ECE241 Final Project
‘Astro Party’

Nicholas Tran 1005305251
Kyle Blackie 1004831113

December 2, 2019
ECE241
University of Toronto

Introduction

For the final project in our Digital Systems course, we designed a circuit implementing a two player game where players control spaceships and try to hit each other with projectiles. Players controlled the movement and shooting of their respective ship and the navigation of menu screens using the keyboard, and each players' score was displayed on the HEX displays. The first one to 5 eliminations won.

The motivation of this project was to showcase everything we learned in computing, storing, and controlling digital circuits in this course. This manifested itself in creating a two player, arcade style game that used these aspects of hardware design in a way that was both challenging and rewarding to complete.

The Design

Block Diagram illustrating the parts of the Project:

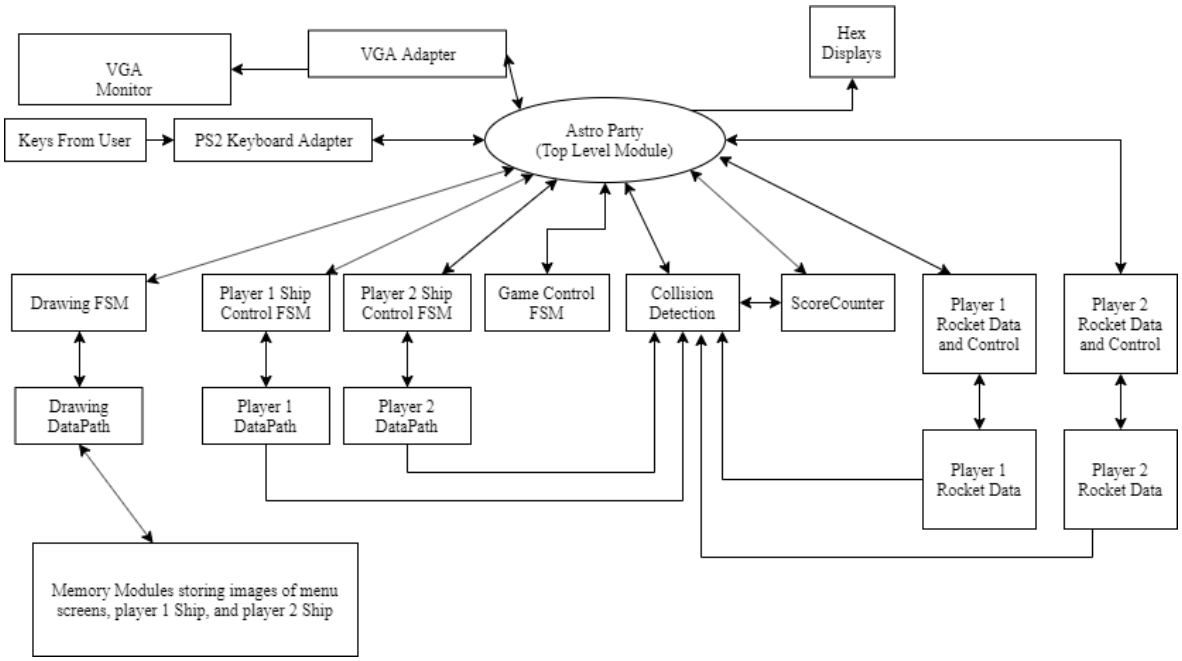


Figure 1. Design Block Diagram

Game Control Finite State Machine (FSM)

The Game Control FSM controls what state the game is in. When the project is loaded, the FSM waits at the menu state until ESC is pressed on the keyboard, moving into the play state, where it stays until a winner is found. Once a winner is found, it moves to the newgame state, where it waits until ENTER is pressed before moving back to the menu state, allowing the game to be played an infinite amount of times. This FSM communicates with the drawing FSM to indicate what menu screen to draw and when the Draw FSM can advance to its next states.

Drawing FSM

The Drawing FSM controls what object is being drawn to the VGA adapter at a certain time. Initially, this FSM loads the menu screen into the VGA adapter. Once the Game Control FSM is in its play state, the

Drawing FSM loops through the steps of drawing, erasing, and updating objects by sending the relevant enable signals to the Drawing, Ships, and Rocket Datapaths. The basic algorithm logic while the game control FSM is in the play state is as follows:

- Draw ships for Player 1, then Player 2 (Each ship is 19x19 pixels, or 289 pixels)
- Draw Rockets for Player 1, then Player 2 (Each rocket is 3x3 pixels, or 9 pixels)
- Delay so updates only happen at 60Hz
- Erase Player 1 ship, then erase Player 2 ship
- Erase Player 1 Rocket, then Player 2 Rocket
- Update Registers
- Go back to step 1

This loops until a winner has been detected, where the Drawing FSM sends a signal to the Display Datapath to draw the correct winner screen to the VGA adapter.

Drawing Datapath

The Drawing Datapath holds all the data to draw objects to the VGA adapter depending on what enable signal is being received. It receives enable signals from the Drawing FSM, and receives relevant XY and orientation data from the Ship and Rocket datapaths. It also contains counters that keep track of the number of pixels it has plotted during a certain Draw FSM state. This module retrieves the XY position of the current object it is drawing, and the colour Data from a memory module, and maps out to the VGA adapter a “frame” of the game before it is outputted to the screen.

Player 1 and Player 2 Ship Control FSM

The Player 1 Ship and Player 2 Ship Control FSMs are used to update the orientation of each respective ship. Data from the keyboard is sent through the top level module, where the Ship Control FSMs check the data for a relevant keypress (W for player 1, P for player 2). If one of those keypresses are detected, the FSM sends an enable signal to the respective Ship Datapath, updating the orientation of the ship.

Player 1 and Player 2 Ship DataPath

Each ship datapath contains three registers. Two of them keep track of the XY position of the top left pixel of the ship, and the third holds the orientation of the ship. A direction of 00 corresponds to north, 01 to East, 10 to South, and 11 to West. When an update signal is received from the Drawing FSM, the datapath increments or decrements the xy position registers depending on the ships orientation, and whether it is against a wall. When a rotate signal is received from the Ship’s Control FSM, the datapath “rotates” the ship 90 degrees by adding 1 bit to its orientation register.

Player 1 and Player 2 Rocket Control FSM

The rocket control FSM is similar to the Ship Control FSM, where it checks the keyboard data for a certain key press (Q for player 1, and O for player 2), and if it receives a relevant key press, it sends a fire enable signal to the Rocket Datapath of that ship, signalling that the player wants to fire a rocket.

Player 1 and Player 2 Rocket Datapath

The rocket module keeps track of the X and Y position of a rocket and its orientation and whether there is a rocket currently on the field from that ship.When a shoot enable signal is received from the Rocket

Control module, it checks if there is currently a projectile on the field from its ship. If there is, it will not fire another shot. However, if there is not a projectile already on the field, it will update the state register and output a starting X and Y position for the projectile based on the X and Y position of its ship.

Collision Detection

The collision detection module receives all XY position data from both Ship Data modules and both Rocket modules. If a ship and a projectile, or ship and ship collide, their respective hitboxes overlap, signalling a collision. If a player was hit by a projectile, a P1Hit or P2Hit signal is sent to the Drawing FSM, starting a new round, and to the ScoreCounter module, where it increments the score of the player who shot the other. If two ships collide, a new round starts, but no score is updated.

ScoreCounter

This module holds a counter for each player's score. Each time a player is hit by a projectile, it increments the counter of the player who shot the other player by two points, due to the hit enable signal being high for two clock ticks. Once a player reaches 10 points (5 eliminations), the module sends a signal indicating which player has won to the Drawing FSM, outputting a win screen to the VGA output.

Memory Modules

The memory modules initialize on chip memory using certain images. Each ship uses one module per orientation, initialized with a 19x19 pixel picture converted to a Memory Initialization File. The menu and win screens are also stored with memory modules, each holding a 160x120 pixel image - a quadrant of the background. The memory modules output a 3 bit colour output given a certain memory address.

Vga and PS2 Adapter

Both the VGA and PS2 adapter code was taken from IP cores provided to us during the course [1], [2]. The VGA adapter takes X ,Y and 3 bit colour information from the Drawing Datapath, and plots a pixel with that colour at that position on the screen when a writeEn signal is received from the Drawing FSM.

The PS2 adapter takes keystrokes from the keyboard and outputs an 8 bit signal referring to the last data received from the PS2 bus, and a new data signal, which is driven high for one clock cycle whenever new data is received. This data is used by the Ship Control, and the Game Control FSM modules to update various aspects of the game.

Report on Success

The following outlines our weekly progress:

Week 1:
After working on understanding the logic behind drawing and erasing images to the VGA display we managed to display the ships stored in mif files and eventually make them move across the screen using the keys on the FPGA board.

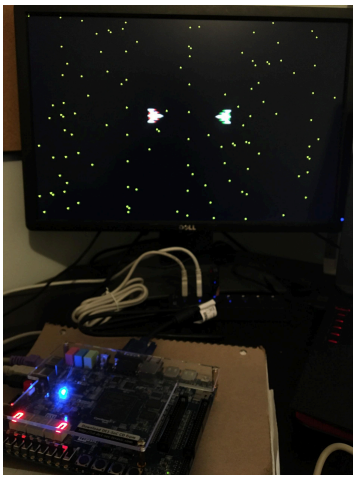
Week 2:
The goal of this week was to implement a shooting mechanic for the ships. This was implemented similarly to how the ships were drawn in week 1. The projectiles were instantiated in their own registers with positions dependent on the ship and would then travel across the screen by being drawn and erased like a ship would.

We also set a milestone to implement collision detection which was successfully met. The game would now detect when a player's projectile hit the opposing player's ship and would score accordingly. We then decided to add a new game mechanic where if the ships collided it would end the round as a draw. Bounds were also added that prevented the ships from flying off screen.

Week 3:
For the final milestone the goal was to add more polish to the game. This was done by adding in different game states (Menu, Play, Gameover) that were managed by a Finite State Machine and implementing keyboard controls which gave the final game feel more complete when playing.

Parts that didn't work:
While the project met all of the milestones there was still some issues with it. The main problem came from some tearing that would occur on the ships. This was most likely due to reading and writing from the VGA simultaneously and at high frequencies.

Overall, the project was a success. All of the initial goals were met on time and as a result the final product was a complete two player space shooter game.



What Would you do Differently

Given the chance to start this project over again there are several changes that would be made to better the project.

1. Organise the Verilog modules better
Throughout development, the code was continuously getting updated as new features were added into the project. However, when this code was added it was often done by stacking extra states and registers throughout the existing code with less thought towards structure of the overall project. This caused the code to lack some clarity which made the debugging process more difficult than it should have been.

2. Allow for multiple projectiles
In its current state, the game only allows for one projectile to be on the screen at a time per ship as this was easier to program at the time. Given the chance to update this, we would try to allow for multiple projectiles to be fired from each ship which would require multiple registers for each of the projectiles as well as a frequency counter to control the rate of fire.

3. Add music and sound effects
The project as it is plays well however, adding in sound effects and background music would give the project more of a finished feel. In order to achieve this we would have needed to schedule more time towards working on getting audio to work as it proved more complex than we had initially thought.

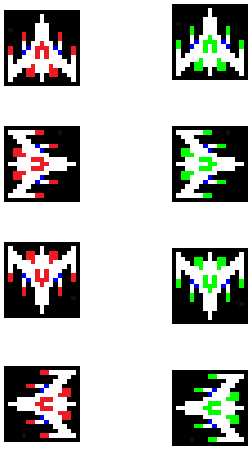
References

[1] S. Vafaee, "Downloads," *VGA Adapter*. [Online]. Available: http://www.eecg.toronto.edu/~jayar/ece241_08F/vga/vga-download.html. [Accessed: 01-Dec-2019].

[2] "PS/2 Controller," *PS/2 Controller*. [Online]. Available: http://www.eecg.toronto.edu/~pc/courses/241/DE1_SoC_cores/ps2/ps2.html. [Accessed: 01-Dec-2019].

Appendix:

Images Used:



Code:

```
module AstroParty(
		CLOCK_50,				//	On Board 50 MHz
		// Your inputs and outputs here
		KEY, SW, HEX0, HEX5,			//	On Board Inputs
		// The ports below are for the VGA output.  Do not change.
		VGA_CLK,				//	VGA Clock
		VGA_HS,					//	VGA H_SYNC
		VGA_VS,					//	VGA V_SYNC
		VGA_BLANK_N,				//	VGA BLANK
		VGA_SYNC_N,				//	VGA SYNC
		VGA_R,					//	VGA Red[9:0]
		VGA_G,					//	VGA Green[9:0]
		VGA_B,					//	VGA Blue[9:0]

		PS2_CLK,				//	PS2 Clock
		PS2_DAT					//	PS2 Data
	);

	input	CLOCK_50;				//	50 MHz
	input	[3:0] KEY;				//	FPGA Keys
	input	[9:0] SW;				//	FPGA Switches

	//Bidirectionals
	inout	PS2_CLK;
	inout	PS2_DAT;

	//Outputs
	output [6:0] HEX0, HEX5;
	output	VGA_CLK;				//	VGA Clock
	output	VGA_HS;					//	VGA H_SYNC
	output	VGA_VS;					//	VGA V_SYNC
	output	VGA_BLANK_N;				//	VGA BLANK
	output	VGA_SYNC_N;				//	VGA SYNC
	output	[7:0]	VGA_R;				//	VGA Red[7:0]
	output	[7:0]	VGA_G;				//	VGA Green[7:0]
	output	[7:0]	VGA_B;				//	VGA Blue[7:0]

	wire resetn;
	assign resetn = SW[0];

	//VGA Related Wires
	wire [2:0] colour;
	wire [8:0] x;
	wire [7:0] y;
	wire writeEn;

	//PS2 Related Wires
	wire [7:0] keyData;
	wire keyPressed;
```

```
//Ship Data and Control
wire [8:0] x1_pos, x2_pos;                          // x position of both ships
wire [7:0] y1_pos, y2_pos;                          // y position of both ships
wire [1:0] p1Orientation, p2Orientation;            // Orientation, 00 = North, 01 = East, 10= South, 11= West
wire p1RotateEnable, p2RotateEnable;                // Enable signal to rotate ship
wire fireshot1, fireshot2;                          // Enable signal to fire shot

//Rocket Data and Control
wire [8:0] x_rkt1, x_rkt2;                          // x position of both rockets
wire [7:0]  y_rkt1, y_rkt2;                         // y position of both rockets
wire [1:0] shot_1Orientation, shot_2Orientation;    // shot orientation, same as ship orientation (00 = North...)
wire shotInAir1, shotInAir2;                        // State if shot is in air

//Drawing Control and Datapath Wires
//Counters
wire [8:0] dotCounter;                              // Counter to draw and erase ships and rockets
wire [17:0] blackCounter;                           // Counter for background images
wire [24:0] freqCounter;                            // Delay so update only happens at 60Hz
//Enable Signals
wire update, init, erase, ld_p1, ld_p2, ld_rkt1, ld_rkt2;       // Main enable signals for drawing
wire ld_p1win, ld_p2win, ld_menu;                               // Menu Screens
//Reset Signals
wire countReset;                                                //Resets counters
wire gameStart;                                                 //Enable signal to start game
wire newRound;                                                  //Enable signal for new round

wire [3:0] p1Score, p2Score;                        //Count score for respective ship
wire p1Win, p2Win;                                  // Signal indicated p1 or p2 has won
wire shipsCollide;                                  //Signal for if ships collide with each other

//Game Control FSM Wires
wire playGame, menuScreen, newGame;

//Module Instantiation
vga_adapter VGA(
                        .resetn(resetn),
                        .clock(CLOCK_50),
                        .colour(colour),
                        .x(x),
                        .y(y),
                        .plot(writeEn),
                        .VGA_R(VGA_R),
                        .VGA_G(VGA_G),
                        .VGA_B(VGA_B),
                        .VGA_HS(VGA_HS),
                        .VGA_VS(VGA_VS),
                        .VGA_BLANK(VGA_BLANK_N),
                        .VGA_SYNC(VGA_SYNC_N),
                        .VGA_CLK(VGA_CLK));
```

```
        defparam VGA.RESOLUTION = "320x240";
        defparam VGA.MONOCHROME = "FALSE";
        defparam VGA.BITS_PER_COLOUR_CHANNEL = 1;
        defparam VGA.BACKGROUND_IMAGE = "/Images/black.mif";

PS2_Controller ps2(                                 //Keyboard input output module
        .CLOCK_50(CLOCK_50),
         .reset(~resetn),
         .PS2_CLK(PS2_CLK),
         .PS2_DAT(PS2_DAT),
         .received_data(keyData),
         .received_data_en(keyPressed));

gameControl gameFSM (                               //Game State FSM
        .clock(CLOCK_50),
         .reset(resetn),
         .keyCode(keyData),
         .newCode(keyPressed),
         .p1Win(p1Win),
         .p2Win(p2Win),
         .gamePlay(playGame),
         .menuScreen(menuScreen),
         .newGame(newGame));

gamedelay delay3(                                   //Delays movement of ships when new round starts by 1s
        .clock(CLOCK_50),
        .reset(resetn),
        .countReset(countReset),
        .newGame(newGame),
        .enable(gameStart),
        .init(init),
        .newRound(newRound));

control1 Ship1Ctrl(                                 //Ship 1 Control FSM
        .clock(CLOCK_50),
         .reset(resetn),
         .keyCode(keyData),
         .newCode(keyPressed),
        .countenable(p1RotateEnable));

p1Data ship1Data(                                   // Ship 1 Datapath
        .clock(CLOCK_50),
        .reset(resetn),
        .rotate(p1RotateEnable),
         .update(update),
        .init(init),
        .x(x1_pos),
        .y(y1_pos),
        .p1Orientation(p1Orientation));
```

```
control2 Ship2Ctrl(                                    //Ship 2 Control FSM
        .clock(CLOCK_50),
        .reset(resetn),
         .keyCode(keyData),
        .newCode(keyPressed),
        .countenable(p2RotateEnable));

p2Data ship2Data(                                      // Ship 2 Datapath
        .clock(CLOCK_50),
         .reset(resetn),
        .rotate(p2RotateEnable),
        .update(update),
        .init(init),
        .x(x2_pos),
        .y(y2_pos),
         .p2Orientation(p2Orientation));

shotControl1 ShotShip1(                                //Ship 1 Rocket FSM
        .clock(CLOCK_50),
        .reset(resetn),
         .keyCode(keyData),
        .newCode(keyPressed),
        .shotEnable(fireshot1));

rocket rkt1(                                           //Ship 1 Rocket Datapath
        .clock(CLOCK_50),
         .reset(resetn),
        .shotInAir(shotInAir1),
        .update(update),
        .init(init),
        .fireShot(fireshot1),
        .orientation(p1Orientation),
         .ship_x(x1_pos),
        .ship_y(y1_pos),
        .x_shot(x_rkt1),
        .y_shot(y_rkt1),
        .shot_orientation(shot_1Orientation),
        .x_rocket(x_rkt2),                 //Opponents rocket x
        .y_rocket(y_rkt2));                //Opponents rocket y


shotControl2 ShotShip2(                                //Ship 2 Rocket FSM
        .clock(CLOCK_50),
        .reset(resetn),
        .keyCode(keyData),
        .newCode(keyPressed),
        .shotEnable(fireshot2));

rocket rkt2(
        .clock(CLOCK_50),
```

```
        .reset(resetn),
        .shotInAir(shotInAir2),
        .update(update),
        .init(init),
        .fireShot(fireshot2),
        .orientation(p2Orientation),
        .ship_x(x2_pos),
        .ship_y(y2_pos),
        .x_shot(x_rkt2),
        .y_shot(y_rkt2),
        .shot_orientation(shot_2Orientation),
         .x_rocket(x_rkt1),                    //Opponent rocket x
        .y_rocket(y_rkt1));                    //Opponent rocket y


collision collisionDetect (                    //Collision Tracker
        .clock(CLOCK_50),
        .reset(resetn),
        .init(init),
        .x1_pos(x1_pos),
        .x2_pos(x2_pos),
        .y1_pos(y1_pos),
        .y2_pos(y2_pos),
        .x1shot(x_rkt1),
        .x2shot(x_rkt2),
        .y1shot(y_rkt1),
        .y2shot(y_rkt2),
        .p1Hit(p1Hit),
        .p2Hit(p2Hit),
        .shipsCollide(shipsCollide),
        .shotInAir1(shotInAir1),
        .shotInAir2(shotInAir2));

scoreCounter scores(                           //Keeps track of scores
        .clock(CLOCK_50),
        .reset(resetn),
        .newGame(newGame),
        .p1Hit(p1Hit),
        .p2Hit(p2Hit),
        .scoreP1(p1Score),
        .scoreP2(p2Score),
        .p1Win(p1Win),
        .p2Win(p2Win));

displayCtrl disp1(                             //Display FSM
        .clock(CLOCK_50),
        .reset(resetn),
        .shotInAir1(shotInAir1),
        .shotInAir2(shotInAir2),
        .dotCounter(dotCounter),
```

```
        .blackCounter(blackCounter),
        .freqCounter(freqCounter),
        .p1Hit(p1Hit),
        .p2Hit(p2Hit),
         .newGame(newGame),
        .shipsCollide(shipsCollide),
        .playGame(playGame),
        .menuScreen(menuScreen),
        .p1Win(p1Win),
        .p2Win(p2Win),
        .gameStart(gameStart),
        .ld_plot(writeEn),
        .erase(erase),
        .update(update),
        .newRound(newRound),
        .init(init),
        .ld_p1(ld_p1),
         .ld_p2(ld_p2),
        .ld_rkt1(ld_rkt1),
        .ld_rkt2(ld_rkt2),
        .countReset(countReset),
        .ld_menu(ld_menu),
        .ld_p1win(ld_p1win),
        .ld_p2win(ld_p2win), );

displayData data1(
        .clock(CLOCK_50),
        .reset(resetn),
        .erase(erase),
        .init(init),
        .ld_plot(writeEn),
        .countReset(countReset),
        .x_pos1(x1_pos),
        .ld_rkt1(ld_rkt1),
        .ld_rkt2(ld_rkt2),
        .ld_p1(ld_p1),
        .ld_p2(ld_p2),
        .ld_menu(ld_menu),
        .ld_p1win(ld_p1win),
         .ld_p2win(ld_p2win),
        .y_pos1(y1_pos),
        .x_pos2(x2_pos),
        .y_pos2(y2_pos),
        .p1Orientation(p1Orientation),
        .p2Orientation(p2Orientation),
        .x_rkt1(x_rkt1),
        .x_rkt2(x_rkt2),
        .y_rkt1(y_rkt1),
        .y_rkt2(y_rkt2),
        .rkt1Orientation(shot_1Orientation),
```

```
                .rkt2Orientation(shot_2Orientation),
                .x_vga(x),
                .y_vga(y),
                .colour_out(colour),
                .dotCounter(dotCounter),
                .freqCounter(freqCounter),
                .blackCounter(blackCounter));

        seg7 u2 (.i(p2Score), .hex(HEX0));
        seg7 u1 (.i(p1Score), .hex(HEX5));
endmodule


//Game Control FSM
module gameControl(
        input clock, reset,
        input [7:0] keyCode,
        input newCode,
        input p1Win, p2Win,
        output reg gamePlay, menuScreen, newGame);

        reg [3:0] current_state, next_state;

        localparam      S_MENU          = 4'd0,
                        S_PLAY          = 4'd1,
                        S_NEWGAME       = 4'd2;

        //State Table
        always@(*)
        begin
        case(current_state)
                //Loops until ESC is pressed
                S_MENU: next_state = (newCode && keyCode == 8'h76)? S_PLAY: S_MENU;
                //Loops until a winner is found
                S_PLAY: next_state = (p1Win || p2Win)? S_NEWGAME: S_PLAY;
                //Loops until Enter is pressed, then return to menu state
                S_NEWGAME: next_state = (newCode && keyCode == 8'h5A)? S_MENU: S_NEWGAME;
                default: next_state = S_MENU;
        endcase
        end

        //State Registers
        always@(*)
        begin
        menuScreen      = 1'b0;
        gamePlay        = 1'b0;
        newGame         = 1'b0;

        case(current_state)
                S_MENU: begin
```

```
                        menuScreen = 1'b1;
                        end
                S_PLAY: begin
                        gamePlay= 1'b1;
                        end
                S_NEWGAME: begin
                        newGame = 1'b1;
                        end
        endcase
        end

        //State transitions
        always@(posedge clock)
        begin
        if(!reset)
                current_state <= S_MENU;
        else
                current_state <= next_state;
        end
endmodule


//Game Delay Module
module gamedelay(
        input clock,
        input reset,
        input countReset, newRound, init,
        input newGame,
        output reg enable);

        reg [30:0] counter;

        always@(posedge clock)
        begin
        if (!reset || newGame || newRound)              //Reset the counters
                begin
                counter <= 31'd0;
                enable <= 0;
                end
        else
                begin
                if (counter == 31'd49999999)            //If counted for one second, send enable
                        begin
                        enable <= 1;
                        counter <= 31'b0;
                        end
                else                                                    //Count up
                        begin
                        counter <= counter + 1;
                        enable <= 0;
```

```
                end
            end
        end
endmodule
```

//Ship 1 Control FSM

```
module control1(
        input clock,
        input reset,
        input [7:0] keyCode,
        input newCode,
        output reg countenable);

        reg [3:0] current_state, next_state;

        localparam      S_1     = 4'd0,
                        S_2     = 4'd1,
                        S_3     = 4'd2,
                        S_4     = 4'd3,
                        S_5     = 4'd4;

        //State Table
        always@(*)
        begin
        case(current_state)
                //Loop until W is pressed
                S_1: next_state = (newCode && keyCode == 8'h1D)? S_2: S_1;
                S_2: next_state = S_3; //Send enable signal
                //Loop until break code is received
                S_3: next_state = (newCode && keyCode == 8'hF0)? S_4: S_3;
                //Loop until W is sent again (from releasing the key)
                S_4: next_state = (newCode && keyCode == 8'h1D)? S_5: S_4;
                S_5: next_state = (!newCode)? S_1: S_5; //No new data
                default: next_state = S_1;
                endcase
        end

        //State Registers
        always@(*)
        begin
        countenable <= 1'b0;
        case(current_state)
                S_2: begin
                        countenable <= 1'b1;
                        end
        endcase
        end
        //State Transitions
        always@(posedge clock)
```

```
	begin
	if(!reset)
		current_state <= S_1;
	else
		current_state <= next_state;
	end
endmodule

//Player 1 Ship Datapath
module p1Data (
	input clock,
	input reset,
	input rotate, update, init,
	output reg [8:0] x,
	output reg [7:0] y,
	output reg [1:0] p1Orientation);

	always@(posedge clock)
	begin
	if (!reset || init)                                  //Resetting the registers
		begin
		x <= 9'd5;                                   //Starting x pos
		y <= 8'd103;                                 //Starting y pos
		p1Orientation <= 2'b01;                      //Starting orientation
		end

	if(rotate)                                           //Rotate the ship
		begin
		if (p1Orientation == 2'b11)
			p1Orientation <= 2'b0;
		else
			p1Orientation <= p1Orientation + 1'b1;
		end

	if (update)                                          //Increment or Decrement Position
		begin
		if (p1Orientation == 2'b00)                  //Ship facing North
			begin
			//top bound
			if(y != 8'd3)
				begin
				y <= y - 1'b1;
				end
			end
		else if (p1Orientation == 2'b01)             //Ship facing East
			begin
			// right bound
			if( x != 9'd300)
				begin
				x <= x + 1'b1;
```

```
                        end
                end
        else if(p1Orientation == 2'b10)                 //Ship facing South
                begin
                // bottom bound
                if( y != 8'd221)
                        begin
                        y <= y + 1'b1;
                        end
                end
        else begin                                      //Ship facing West
                // left bound
                if( x != 9'd3)
                        begin
                        x <= x - 1'b1;
                        end
                end
        end
    end
endmodule

//Ship 2 Control FSM
module control2(
        input clock,
        input reset,
        input [7:0] keyCode,
        input newCode,
        output reg countenable);

        reg [3:0] current_state, next_state;
        localparam      S_1     = 4'd0,
                        S_2     = 4'd1,
                        S_3     = 4'd2,
                        S_4     = 4'd3,
                        S_5     = 4'd4;

        always@(*)
        begin
        case(current_state)
                //Loop until P is pressed
                S_1: next_state = (newCode && keyCode == 8'h4D)? S_2: S_1;
                S_2: next_state = S_3; //Send enable signal
                //Loop until break code is received
                S_3: next_state = (newCode && keyCode == 8'hF0)? S_4: S_3;
                //Loop until P is sent again (from releasing the key)
                S_4: next_state = (newCode && keyCode == 8'h4D)? S_5: S_4;
                S_5: next_state = (!newCode)? S_1: S_5; //No new data
                default: next_state = S_1;
                endcase
        end
```

```
	always@(*)
	begin
	countenable <= 1'b0;

	case(current_state)
	S_2: begin
		countenable <= 1'b1;
		end
	endcase
	end

	always@(posedge clock)
	begin
	if(!reset)
		current_state <= S_1;
	else
		current_state <= next_state;
	end
endmodule

//Player 2 Ship Datapath
module p2Data (
	input clock,
	input reset,
	input rotate, update, init,
	output reg [8:0] x,
	output reg [7:0] y,
	output reg [1:0] p2Orientation);

	always@(posedge clock)
	begin
	if (!reset || init)                                    //Resetting Registers
		begin
		x <= 9'd300;                                   // Starting x pos
		y <= 8'd103;                                   // Starting y pos
		p2Orientation <= 2'b11;                        //Starting orientation: west
		end

	if(rotate)                                     //Rotate enable signal was received
		begin
		if (p2Orientation == 2'b11)            //If facing west, reset so facing north
			p2Orientation <= 2'b0;
		else
			p2Orientation <= p2Orientation + 1'b1;                  //Rotate
		end

	if (update)                                                    //Increnting / Decrementing register
		begin
		if (p2Orientation == 2'b00)
```

```
                        begin
                        //top bound
                        if(y != 8'd3)
                                begin
                                y <= y - 1'b1;
                                end
                        end
                else if(p2Orientation == 2'b01)
                        begin
                        // right bound
                        if( x != 9'd300)
                                begin
                                x <= x + 1'b1;
                                end
                        end
                else if(p2Orientation == 2'b10)
                        begin
                        // bottom bound
                        if( y != 8'd221)
                                begin
                                y <= y + 1'b1;
                                end
                        end
                else
                        begin
                        // left bound
                        if( x != 9'd3)
                                begin
                                x <= x - 1'b1;
                                end
                        end
                end
        end
endmodule

//Player 1 Rocket FSM
module shotControl1(
        input clock,
        input reset,
        input [7:0] keyCode,
        input newCode,
        output reg shotEnable);

        reg [3:0] current_state, next_state;
        localparam      S_1     = 4'd0,
                        S_2     = 4'd1,
                        S_3     = 4'd2,
                        S_4     = 4'd3,
                        S_5     = 4'd4;
        //State Table
```

```
	always@(*)
	begin
	case(current_state)
		//Loop until Q is pressed
		S_1: next_state = (newCode && keyCode == 8'h15)? S_2: S_1;
		S_2: next_state = S_3; //Send enable signal
		//Loop until break code is detected
		S_3: next_state = (newCode && keyCode == 8'hF0)? S_4: S_3;
		//Loop until release confirms Q was released
		S_4: next_state = (newCode && keyCode == 8'h15)? S_5: S_4;
		S_5: next_state = (!newCode)? S_1: S_5; //No new Codes, return to start
		default: next_state = S_1;
		endcase
	end

	//State registers
	always@(*)
	begin
	shotEnable <= 1'b0;

	case(current_state)
		S_2: begin
			shotEnable <= 1'b1;
			end
	endcase
	end

	//State Transitions
	always@(posedge clock)
	begin
	if(!reset)
		current_state <= S_1;
	else
		current_state <= next_state;
	end
endmodule

//Player 2 Rocket FSM
module shotControl2(
	input clock,
	input reset,
	input [7:0] keyCode,
	input newCode,
	output reg shotEnable);

	reg [3:0] current_state, next_state;
	localparam	S_1	= 4'd0,
			S_2	= 4'd1,
			S_3	= 4'd2,
			S_4	= 4'd3,
```

```
                S_5     = 4'd4;

        //State Table
        always@(*)
        begin
        case(current_state)
                //Loop until O is pressed
                S_1: next_state = (newCode && keyCode == 8'h44)? S_2: S_1;
                S_2: next_state = S_3; //Send enable signal to rocket datapath
                //Loop until break code
                S_3: next_state = (newCode && keyCode == 8'hF0)? S_4: S_3;
                //O is released
                S_4: next_state = (newCode && keyCode == 8'h44)? S_5: S_4;
                S_5: next_state = (!newCode)? S_1: S_5; //No new Codes
                default: next_state = S_1;
                endcase
        end

        //State Registers
        always@(*)
        begin
        shotEnable <= 1'b0;

        case(current_state)
                S_2: begin
                        shotEnable <= 1'b1;
                        end
        endcase
        end

        //State Transitions
        always@(posedge clock)
        begin
        if(!reset)
                current_state <= S_1;
        else
                current_state <= next_state;
        end
endmodule


//Rocket Datapath
module rocket(
        input clock,
        input reset, init,
        input update,
        input fireShot,
        input [1:0] orientation,                        //Orientation of ship
        input [8:0] ship_x, x_rocket,                   //XY position of ship
        input [7:0] ship_y, y_rocket,                   //XY Position of opponent rocket
```

```
output reg [8:0] x_shot,                                    //XY position of own shot
output reg [7:0] y_shot,
output reg [1:0] shot_orientation,                          //Orientation of own shot
output reg shotInAir);                                      //State is theres a shot in the air

always@(posedge clock)
begin
if (!reset || init)                         //Resetting Registers
        begin
        x_shot <= 9'b0;
        y_shot <= 8'd3;
        shotInAir <= 0;
        end
else
        begin
        if (!shotInAir && fireShot)                 //If no shot on field and receives enable
                begin
                shotInAir <= 1'b1;
                if (orientation == 2'b00)                   //If players ship is facing North
                        begin
                        x_shot <= ship_x + 4'd8;    //Starting position of rocket
                        y_shot <= ship_y - 2'd3;
                        shot_orientation <= 2'b00;  //Orientation of rocket
                        end
                else if (orientation == 2'b01)      //Ship facing East
                        begin
                        x_shot <= ship_x + 6'd18;
                        y_shot <= ship_y + 4'd7;
                        shot_orientation <= 2'b01;
                        end
                else if (orientation == 2'b10)      //Ship facing South
                        begin
                        x_shot <= ship_x + 4'd8;
                        y_shot <= ship_y + 6'd18;
                        shot_orientation <= 2'b10;
                        end
                else                                        //Ship facing west
                        begin
                        x_shot <= ship_x - 2'd3;
                        y_shot <= ship_y + 4'd7;
                        shot_orientation <= 2'b11;
                        end
                end

        if (update && shotInAir)                    //Update signal from Draw FSM
                begin
                //First check if the two shots have collided with each other
                if ((x_shot >= x_rocket && x_shot <= x_rocket + 8'd2 &&  y_shot >= y_rocket && y_shot <=
y_rocket + 8'd2) || (x_shot+8'd2 >= x_rocket && x_shot+8'd2<= x_rocket+8'd2 && y_shot >= y_rocket && y_shot <= y_rocket
```

```
+ 8'd2) || (x_shot >= x_rocket && x_shot <= x_rocket + 8'd2 && y_shot+8'd2 >= y_rocket && y_shot <= y_rocket+8'd2) ||
(x_shot+8'd2 >= x_rocket && x_shot+8'd2<= x_rocket+8'd2 && y_shot+8'd2 >= y_rocket && y_shot <= y_rocket+8'd2))
                        shotInAir <= 0;                         //Shots cancel out, no more shots on field
                        //No collision with shots, update registers
                        if (shot_orientation <= 2'b00) //Shot moving North
                                begin
                                if (y_shot <= 2'd1)
                                        shotInAir <= 0;
                                else
                                        y_shot <= y_shot -2;
                                end
                        else if (shot_orientation <= 2'b01)     //Shot moving East
                                begin
                                if(x_shot >= 10'd319)
                                        shotInAir <= 0;
                                else
                                        x_shot <= x_shot + 2;
                                end
                        else if(shot_orientation <= 2'b10)      //Shot moving south
                                begin
                                if(y_shot >= 9'd239)
                                        shotInAir <= 0;
                                else
                                        y_shot <= y_shot + 2;
                                end
                        else                                            //Shot moving West
                                begin
                                if( x_shot <= 2'd1)
                                        shotInAir <= 0;
                                else
                                        x_shot <= x_shot - 2;
                                end
                        end
                end
        end
endmodule
```

//Collision Detection

```
module collision(
        input clock,
        input reset, init,
        input shotInAir1, shotInAir2,
        input [9:0] x1_pos, x2_pos,             //X position of ships
        input [8:0] y1_pos, y2_pos,             //Y position of ships
        input [9:0] x1shot, x2shot,             //X position of rockets
        input [8:0] y1shot, y2shot,             //Y position of rockets
        output reg p1Hit, p2Hit, shipsCollide); //Output signals

        always@(posedge clock)
        begin
```

```
	if (!reset || init)                                    //Reset registers
		begin
		p1Hit <= 0;
		p2Hit <= 0;
		shipsCollide <= 0;
		end
	else
		begin
		//If player 2 is hit by player 1 rocket
		if ((x1shot >= x2_pos && x1shot <= x2_pos + 8'd17) && (y1shot >= y2_pos && y1shot <= y2_pos + 8'd17)
&& shotInAir1)
			p2Hit <= 1;

		//If player 1 is hit by player 2 rocket
		if ((x2shot >= x1_pos && x2shot <= x1_pos + 8'd17) && (y2shot >= y1_pos && y2shot <= y1_pos + 8'd17)
&& shotInAir2)
			p1Hit <= 1;

		//If ships collide with each other
		if ((x2_pos >= x1_pos && x2_pos <= x1_pos + 8'd15 && y2_pos >= y1_pos && y2_pos <= y1_pos +
8'd15) ||
		(x2_pos+8'd15 >= x1_pos && x2_pos+8'd15<= x1_pos+8'd15 && y2_pos >= y1_pos && y2_pos <=
y1_pos + 8'd15) ||
		(x2_pos >= x1_pos && x2_pos <= x1_pos + 8'd15 && y2_pos+8'd15 >= y1_pos && y2_pos <=
y1_pos+8'd15) ||
		(x2_pos+8'd15 >= x1_pos && x2_pos+8'd15<= x1_pos+8'd15 && y2_pos+8'd15 >= y1_pos && y2_pos <=
y1_pos+8'd15))
			shipsCollide <= 1;
		end
	end
endmodule

//ScoreCounter module
module scoreCounter(
	input clock,
	input reset,
	input newGame,
	input p1Hit,
	input p2Hit,
	output reg [3:0] scoreP1, scoreP2,
	output reg p1Win, p2Win);

	always@(posedge clock)
	begin
	if (!reset || newGame || p1Win || p2Win)               //If reset or a newGame is starting
		begin
		scoreP1 <= 4'b0;
		scoreP2 <= 4'b0;
		p1Win <= 1'b0;
		p2Win <= 1'b0;
```

```
		end
	else
		begin
		if (scoreP1 == 4'd10)				//Check if P1 has reached 10 points
			begin
			p1Win <= 1'b1;				//P1 Wins
			end
		else if(scoreP2 == 4'd10)		//Check if P2 has reached 10 points
			begin
			p2Win <= 1'b1;				//P2 Wins
			end
		else						//No winner, update the scores
			begin
			if (p1Hit)				//Player 1 hit by rocket
				scoreP2 <= scoreP2 + 1'b1;	//Increment player 2 score
			if (p2Hit)				//Player 2 hit by rocket
				scoreP1 <= scoreP1 + 1'b1;	//Increment player 1 score
			end
		end
	end
endmodule
```

//Display Control FSM

```
module displayCtrl (
	input clock,
	input reset,
	input p1Hit, p2Hit, newGame, shipsCollide, shotInAir1, shotInAir2, gameStart, playGame,
	input p1Win, p2Win, menuScreen,
	input [9:0] dotCounter,
	input [17:0] blackCounter,
	input [24:0] freqCounter,
	output reg ld_plot, erase, update, init, ld_p1, ld_p2, ld_rkt1, ld_rkt2,
	output reg newRound, countReset, ld_menu, ld_p1win, ld_p2win);

	reg [4:0] current_state, next_state;

	localparam	S_MENU			= 5'd0,
			S_INIT			= 5'd1,
			S_WAIT			= 5'd2,
			S_DRAWP1		= 5'd3,
			S_COUNTRESET1		= 5'd4,
			S_DRAWP2		= 5'd5,
			S_COUNTRESET2		= 5'd6,
			S_DRAWRKT1		= 5'd7,
			S_COUNTRESET3		= 5'd8,
			S_DRAWRKT2		= 5'd9,
			S_COUNTRESET4		= 5'd10,
			S_DRAW_WAIT		= 5'd11,
			S_ERASEP1		= 5'd12,
```

```
                S_COUNTRESET5       = 5'd13,
                S_ERASEP2           = 5'd14,
                S_COUNTRESET6       = 5'd15,
                S_ERASERKT1         = 5'd16,
                S_COUNTRESET7       = 5'd17,
                S_ERASERKT2         = 5'd18,
                S_UPDATE            = 5'd19,
                S_P1WIN             = 5'd20,
                S_P2WIN             = 5'd21,
                S_COUNTRESET8       = 5'd22,
                S_COUNTRESET9       = 5'd23;

        always@(*)
        begin: state_table
        case(current_state)
                S_MENU: next_state = (blackCounter <= 18'd76799)? S_MENU: S_COUNTRESET9;   //Draw Menu
                S_INIT: next_state = (blackCounter <= 18'd76799)? S_INIT: S_WAIT;          //Draw Background
                S_WAIT: next_state = gameStart? S_DRAWP1: S_WAIT;                         //Delay for 1 s
                S_DRAWP1: next_state = (dotCounter <= 10'b0100100000)? S_DRAWP1: S_COUNTRESET1; //Draw P1
                S_COUNTRESET1: next_state = S_DRAWP2;                                     //Reset Counters
                S_DRAWP2: next_state= (dotCounter <= 10'b0100100000)? S_DRAWP2: S_COUNTRESET2; //Draw P2
                S_COUNTRESET2: next_state = (shotInAir1)? S_DRAWRKT1: S_COUNTRESET3;      //Reset Counters
                S_DRAWRKT1: next_state = (dotCounter <= 10'd8)? S_DRAWRKT1: S_COUNTRESET3; //Draw P1 RKT
                S_COUNTRESET3: next_state = (shotInAir2)? S_DRAWRKT2: S_COUNTRESET4;      //Reset Counters
                S_DRAWRKT2: next_state = (dotCounter <= 10'd8)? S_DRAWRKT2: S_COUNTRESET4; //Draw P2 RKT
                S_COUNTRESET4: next_state = S_DRAW_WAIT;                                  //Reset Counters
                S_DRAW_WAIT: next_state = (freqCounter < 25'd833333)? S_DRAW_WAIT: S_ERASEP1; //Delay 60Hz
                S_ERASEP1: next_state = (dotCounter <= 10'b0100100000)? S_ERASEP1 : S_COUNTRESET5; //Erase P1
                S_COUNTRESET5: next_state = S_ERASEP2;                                    //Reset Counters
                S_ERASEP2: next_state = (dotCounter <= 10'b0100100000)? S_ERASEP2 : S_COUNTRESET6; //Erase P2
                S_COUNTRESET6: next_state = S_ERASERKT1;                                  //Reset Counters
                S_ERASERKT1: next_state = (dotCounter <= 10'd8)? S_ERASERKT1: S_COUNTRESET7;//Erase P1 RKT
                S_COUNTRESET7: next_state = S_ERASERKT2;                                  //Reset Counters
                S_ERASERKT2: next_state = (dotCounter <= 10'd8)? S_ERASERKT2: S_UPDATE;   //Erase P2 RKT
                S_UPDATE: next_state = S_DRAWP1;                                          //Update Registers
                S_P1WIN: next_state = (blackCounter <= 18'd76799)? S_P1WIN: S_COUNTRESET8; //P1 Win Screen
                S_P2WIN: next_state = (blackCounter <= 18'd76799)? S_P2WIN: S_COUNTRESET8; //P2 Win Screen
                S_COUNTRESET8: next_state = S_MENU;                                       //Reset Counters
                S_COUNTRESET9: next_state = S_INIT;                                       //Reset Counters
                default: next_state = S_MENU;
        endcase
end

//State Registers
always@(*)
begin: enable_signals //Initial output values
ld_plot                 = 1'b0;
update                  = 1'b0;
erase                   = 1'b0;
init                    = 1'b0;
```

```
ld_p1                   = 1'b0;
ld_p2                   = 1'b0;
ld_rkt1                 = 1'b0;
ld_rkt2                 = 1'b0;
countReset              = 1'b0;
newRound                = 1'b0;
ld_menu                 = 1'b0;
ld_p1win                = 1'b0;
ld_p2win                = 1'b0;

case(current_state)
        S_MENU: begin                                   //Menu State, send plot and ld menu signals
                ld_plot <= 1'b1;
                ld_menu <= 1'b1;
                end
        S_INIT: begin                                   //Init state
                init            = 1'b1;
                newRound        = 1'b1;
                ld_plot         = 1'b1;
                end
        S_WAIT: begin
                init            = 1'b1;
                end
        S_DRAWP1: begin                                 //Send signals to Draw Player 1
                ld_plot         = 1'b1;
                erase           = 1'b0;
                ld_p1           = 1'b1;
                end
        S_COUNTRESET1: begin
                countReset = 1'b1;
                end
        S_DRAWP2: begin                                 //Send signals to Draw Player 1
                ld_plot    = 1'b1;
                erase      = 1'b0;
                ld_p2           = 1'b1;
                end
        S_COUNTRESET2: begin
                countReset = 1'b1;
                end
        S_DRAWRKT1: begin                               //Send signals to Draw Player 1 Rocket
                ld_plot    = 1'b1;
                ld_rkt1    = 1'b1;
                end
        S_COUNTRESET3: begin
                countReset = 1'b1;
                end
        S_DRAWRKT2: begin                               //Send signals to Draw Player 2 Rocket
                ld_plot    = 1'b1;
                ld_rkt2    = 1'b1;
                end
```

```
        S_COUNTRESET4: begin
                countReset = 1'b1;
                end
        S_ERASEP1: begin                                    //Send signals to Erase Player 1
                erase       = 1'b1;
                ld_plot     = 1'b1;
                ld_p1               = 1'b1;
                end
        S_COUNTRESET5: begin
                countReset = 1'b1;
                end
        S_ERASEP2: begin                                    //Send signals to Erase Player 2
                erase       = 1'b1;
                ld_plot     = 1'b1;
                ld_p2               = 1'b1;
                end
        S_COUNTRESET6: begin
                countReset = 1'b1;
                end
        S_ERASERKT1: begin                                  //Send signals to Erase Player 2 Rocket
                ld_plot     = 1'b1;
                erase               = 1'b1;
                ld_rkt1     = 1'b1;
                end
        S_COUNTRESET7: begin
                countReset = 1'b1;
                end
        S_ERASERKT2: begin                                  //Send signals to Erase Player 2 Rocket
                ld_plot     = 1'b1;
                erase       = 1'b1;
                ld_rkt2     = 1'b1;
                end
        S_UPDATE: begin                                     //Send signals to update the data registers
                update      = 1'b1;
                end
        S_P1WIN: begin                                      //P1 Won, send signals to load p1 win screen
                ld_plot = 1'b1;
                ld_p1win = 1'b1;
                end
        S_P2WIN: begin                                      //P1 Won, send signals to load p1 win screen
                ld_plot = 1'b1;
                ld_p2win = 1'b1;
                end
        S_COUNTRESET8: begin
                countReset = 1'b1;
                end
        S_COUNTRESET9: begin
                countReset = 1'b1;
                end
endcase
```

```
end

//State Transitions
always@(posedge clock)
	begin: state_FFs
	if(!reset || menuScreen)                          //If reset or in menu State from game control FSM
		current_state <= S_MENU;
	else if (shipsCollide || p1Hit || p2Hit)          //If ships collide from collision module
			current_state <= S_INIT;          //Reset the round
	else if (p1Win)                                   //P1 Wins
		current_state <= S_P1WIN;
	else if (p2Win)                                   //P2 Wins
		current_state <= S_P2WIN;
	else if (playGame)                                //Play enable signal from game control FSM
		current_state <= next_state;              //Allows the draw FSM to advance to next states
	end
endmodule

//Display Datapath Module
module displayData (
	input clock,
	input reset,
	input erase, init, ld_plot, ld_p1, ld_p2, ld_rkt1, ld_rkt2, countReset, ld_menu, ld_p1win, ld_p2win,
	input [8:0] x_pos1, x_pos2, x_rkt1, x_rkt2,                    //X position of ships, rockets
	input [7:0] y_pos1, y_pos2, y_rkt1, y_rkt2,                    //Y position of ships, rockets
	input [1:0] p1Orientation, p2Orientation,                      //Orientation of ships
	input [1:0] rkt1Orientation, rkt2Orientation,                  // Orientation of rockets
	output reg [8:0] x_vga,                                        //x out to VGA
	output reg [7:0] y_vga,                                        //Y out to VGA
	output reg [2:0] colour_out,                                   //Colour data to VGA
	output reg [9:0] dotCounter,
	output reg [24:0] freqCounter,
	output reg [17:0] blackCounter);

	//Counter register
	reg [5:0] x_counter;
	reg [5:0] y_counter;
	reg [8:0] blackCounter_x;
	reg [7:0] blackCounter_y;

	//colour data wires
	wire [2:0] colourp1North, colourp1East, colourp1South, colourp1West;          //Player 1
	wire [2:0] colourp2North, colourp2East, colourp2South, colourp2West;          //Player 2
	wire [2:0] menuTL, menuTR, menuBL, menuBR;                                    //Menu Quadrants
	wire [2:0] P1WinTL, P1WinTR, P1WinBL, P1WinBR;                                //P1 Screen Quadrants
	wire [2:0] P2WinTL, P2WinTR, P2WinBL, P2WinBR;                                //P2 Screen Quadrants


	//Address for accessing memory modules
	wire [8:0] address;
```

```
assign address = (y_counter * 17) + x_counter;                  //For accessing mif files for ship drawing
reg[8:0] bg_x;
reg [7:0] bg_y;
wire [14:0] bgTL, bgTR, bgBL, bgBR;
assign bgTL = (bg_y * 160) + bg_x;                              //For accessing mif files in top left quadrant
assign bgTR = (bg_y * 160) + (bg_x - 159);                      //For accessing mif files in top Right quadrant
assign bgBL = ((bg_y - 120) * 160) + bg_x;                      //For accessing mif files in Bottom left quadrant
assign bgBR = ((bg_y - 120) * 160) + (bg_x - 159);              //For accessing mif files in top right quadrant

//Getting colour data
p1North c1 (.address(address), .clock(clock), .q(colourp1North));   //Player 1 Colour Data
p1East c2 (.address(address), .clock(clock), .q(colourp1East));
p1South c3 (.address(address), .clock(clock), .q(colourp1South));
p1West c4 (.address(address), .clock(clock), .q(colourp1West));

p2North c5 (.address(address), .clock(clock), .q(colourp2North));   //Player 2 Colour Data
p2East c6 (.address(address), .clock(clock), .q(colourp2East));
p2South c7 (.address(address), .clock(clock), .q(colourp2South));
p2West c8 (.address(address), .clock(clock), .q(colourp2West));

MenuTL c9 (.address(bgTL), .clock(clock), .q(menuTL));          //Menu Screen Colour Data
MenuTR c10 (.address(bgTR), .clock(clock), .q(menuTR));
MenuBL c11 (.address(bgBL), .clock(clock), .q(menuBL));
MenuBR c12 (.address(bgBR), .clock(clock), .q(menuBR));

p1TL c13 (.address(bgTL), .clock(clock), .q(P1WinTL));          //P1 Win screen colour data
p1TR c14 (.address(bgTR), .clock(clock), .q(P1WinTR));
p1BL c15 (.address(bgBL), .clock(clock), .q(P1WinBL));
p1BR c16 (.address(bgBR), .clock(clock), .q(P1WinBR));

p2TL c17 (.address(bgTL), .clock(clock), .q(P2WinTL));          //P2 Win screen colour data
p2TR c18 (.address(bgTR), .clock(clock), .q(P2WinTR));
p2BL c19 (.address(bgBL), .clock(clock), .q(P2WinBL));
p2BR c20 (.address(bgBR), .clock(clock), .q(P2WinBR));

always@(posedge clock)
begin
if(!reset)                                                      //Reset registers
        begin
        x_vga <= 9'b0;
        y_vga <= 8'b0;
        bg_x <= 9'd0;
        bg_y <= 8'd0;
        dotCounter <= 10'b0;
        colour_out <= 3'b0;
        blackCounter_x <= 9'b0;
        blackCounter_y <= 8'b0;

        x_counter <= 6'b0;
        y_counter <= 6'b0;
```

```
		blackCounter <= 18'b0;
		freqCounter <= 25'b0;
		end
else
		begin
		if (init && ld_plot)                                    //Init state, draw background
			begin
			if (blackCounter == 18'd76800)                  //Counts the number of pixels up to 320x240
				begin
				blackCounter <= 18'b0;
				end
			else
				begin
				x_vga <= blackCounter_x;
				y_vga <= blackCounter_y;
				blackCounter <= blackCounter + 1'b1;
				if(x_vga == y_vga*y_vga*y_vga*y_vga*y_vga && x_vga != 9'd0) //draw stars function
					colour_out <= 3'b110;
				else
					colour_out <= 3'b0;

				if (blackCounter_x == 9'd320)
					begin
					if (blackCounter_y == 8'd240)                   //Counter has reached the end
						begin
						blackCounter_x <= 9'b0;
						blackCounter_y <= 8'b0;
						end
					else                                            //Reached end of row, go to next row
						begin
						blackCounter_y <= blackCounter_y + 1'b1;
						blackCounter_x <= 9'b0;
						end
					end
				else
					blackCounter_x <= blackCounter_x + 1'b1;        //Update 1 pixel to right
				end
			end



		if (countReset)                                         //Count reset signal, resets all counters/ registers
			begin
			dotCounter <= 10'b0;
			x_counter <= 9'b0;
			y_counter <= 8'b0;
			blackCounter <= 17'b0;
			blackCounter_x <= 9'b0;
			blackCounter_y <= 8'b0;
			bg_x <= 9'd0;
```

```
		bg_y <= 8'd0;
		end


if (ld_menu && ld_plot)				//Loading the menu screen into the VGA adapter
		begin
		if (blackCounter == 18'd76800)
			begin
			blackCounter <= 18'b0;
			end
		else
			begin
			x_vga <= bg_x;
			y_vga <= bg_y;
			if (bg_x <= 9'd159 && bg_y <= 8'd119)		//XY in top left quadrant
				colour_out <= menuTL;

			if (bg_x >= 9'd160 && bg_y <= 8'd119)		//XY in top right quadrant
				colour_out <= menuTR;

			if (bg_x <= 9'd159 && bg_y >= 8'd120)		//XY in bottom left quadrant
				colour_out <= menuBL;

			if (bg_x >= 9'd160 && bg_y >= 8'd120)		//XY in bottom right quadrant
				colour_out <= menuBR;

			if (bg_x == 9'd320)
				begin
				if (bg_y == 8'd240)
					begin
					bg_x <= 9'b0;
					bg_y <= 8'b0;
					end
				else
					begin
					bg_y <= bg_y + 1'b1;
					bg_x <= 9'b0;
					end
				end
			else
				bg_x <= bg_x + 1'b1;
				blackCounter <= blackCounter + 1'b1;
			end
		end

if (ld_p1win && ld_plot)					//Load p1 win screen to VGA
		begin
		if (blackCounter == 18'd76800)
			begin
			blackCounter <= 18'b0;
```

```
		end
else
		begin
		x_vga <= bg_x;
		y_vga <= bg_y;
		if (bg_x <= 9'd159 && bg_y <= 8'd119)			//Top Left quadrant
			colour_out <=P1WinTL;

		if (bg_x >= 9'd160 && bg_y <= 8'd119)			//Top right quadrant
			colour_out <= P1WinTR;

		if (bg_x <= 9'd159 && bg_y >= 8'd120)			//Bottom left quadrant
			colour_out <= P1WinBL;

		if (bg_x >= 9'd160 && bg_y >= 8'd120)			//Bottom right quadrant
			colour_out <= P1WinBR;

		if (bg_x == 9'd320)
			begin
			if (bg_y == 8'd240)
				begin
				bg_x <= 9'b0;
				bg_y <= 8'b0;
				end
			else
				begin
				bg_y <= bg_y + 1'b1;
				bg_x <= 9'b0;
				end
			end
		else
			bg_x <= bg_x + 1'b1;
			blackCounter <= blackCounter + 1'b1;
		end
end


if (ld_p2win && ld_plot)					//Load P2 Win screen to VGA
begin
if (blackCounter == 18'd76800)
		begin
		blackCounter <= 18'b0;
		end
else
		begin
		x_vga <= bg_x;
		y_vga <= bg_y;
		if (bg_x <= 9'd159 && bg_y <= 8'd119)
			colour_out <= P2WinTL;
```

```
                if (bg_x >= 9'd160 && bg_y <= 8'd119)
                        colour_out <= P2WinTR;

                if (bg_x <= 9'd159 && bg_y >= 8'd120)
                        colour_out <= P2WinBL;

                if (bg_x >= 9'd160 && bg_y >= 8'd120)
                        colour_out <= P2WinBR;

                if (bg_x == 9'd320)
                        begin
                        if (bg_y == 8'd240)
                                begin
                                bg_x <= 9'b0;
                                bg_y <= 8'b0;
                                end
                        else
                                begin
                                bg_y <= bg_y + 1'b1;
                                bg_x <= 9'b0;
                                end
                        end
                else
                        bg_x <= bg_x + 1'b1;
                        blackCounter <= blackCounter + 1'b1;
                end
        end

if(ld_p1 && erase && ld_plot)                                   //Erasing P1 ship
        begin
        if (dotCounter == 10'b0100100001)
                begin
                dotCounter <= 10'b0;
                x_counter <= 9'b0;
                y_counter <= 8'b0;
                end
        else
        begin
                x_vga <= x_pos1 + x_counter;
                y_vga <= y_pos1 + y_counter;
                dotCounter <= dotCounter + 1'b1;
                if(x_vga == y_vga*y_vga*y_vga*y_vga*y_vga && x_vga != 9'd0)
                                colour_out <= 3'b110;
                else
                        colour_out <= 3'b0;


                if (x_counter == 6'd17)
                        begin
                        if (y_counter == 6'd17)
```

```
                        begin
                        x_counter <= 6'b0;
                        y_counter <= 6'b0;
                        end
                else
                        begin
                        y_counter <= y_counter + 1'b1;
                        x_counter <= 6'b0;
                        end
                end
        else
                x_counter <= x_counter + 1'b1;
        end
end


if(ld_p2 && erase && ld_plot)                   //Erasing p2 ship
begin
if (dotCounter == 10'b0100100001)
        begin
        dotCounter <= 10'b0;
        x_counter <= 9'b0;
        y_counter <= 8'b0;
        end
else
begin
        x_vga <= x_pos2 + x_counter;
        y_vga <= y_pos2 + y_counter;
        dotCounter <= dotCounter + 1'b1;
        if(x_vga == y_vga*y_vga*y_vga*y_vga*y_vga && x_vga != 9'd0)
                        colour_out <= 3'b110;
        else
                colour_out <= 3'b0;

        if (x_counter == 6'd17)
                begin
                if (y_counter == 6'd17)
                        begin
                        x_counter <= 6'b0;
                        y_counter <= 6'b0;
                        end
                else
                        begin
                        y_counter <= y_counter + 1'b1;
                        x_counter <= 6'b0;
                        end
                end
        else
                x_counter <= x_counter + 1'b1;
        end
```

```
end


if (ld_plot && !erase && ld_p1)                         //Drawing out Player 1 ship
        begin
        if (dotCounter == 10'b0100100001)
        begin
        dotCounter <= 10'b0;
        x_counter <= 6'b0;
        y_counter <= 6'b0;
        end
else
        begin
        x_vga <= x_pos1 + x_counter;
        y_vga <= y_pos1 + y_counter;
        dotCounter <= dotCounter + 1'b1;
        if (p1Orientation == 2'b00)             //Facing north, use north oriented colour data
                begin
                colour_out <= colourp1North;
                end

        if (p1Orientation == 2'b01)             //Facing east, use north oriented colour data
                begin
                colour_out <= colourp1East;
                end

        if (p1Orientation == 2'b10)             //Facing South, use north oriented colour data
                begin
                colour_out <= colourp1South;
                end

        if (p1Orientation == 2'b11)             //Facing West, use north oriented colour data
                begin
                colour_out <= colourp1West;
                end

        if (x_counter == 6'd17)
                begin
                if (y_counter == 6'd17)
                        begin
                        x_counter <= 6'b0;
                        y_counter <= 6'b0;
                        end
                else
                        begin
                        y_counter <= y_counter + 1'b1;
                        x_counter <= 6'b0;
                        end
                end
        else
```

```
                x_counter <= x_counter + 1'b1;

        end
end


if (ld_plot && !erase && ld_p2)                 //Draw p2 ship
        begin
        if (dotCounter == 10'b0100100001)
        begin
        dotCounter <= 10'b0;
        x_counter <= 6'b0;
        y_counter <= 6'b0;
        end
else
        begin
        x_vga <= x_pos2 + x_counter;
        y_vga <= y_pos2 + y_counter;
        dotCounter <= dotCounter + 1'b1;
        if (p2Orientation == 2'b00)             //Facing north, use north oriented colour data
                begin
                colour_out <= colourp2North;
                end

        if (p2Orientation == 2'b01)             //Facing north, use north oriented colour data
                begin
                colour_out <= colourp2East;
                end

        if (p2Orientation == 2'b10)             //Facing north, use north oriented colour data
                begin
                colour_out <= colourp2South;
                end

        if (p2Orientation == 2'b11)             //Facing north, use north oriented colour data
                begin
                colour_out <= colourp2West;
                end

        if (x_counter == 6'd17)
                begin
                if (y_counter == 6'd17)
                        begin
                        x_counter <= 6'b0;
                        y_counter <= 6'b0;
                        end
                else
                        begin
                        y_counter <= y_counter + 1'b1;
                        x_counter <= 6'b0;
```

```
                        end
                end
        else
                x_counter <= x_counter + 1'b1;

        end
end


//Drawing and erasing rockets
if (ld_plot && !erase && ld_rkt1)                       //Draw P1 Rocket 3x3 pixel square
        begin
                if (dotCounter == 10'd9)
                        begin
                        dotCounter <= 10'b0;
                        end
                else
                        begin
                        if (x_counter == 6'd3 && y_counter !== 6'd3)
                                begin
                                y_counter <= y_counter + 1'b1;
                                x_counter <= 6'b0;
                                end
                        else
                                begin
                                x_counter <= x_counter + 1'b1;
                                x_vga <= x_rkt1 + x_counter;
                                y_vga <= y_rkt1 + y_counter;
                                colour_out <= 3'b100;
                                dotCounter <= dotCounter + 1;
                                end
                        end
                end

if (ld_plot && !erase && ld_rkt2)                       //Draw P2 Rocket 3x3 pixel square
        begin
                if (dotCounter == 10'd9)
                        begin
                        dotCounter <= 10'b0;
                        end
                else
                        begin
                        if (x_counter == 6'd3 && y_counter !== 6'd3)
                                begin
                                y_counter <= y_counter + 1'b1;
                                x_counter <= 6'b0;
                                end
                        else
                                begin
                                x_counter <= x_counter + 1'b1;
```

```
                                        x_vga <= x_rkt2 + x_counter;
                                        y_vga <= y_rkt2 + y_counter;
                                        colour_out <= 3'b010;
                                        dotCounter <= dotCounter + 1;
                                        end
                                end
                        end


if (ld_plot && erase && ld_rkt1)                        //Erasing P1 Rocket
        begin
                if (dotCounter == 10'd9)
                        begin
                        dotCounter <= 10'b0;
                        x_counter <= 6'd0;
                        y_counter <= 6'd0;
                        end
                else
                        begin
                        if(x_vga == y_vga*y_vga*y_vga*y_vga*y_vga && x_vga != 9'd0)
                                colour_out <= 3'b110;
                        else
                        colour_out <= 3'b0;

                        if (x_counter == 6'd3 && y_counter !== 6'd3)
                                begin
                                y_counter <= y_counter + 1'b1;
                                x_counter <= 6'b0;
                                end
                        else
                                begin
                                x_counter <= x_counter + 1'b1;
                                x_vga <= x_rkt1 + x_counter;
                                y_vga <= y_rkt1 + y_counter;
                                dotCounter <= dotCounter + 1;
                                end
                        end
                end


if (ld_plot && erase && ld_rkt2)                        //Erasing P2 Rocket
        begin
                if (dotCounter == 10'd9)
                        begin
                        dotCounter <= 10'b0;
                        x_counter <= 6'd0;
                        y_counter <= 6'd0;
                        end
```

```
                else
                        begin
                        if(x_vga == y_vga*y_vga*y_vga*y_vga*y_vga && x_vga != 9'd0)
                                colour_out <= 3'b110;
                        else
                        colour_out <= 3'b0;

                        if (x_counter == 6'd3 && y_counter !== 6'd3)
                                begin
                                y_counter <= y_counter + 1'b1;
                                x_counter <= 6'b0;
                                end
                        else
                                begin
                                x_counter <= x_counter + 1'b1;
                                x_vga <= x_rkt2 + x_counter;
                                y_vga <= y_rkt2 + y_counter;
                                dotCounter <= dotCounter + 1;
                                end
                        end
                end

        //Freq Counter, used to pause the FSM to a rate where updates happen at 60Hz
        if (freqCounter == 25'd833333)
                begin
                freqCounter <= 25'd0;
                end
        else
                freqCounter <= freqCounter +1;
        end
    end
endmodule
```