in Verilog is shown in Figure 4. Two inputs, one being the 25 MHz clock and a reset signal. The Clock will drive two counters, one counting horizontally, which counts from 0 – 800, 16p front porch, 96 p horizontal synch, 48 p back porch and 640 p active. The other counter will count vertically, which only occurs when the horizontal clock reaches 800, this counter will count from 0 – 524, 480 p active, 11 p front porch 2 p vertical sync and 31 p back porch. Refer Figure 3 for the timing given.

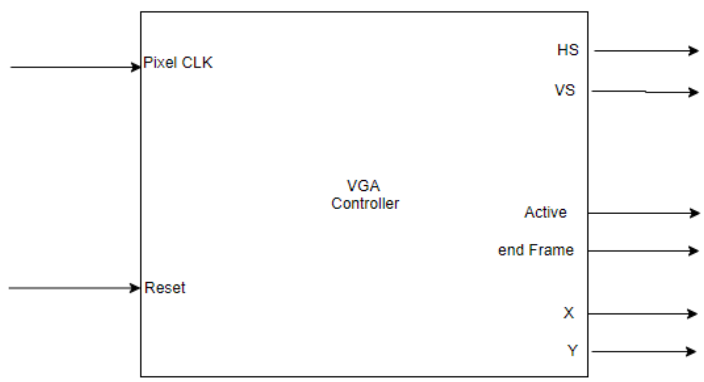


Figure 4: VGA Controller Block Diagram

There are two outputs that directly connect to the VGA connector, Horizontal Synch and Vertical Synch which are active low whenever it is in the period shown Figure 3. The last four output signals are to be used by the drawing module, where X and Y give the position inside the Active region. These values must be divided by 2 or shifted right once this will ensure that each pixel is counted twice converting our 640x480 p display to 320x240 p. Then one must add 40 pixels on vertical active region's lower bound and subtract 40 pixels from the upper bound, this will turn the active region from 0-480 into

40 - 440, or 0 - 400. Ensuring that the controller will be configured as a 320x200 p resolution by shaving off 20 pixels from each end of the vertical screen. The active signal will notify the drawing module that it is inside the active region, and the end frame will tell it when the last pixel has been planted. [1] An example of the code used to generate these signals is shown in Figure 5.

```
//Active and screenend signals
assign o_active = ~((h_count < H_ACT_ST) | (v_count > V_ACT_EN - 40 - 1) //  output ac
                                         | (v_count < 40));      // high when in acti
assign o_endframe = ((v_count == SCREEN - 1) & (h_count == LINE));

//Making sure X and Y output positions stay within active regions depending on timing
assign o_xpos = (h_count < H_ACT_ST) ? 0 : (h_count - H_ACT_ST) >> 1;
assign o_ypos = (v_count >= V_ACT_EN) ? (V_ACT_EN - 40 - 1) : (v_count -40) >> 1;
```

Figure 5: VGA signal Example

3.Memory

In order to display an image onto a screen the design must include a Frame Buffer, or a section in memory dedicated to displaying an image on the screen. Using the controller talked about previously, the resolution will be 320x200 p where in each given frame there will be 64,000 pixels. If one decided to use all 4 bits for each RGB color that puts the total memory requirement for 1 frame buffer at 768 Kbits. To effectively draw and display graphics, the method of double buffering will be used, in which 1 frame buffer will be used to display while the other will be used to draw. This eliminates any chance of stutters and tearing. Because of this and the fact that the Artix-7 memory budget is at 1800 Kbits, using 4bits per RGB is not a viable option. Limiting the number of colors one can represent by reducing the number of bits per RGB from 4 to 2 will give 6 bits per pixel, bringing the total memory requirement down to 384 Kbits. Allowing enough room for two frame buffers putting the total to 768 Kbits. The implementation of each frame buffer is shown in Figure

6, when write enable is high then memory will update that specified address, while when low it will output memory from that address.

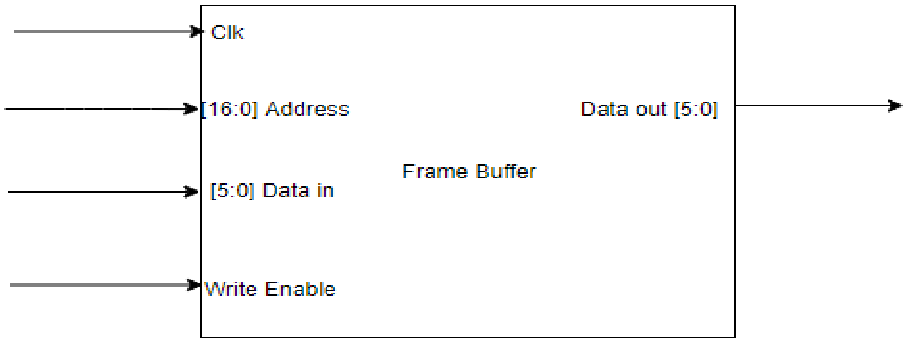


Figure 6: Memory Block Diagram

In addition to the two frame buffers there will be a read only memory allocated for a custom background and obstacle map as well as a 10 32x32 p sprites adding an addition 394.24 Kbits, this sets the total memory budget to 1162.24 Kbits. The implementation for the two read only memory slots copy Figure 6 without the write enable pin or data in pins. An excel spreadsheet is used to create the sprites and backgrounds figure RAWR shows the sprite sheet used.

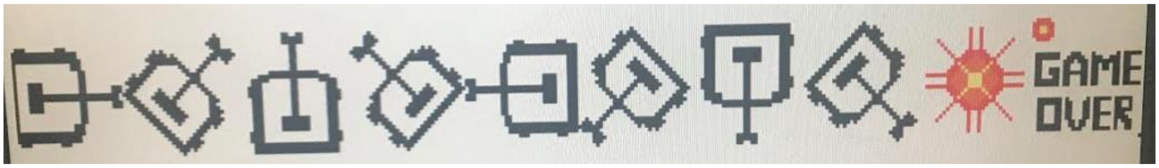


Figure 7: Sprite Sheet

4.Drawing Module

The Drawing module is composed of two blocks, one drawing and updating each frame on one of the frame buffers while the other small section outputs the RGB values onto the physical VGA connector. The output section updates a 6bit color register every pixel clock (25 MHz) as long as the active signal is high. Then the drawing module will use the X and Y output from the VGA controller to find the address of the respected pixel point. Which frame buffer the color register pulls from is decided on whichever's frame buffer's write pin is not enabled. So, if frame buffer 1's write pin is enabled then the output will be pulled from frame buffer 2 until the end frame pin is enabled at which point the write pins will be inverted. This successfully implements double buffering.

As for the drawing state machine as shown in Figure 8 on the next page has 7 total states aside from state machine reset pin. The states follow the order of objects to be drawn onto the active frame buffer. This order will be: 1. Background, 2. Player 1 bullets, 3. Player 2 bullets, 4. Player 1. After this it will remain in state 5 where once the end frame signal is high will update every object in game as well as applying the physics of these objects. If a player wins, the game over signal will be set high and updating of objects will no longer happen and continue onto the last state where a game over sign is drawn in the middle until the reset button is pressed resetting all status registers. These status register dictate the location of each object as well as the direction, the players have an extra status register which dictates the sprite to be used.

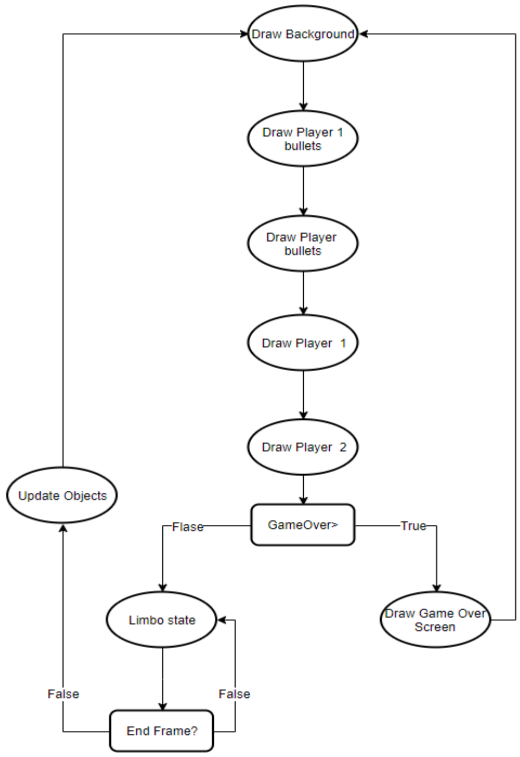


Figure 8: Drawing State Machine

The first step is to draw the background, this is done by incrementing an address register that inputs into the read only memory of the background memory. This address is copied to the active frame buffers address depending on which write pin is high. An address buffer is added between the frame buffer and the background memory since retrieving data is a two-cycle process. Taking advantage of the sweep through background memory we check 1 pixel in front of each corner of an object using their X and Y registers. If it is approaching an object in the background it saves the direction of the object to be used in the updating state.

The next state is drawing each players bullet, the bullet exists a 6x6 p sprite on the sprite sheet as the 9th indexed sprite. This must be extracted and used with each bullets X and Y register to successfully draw. This is done by having the draw X and draw Y counters count to 6, every time X reaches 6 increment Y until Y reaches 6 as well. These counters will be used increment through the address of the sprite by updating the address register according to this equation: 9*32 + drawX + (320*drawY). In order to find the address location in the frame buffer one must use this: 320+ drawX+ X + (320*(drawY+Y)). The address is then saved onto one address buffer because of the memory delay. This is repeated a total of 3 times per player, drawing them onto the active frame buffer

The last of the drawing states focuses on drawing the players, this is accomplished in the same manner as the bullets but because the players sprites are constantly changing a the sprite register for each player is put into the sprite address: Sprite#*32 + drawX + (320*drawY). Frame buffers address is calculated the same but with the players X and Y values. There is two address buffers for the frame buffer, on top of the memory delay we created a delay when changing the color of the sprite. Blue for player 1 and red for player 2. This is done by checking when the memory data output is equal to the black (000000) and converting the output to blue (000011) or red (110000). After that the output is put onto the active frame buffer. Refer to appendix C for a full code of Drawing player 2 State

The next state will always be a limbo state, where it stays until the end screen pin is high on the VGA controller. When high the all register begins to update depending on control register for each player, where each player has 5 bits, 4 bits for direction and 1 bit to shoot. A case statement is used on each players direction bits, depending on weather its going east, northeast, north, northwest, west, southwest, south, or southeast it will increment the X and Y registers of each player if it is within the bounds of the screen as well as no background object in front of it. The same is done with the bullets but it does not have a dynamic direction register but static register which put onto it when the last bit on the player control register is on. These X and Y registers for the bullets are always updated until it reaches a background object or frame border, when that occurs the status register of that specific bullet is cleared, removing the bullet from the screen on the next screen update. Refer to appendix D for code related to updating Player and bullet locations.

The last section checks to see if either end game event has taken place. To check weather both players have crash into each other one compares the four corners of player 2 using its X and Y registers against the bounds of players 1 using its X and Y registers. In the case that it does occur we change the sprite to the crash sprite and set GameOver bit High. A similar tactic is used for each bullet, if the bullet is active it will check weather or not each bullets 4 corners intercepts the opposites players bounds. Refer to Appendix E for the code related to end game scenarios.

If the GameOver bit is high then it will bypass the updating state and go directly to printing a Game Over screen to signify an end game scenario has occurred. Because of the mess above, code has been provided in appendixes.

5. Conclusion

Ultimately the project was not completely finished the User controls were not communicated by a microcontroller as a UART or SPI receiver were never implemented. Instead the ADC pins were used as well regular pins to control each tank. In all the GPU functioned correctly and shows how very possible it is to design and implement a small GPU for microcontrollers at a cost-effective rate. This design only utilized 12% of the basys 3’s LUTS and DFF slices as well as 78% of its total Memory.

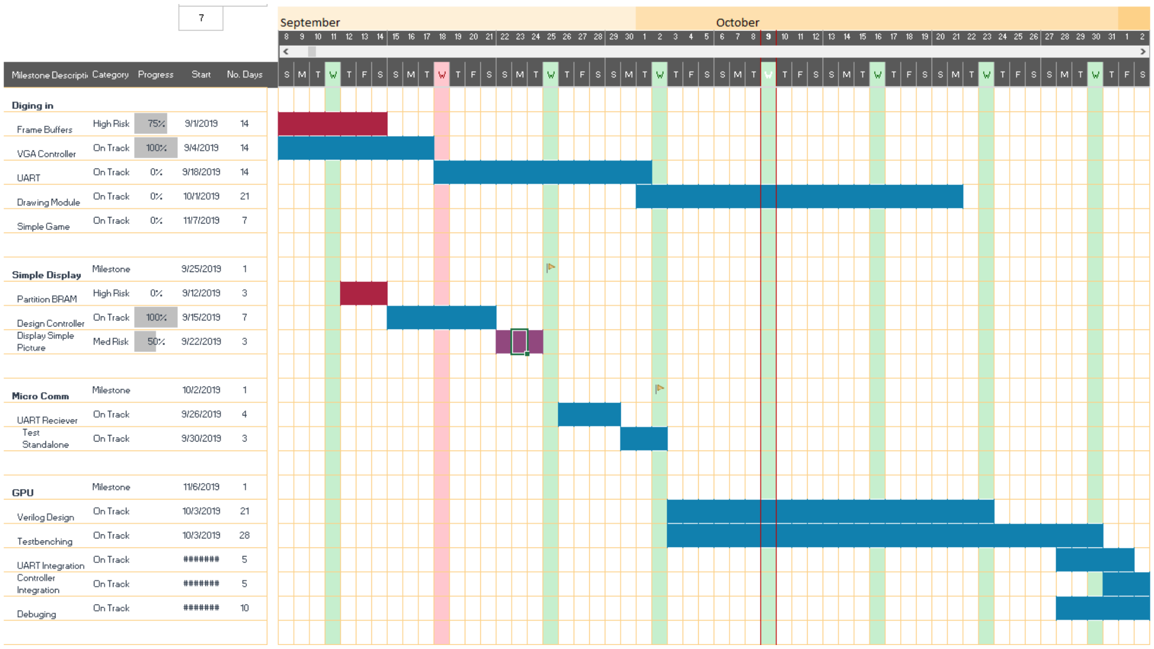
References:

1. “Basys 3_RM”, *Digilent*,[Online]

https://reference.digilentinc.com/_media/reference/programmable-logic/basys-3/basys3_rm.pdf?_ga=2.97874523.519887739.1571795256-2093494689.1567609175&_gac=1.124066168.1571795256.Cj0KCQjw0brtBRDOARIsANMDykaLyOOpDfbnAfzAP9gducMUGoKdDrWVrUnhMjxCyUIIav415cmsJ_YaAo5QEALw_wcB

Appendix A:

Gantt Chart



Appendix B:

Budget

Slave GPU	Running Total			Total Estimate		
Direct Labor:						
Category or individual:	*Rate/Hr*	*Hrs*		*Rate/Hr*	*Hrs*	
Oz	18	176	$3,168.00	18	200	$3,600.00
Total Labor:			$3,168.00			$3,600.00
Consulting Fees:						
Category or individual:	*Rate/Hr*	*Hrs*		*Rate/Hr*	*Hrs*	
Lab IV & V	35	0	$0.00	35	0	$0.00
Lab Tutors	40	0	$0.00	40	0	$0.00
Lab Assistants	40	0	$0.00	40	0	$0.00
Mr. Woodcock	100	0	$0.00	100	0	$0.00
Instructor	200	0	$0.00	200	10	$2,000.00
Total Contract Labor:			$0.00			$2,000.00
Supplies And Materials:			$0.00			$3,852.00
(from Materials Cost worksheet)						
Total Direct Material Cost:			$0.00			$3,852
Equipment Rental:	Value	Rental Rate (For Semester)		Value	Rental Rate	
Power Supply	$699.00	10.00%	$69.90	$699.00	10.00%	$69.90
Total Rental Costs:			$69.90			$69.90
Total Cost:		**Current**	$3,237.90		**Estimate**	$9,521.90

Appendix C:

Draw Player 2 Code

```
4:// Draw Player 2~~~~~~~~~~~~~~~~~
begin
     if(draw_y == 31)
    begin   //Execute only when dor
    //StateChange

        draw_y <= 0;
        draw_x <= 0;
        if(GameOver)
            state <= 5;
        else
            state <= 6;


    end


    //Iterrator of Draw registers
    if(draw_x == 31)
    begin
        draw_x <= 0;
        draw_y <= draw_y + 1;
    end
    else
    begin
        draw_x <= draw_x + 1;
    end

    //address used to find locatior
    addr_s <= pl2_sprite * 32 + (32
    //address of Location of play c
    address_fb1 <= 320 * (pl2_y + c
    address_fb2 <= address_fb1;// I
     address_fb3 <= address_fb2;

    //Change Color to blue (1st pla
    if(dataout_s == 0)
    begin
        data_buff <= 6'b110000;
    end
    else
    begin
        data_buff <= dataout_s;
    end

    //paste to frame Buffers
    if(write_1)
    begin
        addr_1 <= address_fb3;
        datain_1 <= data_buff;
    end
    else
    begin
        addr_2 <= address_fb3;
        datain_2 <= data_buff;
    end
```

Appendix D:

Updating Code: Player1

```
case(p1_stat[5:2])
    EAST:// E
    begin
        p1_sprite <= 0;// update sprit

        if(p1_x + 32 < 320
        && !(p1_bg_dir == EAST) ) // c
            p1_x <= p1_x + 1;

    end

    NORTHEAST:// NE
    begin
        p1_sprite <= 1;// update sprit

        if( p1_x + 32 < 320
        && !(p1_bg_dir == EAST) ) // c
            p1_x <= p1_x + 1;
        if( p1_y > 0
        && !(p1_bg_dir == NORTH) )
            p1_y <= p1_y - 1;

    end
    NORTH:// N
    begin
        p1_sprite <= 2;// update sprit

        if( p1_y > 0
        && !(p1_bg_dir == NORTH))  //c
            p1_y <= p1_y - 1;

```

Appendix D:

Updating Code: Player 2 bullet 2

```
    //bullet 2  pl2
            if(pl2_bl_stat[2][0])
           begin

           if( pl2_bl_x[2] + 8 >= 320
            || pl2_bl_y[2] + 8 >= 200
            || pl2_bl_x[2] <= 2
            || pl2_bl_y[2] <= 2 )
            begin
               pl2_bl_stat[2] <= 0;
            end

           case(pl2_bl_stat[2][4:1])

               EAST:// E
               begin
                   pl2_bl_x[2] <= pl2_bl_x[2] + 2;

                   if(pl2_bbg_dir[2] == EAST)
                       pl2_bl_stat[2] <= 0;

               end

               NORTHEAST:// NE
               begin
                   pl2_bl_x[2] <= pl2_bl_x[2] + 2;
                   pl2_bl_y[2] <= pl2_bl_y[2] - 2;

                   if(pl2_bbg_dir[2] == EAST)
                       pl2_bl_stat[2] <= 0;
                   if(pl2_bbg_dir[2] == NORTH)
                       pl2_bl_stat[2] <= 0;
               end
               NORTH:// N
```

Appendix E:

End Game Scenarios: Player Crashing

```
if( pl_x <= pl2_x && pl2_x <= pl_x + 32 // Left X bound
&&  pl_y <= pl2_y && pl2_y <= pl_y + 32  // Top Y Bound
)begin
GameOver <= 1;
pl_sprite <= 8;
pl2_sprite <= 8;
end
if( pl_x <= pl2_x && pl2_x <= pl_x + 32 // Left X bound
&&  pl_y <= pl2_y + 32 && pl2_y + 32 <= pl_y + 32  // B
)begin
GameOver <= 1;
pl_sprite <= 8;
pl2_sprite <= 8;
end

if( pl_x <= pl2_x + 32 && pl2_x + 32 <= pl_x + 32 // Ri
&&  pl_y <= pl2_y && pl2_y <= pl_y + 32  // Top Y Bound
)begin
GameOver <= 1;
pl_sprite <= 8;
pl2_sprite <= 8;
end

if( pl_x <= pl2_x + 32 && pl2_x + 32 <= pl_x + 32 //Rig
&&  pl_y <= pl2_y + 32 && pl2_y + 32 <= pl_y + 32 // Bo
)begin
GameOver <= 1;
pl_sprite <= 8;
pl2_sprite <= 8;
end
```

Appendix E:

End Game Scenarios: Player 1 bullet 1

```
if(p1_bl_stat[1][0])
begin
if( pl2_x <= p1_bl_x[1] && p1_bl_x[1] <= pl2_x + 32 // Left X bound
&&  pl2_y <= p1_bl_y[1] && p1_bl_y[1] <= pl2_y + 32  // Top Y Bound
)begin
GameOver <= 1;
pl2_sprite <= 8;
p1_bl_stat[1][0] <= 0;
end

if( pl2_x <= p1_bl_x[1] && p1_bl_x[1] <= pl2_x + 32 // Left X bound
&&  pl2_y <= p1_bl_y[1] + 6 && p1_bl_y[1] + 6 <= pl2_y + 32  // Bottom Y Bound
)begin
GameOver <= 1;
pl2_sprite <= 8;
p1_bl_stat[1][0] <= 0;
end

if( pl2_x <= p1_bl_x[1] + 6 && p1_bl_x[1] + 6 <= pl2_x + 32 // Right X bound
&&  pl2_y <= p1_bl_y[1] && p1_bl_y[1] <= pl2_y + 32  // Top Y Bound
)begin
GameOver <= 1;
pl2_sprite <= 8;
p1_bl_stat[1][0] <= 0;
end

if( pl2_x <= p1_bl_x[1] + 6 && p1_bl_x[1] + 6 <= pl2_x + 32 //Right X Bound
&&  pl2_y <= p1_bl_y[1] + 6 && p1_bl_y[1] + 6 <= pl2_y + 32 // Bottom Y Bound
)begin
GameOver <= 1;
pl2_sprite <= 8;
p1_bl_stat[1][0] <= 0;
end
end
```