Asteroids

6.111 Final Report

Daniel Hawkins
Anna Waldo
December 12, 2012

Abstract

Asteroids was an extremely popular and influential video arcade game released in 1979. In the game, the player controls a spaceship in an asteroid field; the objective is to shoot and destroy incoming asteroids while avoiding collision with them. For our 6.111 final project, we used the FPGA labkit and additional hardware to put our own spin on the classic Asteroids game. In addition to recreating the 2D vector graphics and some of the sound effects of the original game, we added a new input device; instead of a joystick, our Asteroids game is controlled by a 3-axis accelerometer.

Contents

1 Overview

In our version of Asteroids, the player controls the spaceship in the middle of the screen with a handheld device containing an accelerometer. Depending on how the accelerometer is tilted, the spaceship will rotate and shoot bullets. When the game starts, asteroids begin to float across the screen at a given speed and direction. The player's goal is to shoot and destroy each of the asteroids and prevent them from hitting the spaceship. When the ship is hit, a life is lost. There is also a sound output for bullet firing and asteroid collision events.

The are several distinct differences between the original version of Asteroids and our own implementation. Firstly, rather than using button inputs, we opted to use an accelerometer for a hand-held controller. By tilting the accelerometer at certain angles, the player can move the ship and shoot bullets. This creates a more intuitive game play. Secondly, due to time constraints, the scoring system of our game is different from the original. Rather than tallying points every time an asteroid is shot, we simply allow the player three lives. Each time an asteroid collides with the spaceship, a life is lost. The game ends when the third life is lost. Also, while in the original game the large asteroids split into smaller fragments after first being shot, we opted to have only one asteroid size that are destroyed immediately when shot. Additionally, while in the original game the spaceship can move around the screen, in our version we have the ship fixed in the middle.

2 Implementation

The main components of our implementation of Asteroids are the analog input, the analog to digital conversion, graphics, game logic, and sound. The graphics implementation is subsequently divided into the ship, bullets, and asteroids. The analog input controls the ship and bullets, while the game logic determines how each of the objects interact on screen and when there should be sound output.

2.1 Analog Input (Anna)

We use an accelerometer as our user control for the game. This input determines the behavior of the ship and firing bullets. The accelerometer we used was the DE-ACCM3D, which has three axes, and produces a certain voltage output signal depending on its tilt and movement. For our game, we only needed two axes, one for spaceship rotation and one for firing bullets. Tilting the accelerometer ninety degrees to the right causes the spaceship to rotate right, and tilting the accelerometer ninety degrees to the left rotates the ship left. Tilting the accelerometer forward causes a bullet to fire. The accelerometer is mounted on a breadboard, and the user holds the breadboard upright and moves it to play the game.

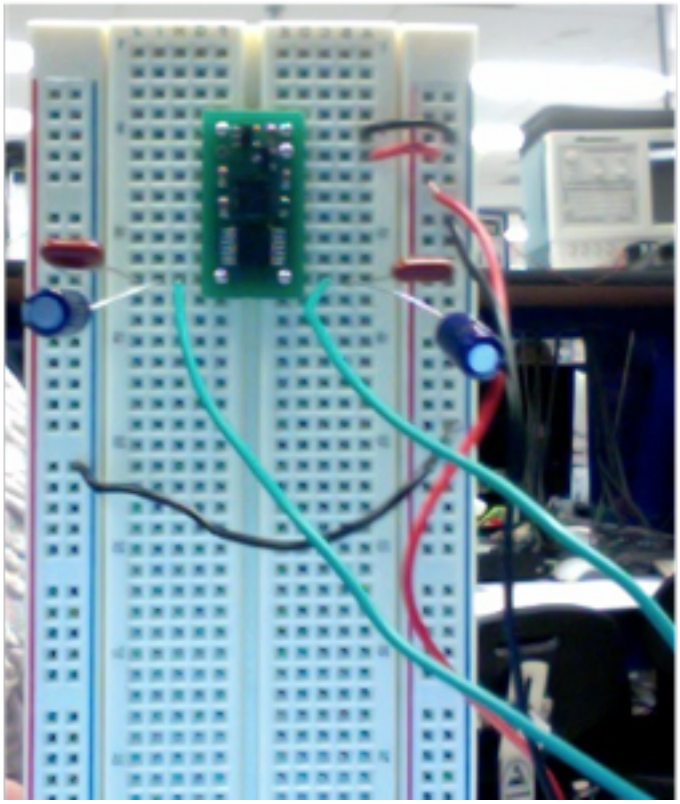

Figure 1. The accelerometer on its breadboard mount. From the above upright position, turning the board ninety degrees the the right will rotate the ship to the right, turning the board ninety degrees to the left will rotate the ship left, and tilting the board forward (into the page) will shoot a bullet. The capacitors prevent excess noise. The two green wires, one of which it outputs the signal from the x axis, and the other form the z axis, carry the analog signals and connect to the analog to digital converter chip.

2.2 Analog to Digital Conversion (Anna)

Because the accelerometer outputs an analog signal, and the FPGA requires a digital signal, we need to convert the analog signal from the accelerometer output into a digital signal to feed into the FPGA and subsequently control the game. This is done with an Analog to Digital Converter (ADC) chip. The chip we used is an Analog Devices AD7824KN ADC, as seen in Figure 2. It can take in four analog input channels, and outputs an eight bit digital output signal for each of the channels. Although the ADC does the actual analog to digital conversion on its own, we still need to program it in order to get useful and valid data. In order to do so, we need to use the other ADC components, which consist of a two-bit address channel, a ready signal, and a read signal.

The address channel determines which of the four analog input channels we receiving a digital output from. Because we are only using two of the input channels, we can ground one bit of the address channel and alternate the other channel to cycle between the two inputs. The read signal is controlled by the FPGA and is an input into the ADC. After we have changed the address to read the x or z axis from the accelerometer, we set the read signal to 0. This signals the ADC to begin converting the data from the analog input specified by the address channel into a new digital output. Once this data is valid, the ADC outputs a ready signal of 1. Once the FPGA receives this, we know the data is valid and we can now use the eight bits of data for our game logic. After we receive this data, we switch addresses again, and the cycle repeats continuously.

In our ADC module, we also translate the eight bit digital signals from the ADC into three one bit signals to feed into the game itself. These signals are turn left and turn right for the rocket, and shoot for the bullets. Depending on what our address is set as, we can determine whether we should translate the eight bit signal into data for the rocket turn signals or the bullet shooting signal.

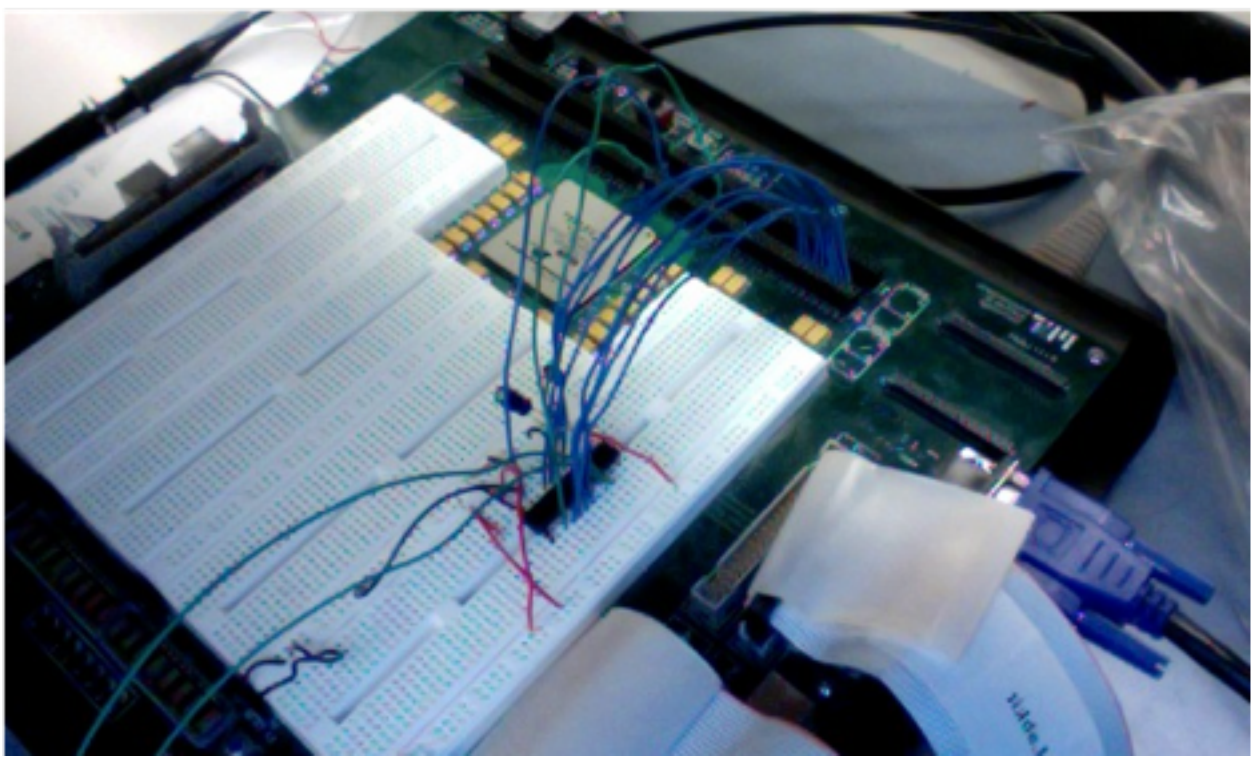

Figure 2. The ADC chip mounted on the 6.111 labkit. The eight blue wires clumped together that go into the user pins are the digital output from the chip, and the two green wires coming in from outside the picture are the two analog inputs from the accelerometer.

2.3 Graphics (Daniel)

The graphics in our project run at 40MHz, for a resolution of 800x600. Most objects are stored in a shape table (shape_table.v) as a list of line segments. This allows them to be easily translated and rotated (by the shape module) to the proper position on the screen.

The correctly-positioned line segments are passed to the Bresenham module, which implements Bresenham's line drawing algorithm: http://en.wikipedia.org/wiki/Bresenham's_line_algorithm. Each line segment is drawn to a frame buffer in a ZBT RAM ,while a different frame buffer is being displayed on the screen. Once all the drawing is complete, on the rising edge of the vsync signal, the roles of the buffers are swapped. This behavior (implemented in frame_buffers.v) produces a glitch-free display and allows us plenty of time to draw the game objects.

We used a few other minor modules, including the ramclock module (to keep the two ZBTs synchronized) and the SVGA module (to control the timing of the video output signals).

2.3.1 Ship (Daniel)

The ship is basically a triangle in the middle of the screen. Its line segments are stored in the shape_table.v file so it can be rotated according to the game's inputs. Its x and y coordinates, and its angle, are kept in BRAMs as part of its 32-bit entity data.

2.3.2 Asteroids (Daniel)

Similar to the ship, asteroids are stored as a list of line segments in the shape table. This way, we can rotate the asteroids to random orientations to make sure they don't all look exactly the same. All asteroids start out as "spawn" objects, which are not displayed on the screen or considered in collision calculations. This allows us to limit the number of asteroids on the screen while also giving the player a chance to destroy them all before they reappear.

2.3.3 Bullets (Anna)

In our game, we define a maximum of five bullets on screen at a time. Given the speed of the asteroids and the speed at which a person can maneuver the accelerometer, this was a reasonable approximation to make. Each bullet is a two-pixel by two-pixel sprite; that is, the 24-bit pixel output is defined by the x and y coordinates and the size of the bullet. Each bullet is represented by one instance of the bullet module. In the bullet module, if we have received a shoot signal from the converted accelerometer signal, a bullet is created in the middle of the screen (the location of the spaceship) and begins to move linearly in a given direction. This direction is determined by the orientation of the spaceship, given as an eight bit input that represents the angle. We use a trigonometry module to determine the sine and cosine of the direction and calculate which direction the bullet should travel in. With each cycle, the bullet increments its movement. Once the bullet reaches the edge of the screen, it ceases to exist.

The bullet graphics are created in the same fashion the graphics are made in Lab 3, in which the xvga module counts the horizontal and vertical pixels on the screen and defines each pixel as some value that corresponds to a certain color.

Because input signals from the accelerometer are not always clean, we also keep track of how far along each bullet is in its trajectory. Only after a bullet has traveled a certain distance can a new bullet be created, to avoid several bullets firing at the same time and overlapping.

The bullets originally used an original trigonometry module separate from the one used by the spaceship. In the final implementation of the game the trig module used by the spaceship was used by the bullet module as well.

2.4 Game Logic (Daniel)

In the game, we check for the reset button and the number of lives lost (how many times an asteroid has hit the ship) to reset the game.

2.5 Collisions

Because there objects on screen were created differently, we created two different collision modules to handle different kinds of objects.

2.5.1 With Sprites (Anna)

Because the bullets are largely defined by their x and y coordinates, handling collisions based on x and y coordinate overlap between objects was the more straightforward, if not elegant, approach. Since there is a limited number of bullets and asteroids, we can blunt force our way through the collision detection by checking for each object if there is an overlap regarding location and size. Because the ship is always in the middle of the screen, the collision module detects asteroid-ship collisions simply by checking if an asteroid has approached the middle of the screen. For each object, there is a hit signal that goes to 1 if a collision occurs and 0 otherwise.

2.5.2 With Vectors (Daniel)

Because all the objects in the shape table have x and y coordinates and a hard-coded radius, collision detection can be performed by comparing the square of the radii with the square of the distance between objects. The collision module (collision.v) is a state machine that takes in pairs of objects and performs the necessary calculations on each one in turn before raising its ready signal for the next object pair.

2.6 Sound (Anna)

A sound output signals every instance of a bullet being fired or a collision occurrence. We picked out two WAV audio clips, one for bullets and one for collisions, and altered the sample rate such that it would be compatible with the 48kHz sample rate of the labkit's AC'97 audio codec. We then used the MATLAB script provided by former 6.111 student Yuta Kuboyama that converts WAV files into COE files. Each converted sound file is saved as a single port ROM. We also use the code supplied in Lab 5 that provides a ready signal from the AC'97 that we synchronize our module with.

For each ready signal, if the game has experienced a collision or bullet firing, we increment the address of the collision or bullet sound memory until it reaches the maximum memory. For each of these addresses, there is a different data output that we retrieve from the ROM. The most significant eight bits of this data is sent to the AC'97, which in turn creates a sound that can be heard on the speaker plugged into the labkit.

3 Testing and Debugging

3.1 Input (Anna)

To make sure we were actually receiving signals from the accelerometer, we hooked up an oscilloscope to each of the outputs and measured the voltage changes. Once we knew the signals going from the accelerometer to the ADC were useful, we implemented a hex display that showed us which state the ADC module was in, determining at which point we were getting stuck due to a certain signal. We also used the hex display to output what the digital signal was for a given accelerometer orientation, which we could then extrapolate from and use for when we were sending signals to the game logic modules.

We also found that simply re-reading the data sheet helped solve many problems, as the ADC requires to run on a 100kHz clock or slower, while we were previously running on a 27MHz clock, and thus not allowing the ADC to generate the proper signals.

We implemented a mock game in which we had a fake ship, represented as a square, similar to the paddle in the Pong Game of Lab 3. Instead of rotating like the final ship would, this fake ship simply translates to the left or right. By hooking up the accelerometer to the fake ship, we could determine if we could in fact control an object on screen with the accelerometer.

3.2 Ship and Asteroids (Daniel)

Once the frame-buffered graphics were working properly, creating the ship and the asteroids as objects in the shape table was simple. The difficult part was getting them to move properly within the movement module (movement.v). There was some trial and error involved here, and

3.3 Bullets (Anna)

Initially, rather than using the accelerometer to signal the bullets, we used a debounced button on the labkit to set the bullets in motion. This avoided any potential issues with the accelerometer not delivering a clean signal and confusing the results. We used the hex display to output the coordinates of the bullet and the LEDs to indicate whether or not the bullets were in existence once signaled to begin firing. To control the direction, we used the eight switches on the labkit to represent the direction of the ship and subsequently the direction in which the bullets should travel.

Once it was determined the bullets were moving when they should, we hooked up the converted accelerometer input to the bullet module. This resulted in the issue of several bullets firing at once, because the accelerometer didn't produce a clean pulse like the button did. To overcome this, we attempted implementing various filtering techniques, until we settled in on monitoring the existing bullet's trajectory and ensuring a certain distance has been traveled before firing a new bullet.

We expanded on the mock game from section 4.1 by adding the bullets to see how the image looked on screen as opposed to the hex display. Through this, we finalized a speed that seemed reasonable for the bullets to travel, and also were able to verify that the bullets were indeed traveling in the correct direction for a given angle input, which was not as intuitive to see based on the hex display.

We also used this mock game to test the sprite-based collision module. We created five fake asteroids, represented as squares that bounced across the screen similar to the puck in the Pong Game. From here we could make sure the bullets were actually being destroyed when they collided with the asteroids, and while we were at it we could also make sure the asteroids and ship were being destroyed accordingly as well.

3.4 Sound (Anna)

We used two buttons on the labkit, one for collisions and one for shooting, to signal the sound module for debugging. The main thing we needed to check in the sound module was that the proper data was being sent for each memory address. To ensure this, we used the hex display to display the data for given sets of memory addresses. If the hex display values matched the values in the COE file, then we knew the proper data was being sent to the AC'97 and speaker.

4 Challenges

While there were several tricky bugs for each individual module, the most challenging part of the project was combining all of them together. Particularly, because the bullets were sprites while the ship and asteroids were vectors, the collision detection between them proved to be more difficult than originally anticipated.

Similarly, when working individually we were each using different clock frequencies and video dimensions, which meant that rather than simply adding all the modules together, we had to alter the clock and video signals for several of the modules so that they would be compatible.

5 Conclusion

Our game takes in an analog to digital converted signal and, through a series of states within the game logic, displays a game similar to that of the original Asteroids. While it is slightly more simplistic in that points are not scored and the ship only rotates, there are added features such as a handheld user control and sound output, and now that we have the basis of the game set, there is much room for expansion for this game.

Appendix: Verilog Code

A Analog to Digital Converter (ADC)

```
module adc(clock, data_bus, adc_rdy,
     address, state, adc_rd, turn_right, turn_left, shoot,
dig_data_rl, dig_data_shoot);

     parameter S_ONE = 2'b00;
     parameter S_WAIT_ADDR = 2'b01;
     parameter S_DATA_VALID = 2'b10;

     input clock;
     input [7:0] data_bus;
     input adc_rdy;

     output reg address = 1'b0;
     output reg [1:0] state = S_ONE;
     reg [1:0] next_state = S_ONE;
     output reg adc_rd = 1'b1;
     output reg turn_right = 0;
     output reg turn_left = 0;
     output reg shoot = 0;
     output reg [7:0] dig_data_rl = 0;
       output reg [7:0] dig_data_shoot = 0;


     always @(posedge clock) begin
          state <= next_state; //updating state every clock tick
     end

     always @(*) begin

          case(state)
           S_ONE: begin
               address = ~address;
               next_state = S_WAIT_ADDR;
           end
           S_WAIT_ADDR: begin
               adc_rd = 1'b0;
               if (adc_rdy == 1'b1) next_state = S_DATA_VALID;
               else next_state = S_WAIT_ADDR;
           end
           S_DATA_VALID: begin
```

```
                if (address == 1'b0) begin dig_data_rl =
data_bus; end
                else begin dig_data_shoot = data_bus; end
                adc_rd = 1'b1;
                next_state = S_ONE;
           end
           default: begin
                next_state = S_ONE;
                adc_rd = 1'b1;
          end
          endcase

 //spaceship
          if (dig_data_rl < 8'b01001000) begin
                turn_right = 1'b1;
                turn_left = 1'b0;
          end
          else if (dig_data_rl > 8'b01100010) begin
                turn_left = 1'b1;
                turn_right = 1'b0;
          end
          else begin turn_right = 1'b0; turn_left = 1'b0; end

//shoot
          if (dig_data_shoot > 8'b01100000) shoot = 1'b1;
          else shoot = 1'b0;

     end

endmodule
```

B Shooter

```
module shooter (input adc_shoot,
     input clock,
     input reset,
     input [10:0] hcount,// horizontal index of current pixel
     input [9:0]    vcount, // vertical index of current pixel
     input hsync,        // horizontal sync signal
     input vsync,
     input blank,        // blanking
     input [7:0] dir, //360 degrees
     input game_over,
     input collision,
     input [10:0] x_start, //debug
     input [10:0] y_start,

     output phsync, // pong game's horizontal sync
     output pvsync, // pong game's vertical sync
     output pblank, // pong game's blanking
     output [23:0] pixel,
     output reg bullet_existence,
     output reg [10:0] x_coord,
     output reg [10:0] y_coord,
     output reg [13:0] xdir,
     output reg [13:0] ydir,
     output [10:0] sine,
     output reg bullet_far
     );

     parameter B_DIMEN = 2'b10; //bullet width and height
     parameter XMIDDLE = 11'd512;
     parameter YMIDDLE = 10'd383;
     parameter XMAX = 11'd1023; //max dimensions of screen
     parameter YMAX = 10'd767;
     parameter SPEED = 3'b011; //bullet speed

     reg signed [10:0] bulletX = XMIDDLE;
     //starting ball in middle of screen
     reg signed [10:0] bulletY = YMIDDLE;
     reg signed [13:0] x_dir, y_dir;
     reg [7:0] orig_dir;

     wire signed [10:0] sin, cos;
```

```
	trig gettrig(.clock(clock), .dir(dir), .sin(sin), .cos
(cos));
	assign sine = sin;

	reg bullet_exists = 1'b0;
	reg [3:0] x_dimen = 4'b0000;
	reg [3:0] y_dimen = 4'b0000;

	wire [23:0] bullet_pixel;

reg [2:0] delay_counter = 3'b000;
reg [3:0] bullet_counter = 0;

	always @(posedge clock) begin
		bullet_existence <= bullet_exists;
		x_coord <= bulletX; y_coord <= bulletY;
		xdir <= x_dir; ydir <= y_dir;
		if (reset || collision || game_over) begin
		//when reset button is pressed or collision occurs
			bullet_exists <= 1'b0;
			bulletX <= 0;
			bulletY <= 0;
		end
		if (bullet_exists == 0) begin
			bullet_far <= 0;
			bulletX <= 0; //debug
			bulletY <= 0;
		end
		if (adc_shoot && game_over == 1'b0 && bullet_exists ==
0) begin
			bullet_exists <= 1'b1;
			orig_dir <= dir;
			x_dir <= (SPEED*cos)/1024;
			y_dir <= (SPEED*sin)/1024;
			bullet_counter <= 0;
			bulletX <= x_start + 6'd32;
			bulletY <= YMIDDLE;
		end
		if (bullet_exists && hcount == XMAX && vcount ==
YMAX ) begin
			if (delay_counter == 3'b110) begin
				if (bullet_counter >= 3'b011) begin
					bullet_far <= 1;
					bullet_counter <= bullet_counter;
				       end
				else bullet_counter <= bullet_counter + 1;
```

```
                    delay_counter <= 0;
                    if (orig_dir < 9'd64) begin //right up
                    bulletX <= bulletX + x_dir;
                     bulletY <= bulletY - y_dir;
                    end
                    else if (orig_dir < 9'd128) begin //left up
                     bulletX <= bulletX - x_dir;
                     bulletY <= bulletY - y_dir;
                    end
                    else if (orig_dir < 9'd192) begin
                    //left down
                     bulletX <= bulletX - x_dir;
                     bulletY <= bulletY + y_dir;
                    end
                    else begin //right down
                     bulletX <= bulletX + x_dir;
                     bulletY <= bulletY + y_dir;
                    end
                    if (((bulletX + 2*SPEED) >= XMAX) ||
                    //hit right wall
                         (bulletX <= 2*SPEED) || //left wall
                         ((bulletY + 2*SPEED) >= YMAX) ||
                         //bottom wall
                         (bulletY <= 2*SPEED)) begin //top wall
                         bullet_exists <= 1'b0;
                         bulletX <= XMIDDLE;
                         bulletY <= YMIDDLE;
                    end
               end
               else delay_counter <= delay_counter + 1;
          end

     end


//blob is taken directly from Lab 3
     blob #(.WIDTH(B_DIMEN),.HEIGHT(B_DIMEN),.COLOR
(24'hFF_FF_00))
               bulletpixel(.x(bulletX), .y(bulletY), .hcount
(hcount), .vcount(vcount),
                    .pixel(bullet_pixel));

     assign pixel = bullet_pixel;
     assign phsync = hsync;
      assign pvsync = vsync;
      assign pblank = blank;
```

```
endmodule


//Note: this trig module was not used in the final version of
the game,
//only in the initial debugging process for the bullet

module trig_table (clock, dir, sin, cos);
     input clock;
     input [7:0] dir;
     output reg signed [10:0] sin, cos;

     always @(posedge clock) begin
     if (dir < 8'd7) begin sin <= 11'b00010110010; cos <=
11'b11111111000; end //., .
     else if (dir < 8'd14) begin sin <= 11'b01000010010; cos <=
11'b11110111010; end //.,
     else if (dir < 8'd21) begin sin <= 11'b01101100001; cos <=
11'b11101000000; end //., .
     else if (dir < 8'd28) begin sin <= 11'b10010010110; cos <=
11'b11010001101; end //., .
     else if (dir < 8'd35) begin sin <= 11'b10110101000; cos <=
11'b10110101000; end //., .
     else if (dir < 8'd43) begin sin <= 11'b11010001101; cos <=
11'b10010010110; end //., .
     else if (dir < 8'd50) begin sin <= 11'b11101000000; cos <=
11'b01101100001; end //., .
     else if (dir < 8'd57) begin sin <= 11'b11110111010; cos <=
11'b00100001001; end //., .
     else if (dir < 8'd64) begin sin <= 11'b11111111000; cos <=
11'b00010110010; end //., .
     else if (dir < 8'd71) begin sin <= 11'b11111111000; cos <=
11'b0001011001; end //., -.
     else if (dir < 8'd78) begin sin <= 11'b11110111010; cos <=
11'b0100001001; end //., -.
     else if (dir < 8'd85) begin sin <= 11'b11101000000; cos <=
11'b01101100001; end //., -.
     else if (dir < 8'd92) begin sin <= 11'b11010001101; cos <=
11'b10010010110; end //., -.
     else if (dir < 8'd97) begin sin <= 11'b10110101000; cos <=
11'b10110101000; end //., -.
     else if (dir < 8'd106) begin sin <= 11'b10010010110; cos <=
11'b11010001101; end //., -.
     else if (dir < 8'd113) begin sin <= 11'b01101100001; cos <=
11'b11101000000; end //., -.
```

```
     else if (dir < 8'd120) begin sin <= 11'b01000010010; cos <=
11'b11110111010; end //., -.
     else if (dir < 8'd128) begin sin <= 11'b00010110010; cos <=
11'b11111111000; end //., -.
     else if (dir < 8'd135) begin sin <= 11'b00010110010; cos <=
11'b11111111000; end //-., -.
     else if (dir < 8'd142) begin sin <= 11'b01000010010; cos <=
11'b11110111010; end //-., -.
     else if (dir < 8'd149) begin sin <= 11'b01101100001; cos <=
11'b11101000000; end //-., -.
     else if (dir < 8'd156) begin sin <= 11'b10010010110; cos <=
11'b11010001101; end //-., -.
     else if (dir < 8'd164) begin sin <= 11'b10110101000; cos <=
11'b10110101000; end //-., -.
     else if (dir < 8'd170) begin sin <= 11'b11010001101; cos <=
11'b10010010110; end //-., -.
     else if (dir < 8'd177) begin sin <= 11'b11101000000; cos <=
11'b01101100001; end //-., -.
     else if (dir < 8'd184) begin sin <= 11'b11110111010; cos <=
11'b01000010010; end //-., -.
     else if (dir < 8'd191) begin sin <= 11'b11111111000; cos <=
11'b00010110010; end //-., -.
     else if (dir < 8'd198) begin sin <= 11'b11111111000; cos <=
11'b00010110010; end //-., .
     else if (dir < 8'd205) begin sin <= 11'b11110111010; cos <=
11'b01000010010; end //-., .
     else if (dir < 8'd212) begin sin <= 11'b11101000000; cos <=
11'b01101100001; end //-., .
     else if (dir < 8'd220) begin sin <= 11'b11010001101; cos <=
11'b10110101000; end //-., .
     else if (dir < 8'd227) begin sin <= 11'b10110101000; cos <=
11'b10110101000; end //-., .
     else if (dir < 8'd235) begin sin <= 11'b10010010110; cos <=
11'b11010001101; end //-., .
     else if (dir < 8'd242) begin sin <= 11'b01101100001; cos <=
11'b11101000000; end //-., .
     else if (dir < 8'd250) begin sin <= 11'b01000010010; cos <=
11'b11110111010; end
     else begin sin <= 11'b00010110010; cos <= 11'b11111111000;
end
     end
endmodule
```

```
//
//This code goes into the top-level labkit.v
//Each adc_shooter and b_e is from one of the five instances of
the shooter module
//

always @(posedge clock_65mhz) begin
     adc_delay <= adc_shoot; //adc_shoot shoot_button
     if (adc_shoot == 1 && adc_delay == 0) begin //adc_shoot

          if ((b_e1 == 0) && (~b_e2 || bf2) && (~b_e3 || bf3) &&
(~b_e4 || bf4) && (~b_e5 || bf5)) begin
               adc_shooter1 <= 1;
               adc_shooter2 <= 0;
               adc_shooter3 <= 0;
               adc_shooter4 <= 0;
               adc_shooter5 <= 0;
          end

          else if ((b_e2 == 0) && (~b_e1 || bf1) && (~b_e3 ||
bf3) && (~b_e4 || bf4) && (~b_e5 || bf5)) begin
               adc_shooter2 <= 1;
               adc_shooter1 <= 0;
               adc_shooter3 <= 0;
               adc_shooter4 <= 0;
               adc_shooter5 <= 0;
          end

          else if ((b_e3 == 0) && (~b_e2 || bf2) && (~b_e1 ||
bf1) && (~b_e4 || bf4) && (~b_e5 || bf5)) begin
               adc_shooter3 <= 1;
               adc_shooter1 <= 0;
               adc_shooter2 <= 0;
               adc_shooter4 <= 0;
               adc_shooter5 <= 0;
          end

          else if ((b_e4 == 0) && (~b_e2 || bf2) && (~b_e3 ||
bf3) && (~b_e1 || bf1) && (~b_e5 || bf5)) begin
               adc_shooter4 <= 1;
               adc_shooter1 <= 0;
               adc_shooter2 <= 0;
               adc_shooter3 <= 0;
               adc_shooter5 <= 0;
          end
```

```
          else if ((b_e5 == 0) && (~b_e2 || bf2) && (~b_e3 ||
bf3) && (~b_e4 || bf4) && (~b_e1 || bf1)) begin
               adc_shooter5 <= 1;
               adc_shooter1 <= 0;
               adc_shooter2 <= 0;
               adc_shooter3 <= 0;
               adc_shooter4 <= 0;
          end

     end
     else begin
          adc_shooter1 <= 0;
          adc_shooter2 <= 0;
          adc_shooter3 <= 0;
          adc_shooter4 <= 0;
          adc_shooter5 <= 0;
     end
end
```

C Collision Using X-Y Coordinates

```
module collision(input clock, input [10:0] a1x, a1y, a2x, a2y,
a3x, a3y, a4x, a4y, a5x, a5y,
                    b1x, b1y, b2x, b2y, b3x, b3y, b4x, b4y, b5x,
b5y,
                    sx, input [9:0] sy,
                    output reg h_a1, h_a2, h_a3, h_a4, h_a5,
                    h_b1, h_b2, h_b3, h_b4, h_b5,
                    h_ship);


     parameter SHIP_SIZE = 64;
     parameter ASIZE = 20; //asteroid size

     always @(posedge clock) begin
//asteroids
          if ( ( (b1x >= a1x) && (b1x <= (a1x + ASIZE)) && (b1y
>= a1y) && (b1y <= (a1y + ASIZE))) ||
               ( (b2x >= a1x) && (b2x <= (a1x + ASIZE)) && (b2y
>= a1y) && (b2y <= (a1y + ASIZE))) ||
( (b3x >= a1x) && (b3x <= (a1x + ASIZE)) && (b3y >= a1y) && (b3y
<= (a1y + ASIZE))) ||
( (b4x >= a1x) && (b4x <= (a1x + ASIZE)) && (b4y >= a1y) && (b4y
<= (a1y + ASIZE))) ||
( (b5x >= a1x) && (b5x <= (a1x + ASIZE)) && (b5y >= a1y) && (b5y
<= (a1y + ASIZE))) ) h_a1 <= 1;
          else h_a1 <= 0;

if ( ( (b1x >= a2x) && (b1x <= (a2x + ASIZE)) && (b1y >= a2y-
ASIZE) && (b1y <= (a2y + ASIZE))) ||
               ( (b2x >= a2x) && (b2x <= (a2x + ASIZE)) && (b2y
>= a2y) && (b2y <= (a2y + ASIZE))) ||
( (b3x >= a2x) && (b3x <= (a2x + ASIZE)) && (b3y >= a2y) && (b3y
<= (a2y + ASIZE))) ||
( (b4x >= a2x) && (b4x <= (a2x + ASIZE)) && (b4y >= a2y) && (b4y
<= (a2y + ASIZE))) ||
( (b5x >= a2x) && (b5x <= (a2x + ASIZE)) && (b5y >= a2y) && (b5y
<= (a2y + ASIZE))) ) h_a2 <= 1;
          else h_a2 <= 0;

if ( ( (b1x >= a3x) && (b1x <= (a3x + ASIZE)) && (b1y >= a3y) &&
(b1y <= (a3y + ASIZE))) ||
               ( (b2x >= a3x) && (b2x <= (a3x + ASIZE)) && (b2y
>= a3y) && (b2y <= (a3y + ASIZE))) ||
```

```
( (b3x >= a3x) && (b3x <= (a3x + ASIZE)) && (b3y >= a3y) && (b3y
<= (a3y + ASIZE))) ||
( (b4x >= a3x) && (b4x <= (a3x + ASIZE)) && (b4y >= a3y) && (b4y
<= (a3y + ASIZE))) ||
( (b5x >= a3x) && (b5x <= (a3x + ASIZE)) && (b5y >= a3y) && (b5y
<= (a3y + ASIZE))) ) h_a3 <= 1;
          else h_a3 <= 0;

if ( ( (b1x >= a4x) && (b1x <= (a4x + ASIZE)) && (b1y >= a4y) &&
(b1y <= (a4y + ASIZE))) ||
               ( (b2x >= a4x) && (b2x <= (a4x + ASIZE)) && (b2y
>= a4y) && (b2y <= (a4y + ASIZE))) ||
( (b3x >= a4x) && (b3x <= (a4x + ASIZE)) && (b3y >= a4y) && (b3y
<= (a4y + ASIZE))) ||
( (b4x >= a4x) && (b4x <= (a4x + ASIZE)) && (b4y >= a4y) && (b4y
<= (a4y + ASIZE))) ||
( (b5x >= a4x) && (b5x <= (a4x + ASIZE)) && (b5y >= a4y) && (b5y
<= (a4y + ASIZE))) ) h_a4 <= 1;
          else h_a4 <= 0;

if ( ( (b1x >= a5x) && (b1x <= (a5x + ASIZE)) && (b1y >= a5y) &&
(b1y <= (a5y + ASIZE))) ||
               ( (b2x >= a5x) && (b2x <= (a5x + ASIZE)) && (b2y
>= a5y) && (b2y <= (a5y + ASIZE))) ||
( (b3x >= a5x) && (b3x <= (a5x + ASIZE)) && (b3y >= a5y) && (b3y
<= (a5y + ASIZE))) ||
( (b4x >= a5x) && (b4x <= (a5x + ASIZE)) && (b4y >= a5y) && (b4y
<= (a5y + ASIZE))) ||
( (b5x >= a5x) && (b5x <= (a5x + ASIZE)) && (b5y >= a5y) && (b5y
<= (a5y + ASIZE))) ) h_a5 <= 1;
          else h_a5 <= 0;

//bullets
if ( ( (b1x >= a1x) && (b1x <= (a1x + ASIZE)) && (b1y >= a1y) &&
(b1y <= (a1y + ASIZE))) ||
               ( (b1x >= a2x) && (b1x <= (a2x + ASIZE)) && (b1y
>= a2y) && (b1y <= (a2y + ASIZE))) ||
( (b1x >= a3x) && (b1x <= (a3x + ASIZE)) && (b1y >= a3y) && (b1y
<= (a3y + ASIZE))) ||
( (b1x >= a4x) && (b1x <= (a4x + ASIZE)) && (b1y >= a4y) && (b1y
<= (a4y + ASIZE))) ||
( (b1x >= a5x) && (b1x <= (a5x + ASIZE)) && (b1y >= a5y) && (b1y
<= (a5y + ASIZE))) ) h_b1 <= 1;
          else h_b1 <= 0;
```

```
if ( ( (b2x >= a1x) && (b2x <= (a1x + ASIZE)) && (b2y >= a1y) &&
(b2y <= (a1y + ASIZE))) ||
               ( (b2x >= a2x) && (b2x <= (a2x + ASIZE)) && (b2y
>= a2y) && (b2y <= (a2y + ASIZE))) ||
( (b2x >= a3x) && (b2x <= (a3x + ASIZE)) && (b2y >= a3y) && (b2y
<= (a3y + ASIZE))) ||
( (b2x >= a4x) && (b2x <= (a4x + ASIZE)) && (b2y >= a4y) && (b2y
<= (a4y + ASIZE))) ||
( (b2x >= a5x) && (b2x <= (a5x + ASIZE)) && (b2y >= a5y) && (b2y
<= (a5y + ASIZE))) ) h_b2 <= 1;
          else h_b2 <= 0;

if ( ( (b3x >= a1x) && (b3x <= (a1x + ASIZE)) && (b3y >= a1y) &&
(b3y <= (a1y + ASIZE))) ||
               ( (b3x >= a2x) && (b3x <= (a2x + ASIZE)) && (b3y
>= a2y) && (b3y <= (a2y + ASIZE))) ||
( (b3x >= a3x) && (b3x <= (a3x + ASIZE)) && (b3y >= a3y) && (b3y
<= (a3y + ASIZE))) ||
( (b3x >= a4x) && (b3x <= (a4x + ASIZE)) && (b3y >= a4y) && (b3y
<= (a4y + ASIZE))) ||
( (b3x >= a5x) && (b3x <= (a5x + ASIZE)) && (b3y >= a5y) && (b3y
<= (a5y + ASIZE))) ) h_b3 <= 1;
          else h_b3 <= 0;

if ( ( (b4x >= a1x) && (b4x <= (a1x + ASIZE)) && (b4y >= a1y) &&
(b4y <= (a1y + ASIZE))) ||
               ( (b4x >= a2x) && (b4x <= (a2x + ASIZE)) && (b4y
>= a2y) && (b4y <= (a2y + ASIZE))) ||
( (b4x >= a3x) && (b4x <= (a3x + ASIZE)) && (b4y >= a3y) && (b4y
<= (a3y + ASIZE))) ||
( (b4x >= a4x) && (b4x <= (a4x + ASIZE)) && (b4y >= a4y) && (b4y
<= (a4y + ASIZE))) ||
( (b4x >= a5x) && (b4x <= (a5x + ASIZE)) && (b4y >= a5y) && (b4y
<= (a5y + ASIZE))) ) h_b4 <= 1;
          else h_b4 <= 0;

if ( ( (b5x >= a1x) && (b5x <= (a1x + ASIZE)) && (b5y >= a1y) &&
(b5y <= (a1y + ASIZE))) ||
               ( (b5x >= a2x) && (b5x <= (a2x + ASIZE)) && (b5y
>= a2y) && (b5y <= (a2y + ASIZE))) ||
( (b5x >= a3x) && (b5x <= (a3x + ASIZE)) && (b5y >= a3y) && (b5y
<= (a3y + ASIZE))) ||
( (b5x >= a4x) && (b5x <= (a4x + ASIZE)) && (b5y >= a4y) && (b5y
<= (a4y + ASIZE))) ||
( (b5x >= a5x) && (b5x <= (a5x + ASIZE)) && (b5y >= a5y) && (b5y
<= (a5y + ASIZE))) ) h_b5 <= 1;
```

```
          else h_b5 <= 0;


//ship

if ( ((sx >= (a1x - SHIP_SIZE)) && (sx <= (a1x + ASIZE)) && (sy
>= (a1y - SHIP_SIZE)) && (sy <= (a1y + ASIZE))) ||
 ((sx >= (a2x - SHIP_SIZE)) && (sx <= (a2x + ASIZE)) && (sy >=
(a2y - SHIP_SIZE)) && (sy <= (a2y + ASIZE))) ||
((sx >= (a3x - SHIP_SIZE)) && (sx <= (a3x + ASIZE)) && (sy >=
(a3y - SHIP_SIZE)) && (sy <= (a3y + ASIZE))) ||
((sx >= (a4x - SHIP_SIZE)) && (sx <= (a4x + ASIZE)) && (sy >=
(a4y - SHIP_SIZE)) && (sy <= (a4y + ASIZE))) ||
((sx >= (a5x - SHIP_SIZE)) && (sx <= (a5x + ASIZE)) && (sy >=
(a5y - SHIP_SIZE)) && (sy <= (a5y + ASIZE))) ) begin
     h_ship <= 1;
     end
          else begin h_ship <= 0; end


     end

endmodule
```

D Sound Output

```
//The ready input comes lab5audio, taken directly from Lab 5
module sound_output(input clock, shoot, collision, ready,
     output reg [7:0] ac97_noise);

     parameter ON = 1'b1;
     parameter OFF = 1'b0;
     parameter SHOOT_MEM = 'd12000;
     parameter COLL_MEM = 'd37000;

     reg [15:0] shoot_addr = 0;
     reg [15:0] collision_addr = 0;
     wire [15:0] shoot_dout;
     wire [15:0] collision_dout;

     fire_sound shoot_fire(.addra(shoot_addr),.clka
(clock),.douta(shoot_dout));
     bang_sound collision_bang(.addra(collision_addr), .clka
(clock), .douta(collision_dout));

     reg count_coll_addr = 0;
     reg count_shoot_addr = 0;


     always @(posedge clock) begin
          if(collision) count_coll_addr <= 1;
          else if (shoot) count_shoot_addr <= 1;
          if(ready) begin
               if (count_coll_addr) begin
                    if (collision_addr < (COLL_MEM-1) ) begin
                         collision_addr <= collision_addr + 1;
                    end
                    else begin
                         collision_addr <= 0;
                         count_coll_addr <= 0;
                    end
               end
               if (count_shoot_addr) begin
                    if (shoot_addr < (SHOOT_MEM -1)) begin
                         shoot_addr <= shoot_addr + 1;
                    end
                    else begin
                         shoot_addr <= 0;
                         count_shoot_addr <= 0;
```

```
                    end
               end
          end
     end

     always @(*) begin
          if (count_coll_addr == 1) begin
               ac97_noise = collision_dout[15:8];
          end
          else if (count_shoot_addr == 1) begin
               ac97_noise = shoot_dout[15:8];
          end
          else begin ac97_noise = 8'b0; end
     end


endmodule
```

E Fake Ship

```
//This was only used for debugging
module fakeship (input clock, right, left, collision,
               hsync, vsync, blank, reset,
               input [10:0] hcount,
               input [9:0] vcount,
               output phsync, pvsync, pblank,
               output [23:0] pixel,
               output reg [10:0] x_coord,
               output reg [10:0] y_coord);

     parameter XWIDTH = 64; //for ship
     parameter YHEIGHT = 64;
     parameter XMIDDLE = 11'd490;
     parameter YMIDDLE = 10'd352;
     parameter XMAX = 11'd1023; //max dimensions of screen
     parameter YMAX = 10'd767;

     reg [10:0] shipX = XMIDDLE; //starting in middle of screen
     reg [10:0] shipY = YMIDDLE;

     wire [23:0] ship_pixel;

     always @(posedge clock) begin
          x_coord <= shipX;
          y_coord <= shipY;
          if (collision || reset) shipX <= XMIDDLE;
          if (hcount == XMAX && vcount == YMAX) begin
               if (right && ((shipX + 10'd4 + XWIDTH) <
(XMAX-1))) shipX <= shipX + 10'd4;
               else if (left && (shipX > 10'd8)) shipX <= shipX
- 10'd4;
               else shipX <= shipX;
          end
     end

//blob taken from Lab 3
     bloob #(.WIDTH(XWIDTH),.HEIGHT(YHEIGHT),.COLOR
(24'hFF_FF_FF))
               shippixel(.x(shipX),.y(shipY),.hcount
(hcount),.vcount(vcount),
          .pixel(ship_pixel));

     assign pixel = ship_pixel;
```

```
     assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;


endmodule
```

F Ship

```
module bresenham (

input vclock,
input reset,
input vsync,
input shape_ready,
input [19:0] v0,
input [19:0] v1,
input [31:0] rgba,

output reg bresenham_ready,
output reg [10:0] write_x,
output reg [9:0] write_y,
output reg write_enable,
output reg [31:0] write_rgba);

reg steep;
reg signed [10:0] delta_x;
reg signed [10:0] delta_y;
reg signed [10:0] error;
reg signed [1:0] ystep;
reg signed [10:0] x0, x1, y0, y1, x, y;
reg [2:0] state;

parameter STATE_WAITING = 0;
parameter STATE_SETUP_1 = 1;
parameter STATE_SETUP_2 = 2;
parameter STATE_SETUP_3 = 3;
parameter STATE_DRAWING = 4;

  always @(posedge vclock) begin
    if (reset || !vsync) begin
      write_x <= 0;
      write_y <= 0;
      delta_x <= 0;
      delta_y <= 0;
      error <= 0;
      ystep <= 1;
      write_enable <= 0;
      write_rgba <= 0;
      steep <= 0;
```

```
    state <= STATE_WAITING;
    bresenham_ready <= 1;
  end
  else if (shape_ready && state == STATE_WAITING) begin
    bresenham_ready <= 0;
    x0 <= v0[19:10];
    y0 <= v0[9:0];
    x1 <= v1[19:10];
    y1 <= v1[9:0];
    write_rgba <= rgba;
    write_enable <= 0;
    state <= STATE_SETUP_1; // setup stage 1
  end
  else if (state == STATE_SETUP_1) begin
    if (y1 > y0) begin
      if (x1 > x0) begin
        steep <= y1 - y0 > x1 - x0;
      end
      else begin
        steep <= y1 - y0 > x0 - x1;
      end
    end
    else begin
      if (x1 > x0) begin
        steep <= y0 - y1 > x1 - x0;
      end
      else begin
        steep <= y0 - y1 > x0 - x1;
      end
    end
    state <= STATE_SETUP_2; // setup stage 2
  end
  else if (state == STATE_SETUP_2) begin
    if (steep) begin
      if (y0 > y1) begin
        x0 <= y1;
        x1 <= y0;
        y0 <= x1;
        y1 <= x0;
        delta_x <= y0 - y1;
        delta_y <= (x1 > x0? x1-x0 : x0-x1);
      end
      else begin
        x0 <= y0;
        x1 <= y1;
        y0 <= x0;
```

```
          y1 <= x1;
          delta_x <= y1 - y0;
          delta_y <= (x1 > x0? x1-x0 : x0-x1);
        end
      end
      else begin
        if (x0 > x1) begin
          x0 <= x1;
          x1 <= x0;
          y0 <= y1;
          y1 <= y0;
          delta_x <= x0-x1;
          delta_y <= (y1 > y0? y1-y0 : y0-y1);
        end
        else begin
          delta_x <= x1-x0;
          delta_y <= (y1 > y0? y1-y0 : y0-y1);
        end
      end
      state <= STATE_SETUP_3; // setup stage 3
    end
    else if (state == STATE_SETUP_3) begin
      error <= {delta_x[10],delta_x[10:1]}; // equivalent to
delta_x / 2 for a signed value
      ystep <= (y0 < y1 ? 1 : -1);
      x <= x0;
      y <= y0;
      state <= STATE_DRAWING; // draw line
    end
    else if (state == STATE_DRAWING) begin
      write_enable <= 1;
      if (steep) begin
        write_x <= y[9:0];
        write_y <= x[9:0];
      end
      else begin
        write_x <= x[9:0];
        write_y <= y[9:0];
      end
      x <= x + 1;
      if (error - delta_y < 0) begin
        error <= error - delta_y + delta_x;
        y <= y + ystep;
      end
      else begin
        error <= error - delta_y;
```

```
      end
      if (x == x1) begin
        state <= STATE_WAITING; // line is finished; go back to
wait state
        bresenham_ready <= 1;
      end
    end
  end
endmodule
```

G Collision Module With Vectors

```
module collision_module(
  input clock, reset,
  input [31:0] to_collision_entry,
  input to_collision_ready,
  output reg from_collision_received, from_collision_done
  );

  reg [31:0] primary_entry, secondary_entry;
  reg [2:0] state;
  parameter S_RECEIVING = 3'b000;
  parameter S_PROCESSING = 3'b001;
  parameter S_IDLE = 3'b010;
  parameter S_DIFFERENCE = 3'b011;
  parameter S_SQUARE = 3'b100;
  parameter S_COMPARE = 3'b101;
  parameter S_PAUSE = 3'b110;
  parameter NO_ID = 4'h0;
  parameter AVATAR = 4'h1;
  parameter BULLET = 4'h2;
  parameter ASTEROID = 4'h3;
  parameter EXPLOSION_0 = 4'h6;
  parameter EXPLOSION_1 = 4'h7;
  wire [3:0] primary_id = primary_entry[31:28];
  wire [9:0] primary_x = primary_entry[27:18];
  wire [9:0] primary_y = primary_entry[17:8];
  wire [3:0] secondary_id = secondary_entry[31:28];
  wire [9:0] secondary_x = secondary_entry[27:18];
  wire [9:0] secondary_y = secondary_entry[17:8];
  reg signed [10:0] x_difference, y_difference;
  reg [23:0] x_squared, y_squared;
  reg [12:0] primary_radius_sq, secondary_radius_sq;

  always @(posedge clock) begin
    if (reset) begin
      primary_entry <= 0;
      secondary_entry <= 0;
      from_collision_received <= 0;
      from_collision_done <= 0;
      state <= S_IDLE;
    end
    else if (to_collision_ready) begin
      primary_entry <= to_collision_entry;
```

```
      from_collision_received <= 0;
      from_collision_done <= 0;
      state <= S_RECEIVING;
    end
    else begin
      if (state == S_RECEIVING) begin
        secondary_entry <= to_collision_entry;
        from_collision_received <= 1;
        from_collision_done <= 0;
        state <= S_PROCESSING;
      end
      else if (state == S_PROCESSING) begin
        if (primary_id == NO_ID) begin //Don't waste time
colliding empty objects
          from_collision_received <= 0;
          from_collision_done <= 1;
          state <= S_IDLE;
        end
        else if (secondary_id == NO_ID) begin //Don't waste on
this secondary, but check others
          from_collision_received <= 0;
          from_collision_done <= 0;
          state <= S_PAUSE;
        end
        else begin
          from_collision_received <= 0;
          from_collision_done <= 0;
          state <= S_DIFFERENCE;
        end
      end
      else if (state == S_PAUSE) begin
        from_collision_received <= 0;
        from_collision_done <= 0;
        state <= S_RECEIVING;
      end
      else if (state == S_DIFFERENCE) begin
        from_collision_received <= 0;
        from_collision_done <= 0;
        x_difference = primary_x - secondary_x;
        y_difference = primary_y - secondary_y;
        state <= S_SQUARE;
      end
      else if (state == S_SQUARE) begin
        from_collision_received <= 0;
        from_collision_done <= 0;
        x_squared = x_difference * x_difference;
```

```
        y_squared = y_difference * y_difference;
        state <= S_COMPARE;
      end
      else if (state == S_COMPARE) begin
        if (x_squared + y_squared > primary_radius_sq +
secondary_radius_sq) begin
          from_collision_received <= 0;
          from_collision_done <= 0;
          state <= S_RECEIVING;
        end
        else begin
          from_collision_received <= 0;
          from_collision_done <= 1;
          state <= S_IDLE;
        end
      end
      else if (state == S_IDLE) begin
        from_collision_received <= 0;
        from_collision_done <= 0;
        state <= S_IDLE;
      end
      else begin
        state <= S_PROCESSING;
      end
    end
  end

  always @(primary_id) begin
    case(primary_id)
      AVATAR: primary_radius_sq = 256;
      BULLET: primary_radius_sq = 64;
      default: primary_radius_sq = 64;
    endcase
  end

  always @(secondary_id) begin
    case(secondary_id)
      ASTEROID: secondary_radius_sq = 256;
      default: secondary_radius_sq = 256;
    endcase
  end
endmodule
```

H Frame Buffers

```
module frame_buffers (

input vclock,  // 40MHz clock
input reset,         // 1 to initialize module
input [10:0] hcount, // horizontal index of current pixel
(0..1023)
input [9:0]  vcount, // vertical index of current pixel (0..767)
input hsync,      // SVGA horizontal sync signal (active low)
input vsync,      // SVGA vertical sync signal (active low)
input blank,      // SVGA blanking (1 means output black pixel)

input [10:0] write_x,
input [9:0] write_y,
input [31:0] write_rgba,
input write_enable,

output reg [18:0] addr0,
output reg [18:0] addr1,
output reg [35:0] write_data0,
output reg [35:0] write_data1,
output reg we_0,
output reg we_1,

input [35:0] read_data0,
input [35:0] read_data1,
input write_buf_switch,

output vhsync, // asteroids horizontal sync
output vvsync, // asteroids vertical sync
output vblank, // asteroids blanking
output reg [31:0] pixel // asteroids pixel
);

  parameter DELAY = 2;

  reg erase_cycle;
  reg write_buf_select;
  reg [DELAY:0] hsync_delay, vsync_delay, blank_delay;

  wire [18:0] sync_addr;
  wire [18:0] drawing_addr;
```

```
assign sync_addr = (800 * vcount) + hcount;
assign drawing_addr = (800 * write_y) + write_x;

assign vhsync = hsync_delay[0];
assign vvsync = vsync_delay[0];
assign vblank = blank_delay[0];

always @(posedge vclock) begin

  hsync_delay <= (reset? 0 : {hsync,hsync_delay[DELAY:1]});
  vsync_delay <= (reset? 0 : {vsync,vsync_delay[DELAY:1]});
  blank_delay <= (reset? 0 : {blank,blank_delay[DELAY:1]});

  if (reset) begin
    erase_cycle <= 1;
    write_buf_select <= 0;
    we_0 <= 0;
    we_1 <= 0;
    addr0 <= 0;
    addr1 <= 0;
    write_data0 <= 0;
    write_data1 <= 0;
    pixel <= 0;
  end
  else begin
    if (vsync_delay[DELAY] && !vsync) begin
      // on a negative edge transition in vsync, either:
      if (erase_cycle) begin
        // enter the draw cycle
        erase_cycle <= 0;
      end
      else begin
        // or swap buffers and enter the erase cycle
        erase_cycle <= 1;
        write_buf_select <= !write_buf_select;
      end
    end
    if (write_buf_select == 0) begin
      if (erase_cycle) begin
        addr0 <= sync_addr;
        write_data0 <= 36'h000000000;
        we_0 <= (hcount < 800 && vcount < 600);
      end
      else begin
        addr0 <= drawing_addr;
```

```
          write_data0 <= write_rgba;
          we_0 <= write_enable;
        end
        we_1 <= 0;
        addr1 <= (hcount < 800 && vcount < 600? sync_addr : 0);
        pixel <= read_data1[31:0];
      end
      else begin
        if (erase_cycle) begin
          addr1 <= sync_addr;
          write_data1 <= 36'h000000000;
          we_1 <= (hcount < 800 && vcount < 600);
        end
        else begin
          addr1 <= drawing_addr;
          write_data1 <= write_rgba;
          we_1 <= write_enable;
        end
        we_0 <= 0;
        addr0 <= (hcount < 800 && vcount < 600? sync_addr : 0);
        pixel <= read_data0[31:0];
      end
    end
  end
endmodule
```

I Game Logic

```
module game_logic_module
  (input clock, vsync, reset,
  input [31:0] random,
  output [31:0] to_movement_entry,
  output [31:0] to_movement_private_entry,
  output to_movement_ready,
  input [31:0] from_movement_entry,
  input [31:0] from_movement_private_entry,
  input from_movement_done,
  input [7:0] from_graphics_index,
  output [31:0] to_graphics_entry,
  input from_collision_received, from_collision_done,
  output to_collision_ready,
  output [31:0] to_collision_entry,
  input [7:0] angle,
  input shooting,
  output reg [15:0] score,
  output reg [3:0] state,
  output wire [7:0] addr
  );

  //Internal variables
  reg to_movement_ready_delay_reg;
  reg game_cycle_toggle;
  reg last_vsync;
  reg [31:0] player_data;
  reg [2:0] shot_cycle_counter;
  reg [7:0] working_index;
  reg [1:0] lives;
  reg game_over;

  //State definitions
  parameter S_SPAWNING = 4'b0000;
  parameter S_MOVING = 4'b0001;
  parameter S_COLLIDING = 4'b0010;
  parameter S_COLLISION_PROCESSING = 4'b0011;
  parameter S_DONE = 4'b0100;
  parameter S_RESET = 4'b0101;
  parameter S_PREMOVING = 4'b0110;
  parameter S_DECIDE_SPAWN = 4'b0111;
  parameter S_DECIDE_SHOOT = 4'b1000;
```

```
  parameter S_SHOOTING = 4'b1001;
  parameter S_PRECOLLIDING = 4'b1010;
  parameter S_AVATAR_COLLIDED = 4'b1011;

  //Collision sub states
  reg [2:0] collide_state;
  parameter CS_WAIT_WORKING = 3'b000;
  parameter CS_RECEIVE_WORKING = 3'b001;
  parameter CS_RECEIVE_COLLIDED = 3'b010;
  parameter CS_DESTROY_WORKING = 3'b011;
  parameter CS_DESTROY_COLLIDED = 3'b100;
  parameter CS_CLEANUP = 3'b101;
  parameter CS_FINISH = 3'b110;

  //BRAM parameter declarations
  reg [7:0] addr_reg;
  wire [7:0]  table0_addr;
  wire [7:0]  table1_addr;
  wire table0_we;
  wire table1_we;
  reg we_reg;
  wire we;
  wire [31:0] table0_mem_out;
  wire [31:0] table1_mem_out;
  wire [31:0] public_mem_out,private_mem_out;
  reg [31:0] public_mem_in,private_mem_in;

  //BRAM parameter assignments
  assign table0_addr = game_cycle_toggle ? addr :
from_graphics_index;
  assign table1_addr = game_cycle_toggle ? from_graphics_index :
addr;
  assign table0_we = game_cycle_toggle ? we : 0;
  assign table1_we = game_cycle_toggle ? 0 : we;
  assign to_graphics_entry = game_cycle_toggle ?
table1_mem_out : table0_mem_out;
  assign addr = addr_reg;
  assign we = we_reg;

  //BRAM instantiation
  mybram #(.LOGSIZE(8), .WIDTH(32)) graphics_entity_table0(.addr
(table0_addr),.clk(clock),.we(table0_we),.din
(public_mem_in),.dout(table0_mem_out));
  mybram #(.LOGSIZE(8), .WIDTH(32)) graphics_entity_table1(.addr
(table1_addr),.clk(clock),.we(table1_we),.din
(public_mem_in),.dout(table1_mem_out));
```

```
  mybram #(.LOGSIZE(8), .WIDTH(32)) entity_table(.addr
(addr),.clk(clock),.we(we),.din(public_mem_in),.dout
(public_mem_out));
  mybram #(.LOGSIZE(8), .WIDTH(32)) private_entity_table(.addr
(addr),.clk(clock),.we(we),.din(private_mem_in),.dout
(private_mem_out));

  //Outputs and associated registers
  reg to_movement_ready_reg, to_collision_ready_reg;
  assign to_movement_entry = public_mem_out;
  assign to_movement_private_entry = private_mem_out;
  assign to_movement_ready = to_movement_ready_reg;
  assign to_collision_entry = public_mem_out;
  assign to_collision_ready = to_collision_ready_reg;

  // multi-shot code
  reg [1:0] shots;
  reg bullet_pause;
  wire [7:0] angle_minus = angle - 8;
  wire [7:0] angle_plus = angle + 8;

  always @(posedge clock) begin
    last_vsync <= vsync;

    //RESET THE GAME
    if (reset) begin
      state <= S_RESET;
      game_cycle_toggle <= 0;
      addr_reg <= 8'b1111_1111; //255
      we_reg <= 0;
      public_mem_in <= 0;
      private_mem_in <= 0;
      to_movement_ready_reg <= 0;
      player_data <= 0;
      shot_cycle_counter <= 0;
      working_index <= 0;
      lives <= 2'b11;
      game_over <= 0;
      score <= 0;
    end
    else begin

      //DETECT IF WE ARE STARTING A NEW GAME CYCLE, trigger on
vsync negedge
```

```
      if (last_vsync && ~vsync) begin
        game_cycle_toggle <= ~game_cycle_toggle;
        working_index <= 0;
        state <= S_DECIDE_SPAWN;
        addr_reg <= 8'b1000_0000;
      end

      //DECIDE SPAWN STATE
      //Checks the random number to see if we should spawn
      else if (state == S_DECIDE_SPAWN) begin
        addr_reg <= 8'b1000_0001; //129
        if (random[31:28] == 0) begin
          state <= S_SPAWNING;//SPAWN!;
        end
        else begin
          state <= S_DECIDE_SHOOT;
            end
      end

      //SPAWNING STATE
      //Yes, we should spawn.  Iterate through possible
addresses until one is found or end is reached
      else if (state == S_SPAWNING) begin
        if (addr_reg == 134) begin //We're out of bounds
          addr_reg <= 0;
          state <= S_DECIDE_SHOOT;
        end
        else if (public_mem_out[31:28] == 0) begin //We're in
luck, no enemy here
          addr_reg <= addr_reg - 1;
          we_reg <= 1;
          public_mem_in <= {4'h5,random[27:0]};
          private_mem_in <= {12'b0,1'd1,random[30:28],16'd60};
          state <= S_DECIDE_SHOOT;
        end
        else begin //Nope, try next
          addr_reg <= addr_reg + 1;
          state <= S_SPAWNING;
        end
      end

      //DECIDE SHOOT STATE
      else if (state == S_DECIDE_SHOOT) begin
        if (shooting) begin
          addr_reg <= 8'd2;
```

```
          bullet_pause <= 0;
          state <= S_SHOOTING;
        end
        else begin
          state <= S_PREMOVING;
        end
      end

      //SHOOTING STATE
      //Iterate through addresses where we can place a bullet,
give up if out of bounds
      else if (state == S_SHOOTING) begin
        if (addr_reg == 8'b1000_0001) begin //We're out of
bounds (past 128)
          addr_reg <= 0;
          we_reg <= 0;
          state <= S_PREMOVING;
        end
        else if (we_reg == 1) begin
          addr_reg <= addr_reg + 1;
          we_reg <= 0;
          bullet_pause <= 1;
        end
        else if (public_mem_out[31:28] == 0 && !bullet_pause)
begin //We're in luck, no shot here
          addr_reg <= addr_reg - 1;
          we_reg <= 1;
          //public_mem_in <= {4'd2, player_data[27:8], (shots ==
3 ? angle[7:0] : shots == 2 ? angle_plus[7:0] : angle_minus
[7:0])}; //Avatar pos, input rot
          public_mem_in <= {4'd2, player_data[27:8], player_data
[7:0]};
          private_mem_in <= 32'd120;
          shots <= shots - 1;
          if (shots == 1) state <= S_PREMOVING;
          state <= S_PREMOVING;
        end
        else begin //Nope, try next addr
          addr_reg <= addr_reg + 1;
          state <= S_SHOOTING;
          bullet_pause <= 0;
        end
      end

      //PREMOVING STATE
```

```
      //Preliminary movement stuff
      else if (state == S_PREMOVING) begin
        to_movement_ready_delay_reg <= 1;
        addr_reg <= 0;
        we_reg <= 0;
        state <= S_MOVING;
      end

      //MOVING STATE
      //Send over entities one at a time with ready signal.
Send the next one when movement is done.
      //Write received info
      else if (state == S_MOVING) begin
        if (from_movement_done) begin
          //Write data to memory
          public_mem_in <= from_movement_entry;
          private_mem_in <= from_movement_private_entry;
          we_reg <= 1;
          to_movement_ready_delay_reg <= 0;
          if (addr_reg == 255) begin //Are we done?
            state <= S_PRECOLLIDING;
            end
          else if (addr_reg == 0) begin
            player_data <= from_movement_entry;
            end
        end
        else if (we) begin
          //We have just written, so it must be time to send the
next sample
          to_movement_ready_delay_reg <= 1;
          addr_reg <= addr_reg + 1;
          we_reg <= 0;
        end
        else begin
          //They should have received their sample
          to_movement_ready_delay_reg <= 0;
        end
        to_movement_ready_reg <= to_movement_ready_delay_reg;
      end

      //PRECOLLIDING STATE
      //Preliminary collision stuff
      else if (state == S_PRECOLLIDING) begin
        we_reg <= 0;
        if (addr == 0) begin
```

```
          state <= S_COLLIDING;
          to_collision_ready_reg <= 1;
          working_index <= 0;
          addr_reg <= 8'b1000_0000;
        end
        else begin
          addr_reg <= 0;
        end
      end

      //COLLIDING STATE
      //Checks entity from first 128 addresses (primary) and
checks each one against
      //the other 128 (secondary) one by one, stopping if the
end is reached or a collision
      //is found.  The next secondary is prepared immediately
after the collision module
      //received the previous one
      else if (state == S_COLLIDING) begin
        if (to_collision_ready) begin
          to_collision_ready_reg <= 0;
          addr_reg <= 8'b1000_0000; //128
        end

        if (working_index == 8'b1000_0000) begin
          state <= S_DONE;
        end
        else if (from_collision_received) begin
          if (addr_reg[7] == 0) begin //No collisions were found
            addr_reg <= 8'b1000_0000; //128
            to_collision_ready_reg <= 1;
          end
          else if (addr_reg == 8'b1111_1111) begin //We're at
the last secondary, prepare to send the next primary
            working_index <= working_index + 1;
            addr_reg <= working_index + 1;
          end
          else begin //We haven't finished
            addr_reg <= addr_reg + 1;
          end
        end
        else if (from_collision_done) begin
          state <= S_COLLISION_PROCESSING;
          addr_reg <= working_index;
          collide_state <= CS_WAIT_WORKING;
```

```
          working_index <= addr - 1;
        end
      end

      //COLLISION PROCESSING STATE
      //Complex state with sub states.  Checks if both collision
objects are valid,
      //and deletes them if they are.
      else if (state == S_COLLISION_PROCESSING) begin
        if (collide_state == CS_WAIT_WORKING) begin
          addr_reg <= working_index;
          working_index <= addr;
          collide_state <= CS_RECEIVE_WORKING;
        end
        else if (collide_state == CS_RECEIVE_WORKING) begin
          if (public_mem_out[31:28] == 0 || public_mem_out
[31:28] == 4'h3 || public_mem_out[31:28] == 4'h4) begin //This
collision is no good
            working_index <= working_index + 1;
            addr_reg <= working_index + 1;
            collide_state <= CS_FINISH;
          end
          else begin //Primary passes test, check secondary
            addr_reg <= working_index;
            working_index <= addr;
            collide_state <= CS_RECEIVE_COLLIDED;
          end
        end
        else if (collide_state == CS_RECEIVE_COLLIDED) begin
          if (public_mem_out[31:28] == 4'd5 || public_mem_out
[31:28] == 4'd6 || public_mem_out[31:28] == 4'd7) begin //This
collision is no good (spawner)
            working_index <= addr;
            addr_reg <= working_index;
            collide_state <= CS_FINISH;
          end
          else begin //Secondary passes test, proceed to delete
            addr_reg <= working_index;
            working_index <= addr;
            collide_state <= CS_DESTROY_WORKING;
          end
        end
        else if (collide_state == CS_DESTROY_WORKING) begin
          if (working_index == 8'b0) begin
            lives <= lives - 1;
```

```
      public_mem_in <= 32'b0;
      private_mem_in <= 32'b0;
      we_reg <= 1;
      addr_reg <= lives + 3;
      if (lives == 0) begin
        game_over <= 1;
        state <= S_DONE;
        addr_reg <= 0;
      end
      else begin
        state <= S_AVATAR_COLLIDED;
      end
    end
    else begin
      addr_reg <= working_index;
      working_index <= addr;
      public_mem_in <= 32'h0;
      private_mem_in <= 0;
      we_reg <= 1;
      collide_state <= CS_DESTROY_COLLIDED;
    end
  end
  else if (collide_state == CS_DESTROY_COLLIDED) begin
    addr_reg <= working_index;
    working_index <= addr;
    public_mem_in <= {4'h6,public_mem_out[27:8],8'b0};
    private_mem_in <= 32'd10;
    we_reg <= 1;
    collide_state <= CS_CLEANUP;
    score <= score + 1;
  end
  else if (collide_state == CS_CLEANUP) begin
    we_reg <= 0;
    working_index <= working_index + 1;
    addr_reg <= working_index + 1;
    collide_state <= CS_FINISH;
  end
  else if (collide_state == CS_FINISH) begin
    to_collision_ready_reg <= 1;
    state <= S_COLLIDING;
    addr_reg <= 8'b1000_0000;
  end
end

else if (state == S_AVATAR_COLLIDED) begin
```

```
        if (addr_reg == 8'b1111_1111) begin
          state <= S_DONE;
          we_reg <= 0;
        end
        else begin
          addr_reg <= addr_reg + 1;
          public_mem_in <= 32'b0;
          private_mem_in <= 32'b0;
          we_reg <= 1;
        end
      end

      //DONE STATE
      else if (state == S_DONE) begin
        we_reg <= 0;
        addr_reg <= 0;
        state <= S_DONE;
      end

      //RESET STATE
      //Loads in player and lives HUD
      else if (state == S_RESET) begin
        case (addr_reg) //CASE INDICES LAG BY ONE!!!!!!!
              // Player
          255: begin
                public_mem_in <=
32'b0001_0110010000_0100101100_00000000;
              private_mem_in <= 0;
              player_data <=
32'b0001_0000100000_0000100000_00000000;
              end

          // Lives HUD
          3: begin
                public_mem_in <=
32'b0100_1100010000_0000010000_00000000;
              private_mem_in <= 0;
              end
          4: begin
                public_mem_in <=
32'b0100_1100000000_0000010000_00000000;
              private_mem_in <= 0;
          end
          5: begin
```

```
            public_mem_in <=
32'b0100_1011110000_0000010000_00000000;
            private_mem_in <= 0;
          end

          default: begin
            public_mem_in <=0;
            private_mem_in <= 0;
          end
        endcase

        we_reg <= 1;
        addr_reg <= addr_reg + 1;
        if (addr_reg == 8'b1111_1110) begin //if addr == 254
          state <= S_DONE;
            end
      end

      //WHEN IN DOUBT, IDLE
      else begin
        state <= S_DONE;
      end
    end
  end
endmodule



module mybram #(parameter LOGSIZE=14, WIDTH=1) (input wire
[LOGSIZE-1:0] addr, input wire clk, input wire [WIDTH-1:0] din,
output reg [WIDTH-1:0] dout, input wire we);
  // let the tools infer the right number of BRAMs
  (* ram_style = "block" *)
  reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
  always @(posedge clk) begin
    if (we) mem[addr] <= din;
    dout <= mem[addr];
  end
endmodule



module game_module(
  input clock,
  reset,
  vsync,
```

```
  turn_left,
  turn_right,
  shooting,
  input [7:0] graphics_addr,
  output [15:0] score,
  output [31:0] to_graphics_entry
  );

   //Random Number Generator
   wire [31:0] random;
   random_number_generator rng0(.seed(0), .reset(reset), .clock
(clock),
       .random(random));

   wire [7:0] angle;

   //Game Logic, Movement, Collisions
   wire[31:0] to_movement_entry,
       to_movement_private_entry,
       from_movement_entry,
       from_movement_private_entry;
   wire[2:0] state;
   wire from_movement_done,to_movement_ready;
   wire from_collision_received, from_collision_done,
to_collision_ready;
   wire [31:0] to_collision_entry;
   wire [7:0] addr;

   game_logic_module glm0(clock, vsync, reset, random,
to_movement_entry,
       to_movement_private_entry, to_movement_ready,
from_movement_entry,
       from_movement_private_entry, from_movement_done,
graphics_addr,
       to_graphics_entry, from_collision_received,
from_collision_done,
       to_collision_ready, to_collision_entry, angle, shooting,
score, state, addr);

   movement_module mm0(reset, clock, to_movement_entry,
       to_movement_private_entry, random, to_movement_ready,
from_movement_entry,
       from_movement_private_entry, from_movement_done,
turn_left, turn_right);
```

```
   collision_module cm0(clock, reset, to_collision_entry,
to_collision_ready,
       from_collision_received, from_collision_done);

endmodule
```

J Graphics

```
module graphics(
   input vclock,
   input reset,
   input [7:0] switch,
   output game_vsync,
   output [7:0] entity_index,
   input [31:0] entity_data,
   output [35:0] vram0_write_data,
   input [35:0] vram0_read_data,
   output [18:0] vram0_addr,
   output        vram0_we,
   output [35:0] vram1_write_data,
   input [35:0] vram1_read_data,
   output [18:0] vram1_addr,
   output        vram1_we,
   output [7:0] vga_out_red,
   output [7:0] vga_out_green,
   output [7:0] vga_out_blue,
   output vga_out_sync_b,    // not used
   output vga_out_blank_b,
   output vga_out_pixel_clock,
   output vga_out_hsync,
   output vga_out_vsync
);

   // generate basic SVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   svga svga1(.vclock(vclock), .hcount(hcount), .vcount
(vcount), .hsync(hsync),
       .vsync(vsync),.blank(blank));

   // feed SVGA signals to the game
   wire [31:0] pixel;
   wire vhsync,vvsync,vblank;

   wire [10:0]write_x;
   wire [9:0] write_y;
   wire [31:0] write_rgba;
   wire write_enable;

   wire bresenham_ready;
```

```
   wire shape_ready;

   wire [19:0] v0;
   wire [19:0] v1;

   wire [31:0] rgba;

   shape sh (.vclock(vclock), .reset(reset), .vsync(vvsync),
       .entity_index(entity_index), .entity_data
(entity_data), .v0(v0), .v1(v1),
       .rgba(rgba), .shape_ready(shape_ready), .bresenham_ready
(bresenham_ready));

   bresenham bh (.vclock(vclock), .reset(reset), .vsync(vsync),
       .shape_ready(shape_ready), .v0(v0), .v1(v1), .rgba(rgba),
       .bresenham_ready(bresenham_ready), .write_x
(write_x), .write_y(write_y),
       .write_enable(write_enable), .write_rgba(write_rgba));

   frame_buffers vb (.vclock(vclock), .reset(reset), .hcount
(hcount),
       .vcount(vcount), .hsync(hsync), .vsync(vsync), .blank
(blank),
       .write_x(write_x), .write_y(write_y), .write_rgba
(write_rgba),
       .write_enable(write_enable), .addr0(vram0_addr), .addr1
(vram1_addr),
       .write_data0(vram0_write_data), .write_data1
(vram1_write_data),
       .we_0(vram0_we), .we_1(vram1_we), .read_data0
(vram0_read_data),
       .read_data1(vram1_read_data), .write_buf_switch(switch
[2]), .vhsync(vhsync),
       .vvsync(vvsync), .vblank(vblank), .pixel(pixel));

   // switch[1:0] selects which video generator to use:
   //  00: Game
   //  01: 1 pixel outline of active video area (adjust screen
controls)
   //  10: gradient test pattern
   reg [23:0] rgb;
   reg b,hs,vs;

   always @(posedge vclock) begin
      if (switch[1:0] == 2'b01) begin
```

```
       // 1 pixel outline of visible area (white)
       hs <= hsync;
       vs <= vsync;
       b <= blank;
       rgb <= (hcount==0 | hcount==799 | vcount==0 |
vcount==599) ? 24'hFFFFFF : 0;
      end
       else if (switch[1:0] == 2'b10) begin
       // color bars
       hs <= hsync;
       vs <= vsync;
       b <= blank;
       rgb <= hcount;//write_rgba[31:8];
      end
       else begin
         // default: Game
         hs <= vhsync;
         vs <= vvsync;
         b <= vblank;
         rgb <= (vblank ? 24'h000000 : pixel[31:8]);
      end
   end
   // VGA Output.  In order to meet the setup and hold times of
the
   // AD7125, we send it ~clock_40mhz.
   assign vga_out_red = rgb[23:16];
   assign vga_out_green = rgb[15:8];
   assign vga_out_blue = rgb[7:0];
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~vclock;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;
   assign game_vsync = vvsync;
endmodule
```

K Object Movement

```
module movement_module(input reset, clock,
  input [31:0] from_game_public_entry, from_game_private_entry,
random,
  input movement_ready, output [31:0] to_game_public_entry,
to_game_private_entry,
  output reg movement_done, input turn_left, turn_right);

//CONVENTION: 16 bit x and y values have an implied decimal
point
// before the 6 lowest order bits

parameter NO_ID = 4'h0;
parameter AVATAR = 4'h1;
parameter BULLET = 4'h2;
parameter ASTEROID = 4'h3;
parameter SPAWN = 4'h5;
parameter EXPLOSION_0 = 4'h6;
parameter EXPLOSION_1 = 4'h7;

parameter CARTESIAN = 0;
parameter POLAR = 1;

parameter SC_WIDTH = 800;
parameter SC_HEIGHT = 600;
parameter SC_WIDTH_FP = 16'b11_0010_0000__000000;
parameter SC_HEIGHT_FP = 16'b10_0101_1000__000000;

reg [31:0] current_entry_public;
reg [31:0] current_entry_private;
reg [2:0] module_state;
parameter S_IDLE = 3'd0;
parameter S_PROCESSING = 3'd1;
parameter S_BOUND_CHECK = 3'd2;
parameter S_PRE_BOUND = 3'd4;
parameter S_POST_BOUND = 3'd5;

wire [9:0] x, y, new_x, new_y;
wire [3:0] id;
wire [7:0] angle;
reg [3:0] new_id;
reg [7:0] new_angle;
```

```
assign id = current_entry_public[31:28];
assign x = current_entry_public[27:18];
assign y = current_entry_public[17:8];
assign angle = current_entry_public[7:0];

wire signed [5:0] x_precision, y_precision;
assign x_precision = current_entry_private[31:26];
assign y_precision = current_entry_private[25:20];
wire [5:0] new_x_precision, new_y_precision;
wire [19:0] state;
assign state = current_entry_private[19:0];
reg [19:0] new_state;

wire [15:0] full_x, full_y;
assign full_x = {x,x_precision};
assign full_y = {y,y_precision};
wire[15:0] new_full_x, new_full_y;
assign new_x = new_full_x[15:6];
assign new_x_precision = new_full_x[5:0];
assign new_y = new_full_y[15:6];
assign new_y_precision = new_full_y[5:0];
reg [15:0] delta_x, delta_y;

reg [31:0] player_data;
wire [9:0] player_x;
wire [9:0] player_y;
wire [7:0] player_angle;
assign player_x = player_data[27:18];
assign player_y = player_data[17:8];
assign player_angle = player_data[7:0];

wire signed [7:0] sin_angle, cos_angle;

reg movement_mode;
reg [5:0] mag_reg;

wire [15:0] unchecked_full_x, unchecked_full_y;
reg [15:0] fixed_x,fixed_y;

//By how much are we trying to move
wire [15:0] net_delta_x = (movement_mode ? (mag_reg *
cos_angle) : delta_x);
```

```
wire [15:0] net_delta_y = (movement_mode ? (mag_reg *
sin_angle) : delta_y);

//To where are we trying to move
assign unchecked_full_x = full_x + net_delta_x;
assign unchecked_full_y = full_y + net_delta_y;

//Can we move there, or should we move to the nearest edge?
reg corrected_x, corrected_y;

assign new_full_x = corrected_x ? fixed_x : unchecked_full_x;
assign new_full_y = corrected_y ? fixed_y : unchecked_full_y;

trig trig0(angle,sin_angle,cos_angle);

wire [7:0] final_new_angle = new_angle;

assign to_game_private_entry = {new_x_precision
[5:0],new_y_precision[5:0],new_state[19:0]};
assign to_game_public_entry = {new_id[3:0], new_x[9:0], new_y
[9:0], final_new_angle[7:0]};

wire[11:0] radius;
reg [5:0]radius_int;

always @(posedge clock) begin
  if (reset) begin
    movement_done <= 0;
    module_state <= S_IDLE;
  end
  else begin
    if (module_state == S_IDLE) begin
      if (movement_ready) begin
        module_state <= S_PROCESSING;
      end
      movement_done <= 0;
      current_entry_public <= from_game_public_entry;
      current_entry_private <= from_game_private_entry;
    end

    else if (module_state == S_PROCESSING) begin
      //Move according to joystick and face that direction
      if (id == AVATAR) begin
```

```
        movement_mode <= POLAR;
            mag_reg <= 0;
        new_id <= AVATAR;
            if (turn_left) begin
              new_angle <= angle - 8'b0000_0100;
            end
        else if (turn_right) begin
              new_angle <= angle + 8'b0000_0100;
            end
            else begin
              new_angle <= angle;
            end
        new_state <= state;
        player_data <= current_entry_public;
      end

      //Move in a line at a fixed angle, destroy if hit edge or
time up (state)
      else if (id == BULLET) begin
        movement_mode <= POLAR;
        mag_reg <= 6'sb10_0000;
        new_state <= state;// - 1;
        new_angle <= angle;
        if (state == 0 | x < 7'd16 | x > SC_WIDTH - 7'd16 | y <
7'd16 | y > SC_HEIGHT - 7'd16) begin
          new_id <= NO_ID;
        end
        else begin
          new_id <= BULLET;
        end
      end

      //Overflows the sides of the screen
      else if (id == ASTEROID) begin
        delta_x <= state[0] ? 16'b0000_0000_0100_0000 :
16'b1111_1111_1100_0000;
        delta_y <= state[1] ? 16'b0000_0000_0100_0000 :
16'b1111_1111_1100_0000;
        movement_mode <= CARTESIAN;
        new_id <= ASTEROID;
        new_angle <= angle;
        new_state <= state;
      end

      //Turns into an enemy once timer runs out
```

```
else if (id == SPAWN) begin
  delta_x <= 0;
  delta_y <= 0;
  movement_mode <= CARTESIAN;
  new_angle <= angle - 2;
  if (state[5:0] == 0) begin
    new_id <= ASTEROID;
    new_state <= 0;
  end
  else begin
    new_id <= SPAWN;
    new_state <= state - 1;
  end
end

//First explosion sprite
else if (id == EXPLOSION_0) begin
  delta_x <= 0;
  delta_y <= 0;
  movement_mode <= CARTESIAN;
  new_angle <= angle;
  new_state <= state - 1;
  if (state == 0) begin
    new_id <= EXPLOSION_1;
    new_state <= 20'd10;
  end
  else begin
    new_id <= EXPLOSION_0;
      end
end

//Second explosion sprite
else if (id == EXPLOSION_1) begin
  delta_x <= 0;
  delta_y <= 0;
  movement_mode <= CARTESIAN;
  new_angle <= angle;
  new_state <= state - 1;
  if (state == 0) begin
    new_id <= NO_ID;
  end
  else begin
    new_id <= EXPLOSION_1;
  end
end
```

```
      // Default
      else begin
        delta_x <= 0;
        delta_y <= 0;
        movement_mode <= CARTESIAN;
        new_id <= id;
        new_angle <= angle;
        new_state <= state;
      end
      module_state <= S_BOUND_CHECK;
    end

      //Check if the unit is attempting to move outside playable
area
      else if (module_state == S_BOUND_CHECK) begin
        corrected_x <= 0;
        corrected_y <= 0;

        //Too far left
        if (unchecked_full_x[15:6] < 0 | unchecked_full_x[15:6]
> 950) begin //Magic number, assuming you can't reach there from
right
          corrected_x <= 1;
              fixed_x <= SC_WIDTH_FP;
        end

        //Too far right
        else if (unchecked_full_x[15:6] > SC_WIDTH) begin
          corrected_x <= 1;
              fixed_x <= 0;
            end

        //Too far up
        if (unchecked_full_y[15:6] < 0 | unchecked_full_y[15:6]
> 896) begin //Magic number, assuming you can't reach there from
bottom
          corrected_y <= 1;
              fixed_y <= SC_HEIGHT_FP;
            end

        //Too far down
        else if (unchecked_full_y[15:6] > SC_HEIGHT) begin
              corrected_y <= 1;
```

```
            fixed_y <= 0;
        end

        movement_done <= 1;
        module_state <= S_IDLE;
      end

      else begin
        movement_done <= 0;
      end
    end
  end

  assign radius = {radius_int,6'b0};

  always @(id) begin
    case(id)
      NO_ID:       radius_int = 0;
      AVATAR:      radius_int = 8;
      BULLET:      radius_int = 4;
      ASTEROID:    radius_int = 16;
      SPAWN:       radius_int = 16;
      EXPLOSION_0: radius_int = 16;
      EXPLOSION_1: radius_int = 16;
      default:     radius_int = 16;
    endcase
  end
endmodule
```

L Random Number Generator

```
module random_number_generator
   #(parameter LOG_2_M = 32, A = 22695477, C = 1)
   (input reset,clock,input [LOG_2_M-1:0] seed, output
[LOG_2_M-1:0] random);

   reg[LOG_2_M-1:0] random_reg;
   always @(posedge clock) begin
       if (reset)
          random_reg <= seed;
       else
          random_reg <= A*random_reg + C; //Overflow intended
       end
   assign random = random_reg;
endmodule
```

M Shapes

```
module shape_table (
    input [3:0] id,
    input [3:0] segment,
    output reg signed  [7:0] x0,
    output reg signed [7:0] y0,
    output reg signed [7:0] x1,
    output reg signed [7:0] y1,
    output reg [31:0] rgba,
    output reg ignore
);

    parameter NO_ID = 4'h0;
    parameter AVATAR = 4'h1;
    parameter BULLET = 4'h2;
    parameter ASTEROID = 4'h3;
    parameter LIFE = 4'h4;
    parameter SPAWN = 4'h5;
    parameter EXPLOSION_0 = 4'h6;
    parameter EXPLOSION_1 = 4'h7;

    always @(*) begin
      case (id)
        NO_ID: begin
          x0 = 0;
          y0 = 0;
          x1 = 0;
          y1 = 0;
          ignore = 1;
        end

        AVATAR: begin
          case (segment)
                  4'h0: begin
                      x0 = 6;
                      y0 = 0;
                      x1 = -6;
                      y1 = 4;
                      ignore = 0;
                    end
                    4'h1: begin
                      x0 = -6;
                      y0 = 4;
                      x1 = -6;
```

```
            y1 = -4;
            ignore = 0;
          end
          4'h2: begin
            x0 = -6;
            y0 = -4;
            x1 = 6;
            y1 = 0;
          end
          default: begin
    x0 = 0;
    y0 = 0;
    x1 = 0;
    y1 = 0;
    ignore = 1;
  end
 endcase
 rgba = 32'hFFFFFFFF;
end

BULLET: begin
  case (segment)
    4'h0: begin
      x0 = 1;
      y0 = 1;
      x1 = 1;
      y1 = -1;
      ignore = 0;
    end
    4'h1: begin
      x0 = 1;
      y0 = -1;
      x1 = -1;
      y1 = -1;
      ignore = 0;
    end
    4'h2: begin
      x0 = -1;
      y0 = -1;
      x1 = -1;
      y1 = 1;
      ignore = 0;
    end
    4'h3: begin
      x0 = -1;
```

```
      y0 = 1;
      x1 = 1;
      y1 = 1;
      ignore = 0;
    end
    default: begin
      x0 = 0;
      y0 = 0;
      x1 = 0;
      y1 = 0;
      ignore = 1;
    end
  endcase
  rgba = 32'hFFFFFFFF;
end

LIFE: begin
  ignore = 0;
  case (segment)
    4'h0: begin
      x0 = 0;
      y0 = 6;
      x1 = -4;
      y1 = -6;
              ignore = 0;
    end
    4'h1: begin
      x0 = -4;
      y0 = -6;
      x1 = 4;
      y1 = -6;
              ignore = 0;
    end
    4'h2: begin
      x0 = 4;
      y0 = -6;
      x1 = 0;
      y1 = 6;
              ignore = 0;
    end
    default: begin
      x0 = 0;
      y0 = 0;
      x1 = 0;
      y1 = 0;
```

```
            ignore = 1;
          end
        endcase
        rgba = 32'hFFFF00FF;
      end

/*
      SPAWN: begin
        case (segment)
          4'h0: begin
            x0 = 0;
            y0 = 0;
            x1 = -1;
            y1 = -4;
            ignore = 0;
          end
          4'h1: begin
            x0 = -1;
            y0 = -4;
            x1 = 0;
            y1 = -8;
            ignore = 0;
          end
          4'h2: begin
            x0 = 0;
            y0 = -8;
            x1 = 4;
            y1 = -10;
            ignore = 0;
          end
          4'h3: begin
            x0 = 4;
            y0 = -10;
            x1 = 8;
            y1 = -8;
            ignore = 0;
          end
          4'h4: begin
            x0 = 0;
            y0 = 0;
            x1 = 4;
            y1 = 1;
            ignore = 0;
          end
          4'h5: begin
```

```
    x0 = 4;
    y0 = 1;
    x1 = 7;
    y1 = 4;
    ignore = 0;
  end
  4'h6: begin
    x0 = 7;
    y0 = 4;
    x1 = 6;
    y1 = 8;
    ignore = 0;
  end
  4'h7: begin
    x0 = 6;
    y0 = 8;
    x1 = 3;
    y1 = 11;
    ignore = 0;
  end
  4'h8: begin
    x0 = 0;
    y0 = 0;
    x1 = -3;
    y1 = 3;
    ignore = 0;
  end
  4'h9: begin
    x0 = -3;
    y0 = 3;
    x1 = -7;
    y1 = 4;
    ignore = 0;
  end
  4'hA: begin
    x0 = -7;
    y0 = 4;
    x1 = -10;
    y1 = 1;
    ignore = 0;
  end
  4'hB: begin
    x0 = -10;
    y0 = 1;
    x1 = -11;
    y1 = -3;
```

```
          ignore = 0;
        end
        default: begin
          x0 = 0;
          y0 = 0;
          x1 = 0;
          y1 = 0;
          ignore = 1;
        end
      endcase
      rgba = 32'hFF0000FF;
    end
*/

    EXPLOSION_0: begin
      case (segment)
        4'h0: begin
          x0 = -3;
          y0 = -5;
          x1 = 3;
          y1 = -5;
          ignore = 0;
        end
        4'h1: begin
          x0 = 3;
          y0 = -5;
          x1 = 6;
          y1 = 0;
          ignore = 0;
        end
        4'h2: begin
          x0 = 6;
          y0 = 0;
          x1 = 3;
          y1 = 5;
          ignore = 0;
        end
        4'h3: begin
          x0 = 3;
          y0 = 5;
          x1 = -3;
          y1 = 5;
          ignore = 0;
        end
        4'h4: begin
```

```
        x0 = -3;
        y0 = 5;
        x1 = -6;
        y1 = 0;
        ignore = 0;
      end
      4'h5: begin
        x0 = -6;
        y0 = 0;
        x1 = -3;
        y1 = -5;
        ignore = 0;
      end
      default: begin
        x0 = 0;
        y0 = 0;
        x1 = 0;
        y1 = 0;
        ignore = 1;
      end
    endcase
    rgba = 32'hFF7700FF;
  end

  EXPLOSION_1: begin
    case (segment)
      4'h0: begin
        x0 = -3;
        y0 = -5;
        x1 = 3;
        y1 = -5;
        ignore = 0;
      end
      4'h1: begin
        x0 = 3;
        y0 = -5;
        x1 = 6;
        y1 = 0;
        ignore = 0;
      end
      4'h2: begin
        x0 = 6;
        y0 = 0;
        x1 = 3;
        y1 = 5;
```

```
    ignore = 0;
  end
  4'h3: begin
    x0 = 3;
    y0 = 5;
    x1 = -3;
    y1 = 5;
    ignore = 0;
  end
  4'h4: begin
    x0 = -3;
    y0 = 5;
    x1 = -6;
    y1 = 0;
    ignore = 0;
  end
  4'h5: begin
    x0 = -6;
    y0 = 0;
    x1 = -3;
    y1 = -5;
    ignore = 0;
  end
  4'h6: begin
    x0 = -3;
    y0 = -11;
    x1 = 3;
    y1 = -11;
    ignore = 0;
  end
  4'h7: begin
    x0 = 8;
    y0 = -8;
    x1 = 11;
    y1 = -3;
    ignore = 0;
  end
  4'h8: begin
    x0 = 11;
    y0 = 3;
    x1 = 8;
    y1 = 8;
    ignore = 0;
  end
  4'h9: begin
    x0 = 3;
```

```
      y0 = 11;
      x1 = -3;
      y1 = 11;
      ignore = 0;
    end
    4'hA: begin
      x0 = -8;
      y0 = 8;
      x1 = -11;
      y1 = 3;
      ignore = 0;
    end
    4'hB: begin
      x0 = -11;
      y0 = -3;
      x1 = -8;
      y1 = -8;
      ignore = 0;
    end
    default: begin
      x0 = 0;
      y0 = 0;
      x1 = 0;
      y1 = 0;
      ignore = 1;
    end
  endcase
  rgba = (segment > 5 ? 32'hFF7700FF : 32'h993300FF);
end

ASTEROID: begin
  case (segment)
    4'h0: begin
      x0 = -3;
      y0 = -11;
      x1 = 3;
      y1 = -11;
      ignore = 0;
    end
    4'h1: begin
      x0 = 8;
      y0 = -8;
      x1 = 11;
      y1 = -3;
      ignore = 0;
    end
```

```
4'h2: begin
  x0 = 11;
  y0 = 3;
  x1 = 8;
  y1 = 8;
  ignore = 0;
end
4'h3: begin
  x0 = 3;
  y0 = 11;
  x1 = -3;
  y1 = 11;
  ignore = 0;
end
4'h4: begin
  x0 = -8;
  y0 = 8;
  x1 = -11;
  y1 = 3;
  ignore = 0;
end
4'h5: begin
  x0 = -11;
  y0 = -3;
  x1 = -8;
  y1 = -8;
  ignore = 0;
end
4'h6: begin
  x0 = 3;
  y0 = -11;
  x1 = 3;
  y1 = -5;
  ignore = 0;
end
4'h7: begin
  x0 = 3;
  y0 = -5;
  x1 = 8;
  y1 = -8;
  ignore = 0;
end
4'h8: begin
  x0 = 11;
  y0 = -3;
  x1 = 11;
```

```
    y1 = 3;
    ignore = 0;
  end
  4'h9: begin
    x0 = 8;
    y0 = 8;
    x1 = 3;
    y1 = 5;
    ignore = 0;
  end
  4'hA: begin
    x0 = 3;
    y0 = 5;
    x1 = 3;
    y1 = 11;
    ignore = 0;
  end
  4'hB: begin
    x0 = -3;
    y0 = 11;
    x1 = -8;
    y1 = 8;
    ignore = 0;
  end
  4'hC: begin
    x0 = -11;
    y0 = 3;
    x1 = -6;
    y1 = 0;
    ignore = 0;
  end
  4'hD: begin
    x0 = -6;
    y0 = 0;
    x1 = -11;
    y1 = -3;
    ignore = 0;
  end
  4'hE: begin
    x0 = -8;
    y0 = -8;
    x1 = -3;
    y1 = -11;
    ignore = 0;
  end
  default: begin
```

```
            x0 = 0;
            y0 = 0;
            x1 = 0;
            y1 = 0;
            ignore = 1;
          end
        endcase
        rgba = 32'hFFFFFFFF;
      end

      default: begin
        x0 = 0;
        y0 = 0;
        x1 = 0;
        y1 = 0;
        ignore = 1;
        rgba = 32'h00000000;
      end
    endcase
  end
endmodule

module shape (
  input vclock,
  input reset,
  input vsync,

  output reg [7:0] entity_index,
  input [31:0] entity_data,

  output reg [19:0] v0,
  output reg [19:0] v1,
  output reg [31:0] rgba,
  output reg shape_ready,
  input bresenham_ready
);

  wire [3:0] id;
  wire signed [10:0] x, y;
  wire [7:0] theta;
  wire signed [7:0] sine;
  wire signed [7:0] cosine;
  assign id = entity_data[31:28];
  assign x = {1'b0, entity_data[27:18]};
  assign y = {1'b0, entity_data[17:8]};
```

```
  assign theta = entity_data[7:0];
  reg [3:0] segment;
  wire signed [7:0] x0, x1, y0, y1;
  wire ignore;
  wire [31:0] seg_rgba;

  shape_table st (.id(id), .segment(segment), .x0(x0), .y0
(y0), .x1(x1), .y1(y1),
    .rgba(seg_rgba), .ignore(ignore));

  trig tr (.THETA(theta), .SINE(sine), .COSINE(cosine));

  wire signed [15:0] x0_offset, y0_offset, x1_offset, y1_offset;
  wire signed [8:0] minus_sine_intermediate = ~sine + 8'h01;
  wire signed [7:0] minus_sine = minus_sine_intermediate[7:0];

  assign x0_offset = x0 * cosine + y0 * minus_sine;
  assign y0_offset = x0 * sine + y0 * cosine;
  assign x1_offset = x1 * cosine + y1 * minus_sine;
  assign y1_offset = x1 * sine + y1 * cosine;

  wire signed [10:0] x0s, y0s, x1s, y1s;

  assign x0s = x + x0_offset[15:6];
  assign y0s = y + y0_offset[15:6];
  assign x1s = x + x1_offset[15:6];
  assign y1s = y + y1_offset[15:6];

  reg lookup; // introduces a delay due to apparent setup/hold
time issues

  always @(posedge vclock) begin
    if (reset || !vsync) begin
      entity_index <= 0;
      segment <= 0;
      shape_ready <= 0;
      v0 <= 0;
      v1 <= 0;
      rgba <= 0;
      lookup <= 0;
    end
    else begin
      if (shape_ready == 0 && bresenham_ready && !lookup) begin
        if (segment == 15) begin
          segment <= 0;
          if (entity_index != 255) begin
```

```
          entity_index <= entity_index + 1;
        end
      end
      else begin
        segment <= segment + 1;
      end
      lookup <= 1;
    end
    else if (lookup) begin
      lookup <= 0;
      if (!ignore) begin
        v0 <= {x0s[9:0],y0s[9:0]};
        v1 <= {x1s[9:0],y1s[9:0]};
        rgba <= seg_rgba;
        shape_ready <= 1;
      end
    end
    else if (shape_ready == 1) begin
      shape_ready <= 0;
    end
   end
 end
endmodule
```

N Labkit

```
module labkit(
  // Remove comment from any signals you use in your design!

  // AC97
  output wire beep, audio_reset_b, ac97_synch, ac97_sdata_out,
  input wire ac97_bit_clock, ac97_sdata_in,

  // VGA
  output wire [7:0] vga_out_red, vga_out_green, vga_out_blue,
  output wire vga_out_sync_b, vga_out_blank_b,
vga_out_pixel_clock, vga_out_hsync, vga_out_vsync,


  // NTSC OUT
  /*
  output wire [9:0] tv_out_ycrcb,
  output wire tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
tv_out_i2c_data,
  output wire tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b,
tv_out_blank_b,
  output wire tv_out_subcar_reset;

  */
  // NTSC IN
  /*
  input wire [19:0] tv_in_ycrcb,
  input wire tv_in_data_valid, tv_in_line_clock1,
tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
  output wire tv_in_i2c_clock, tv_in_fifo_read,
tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,
  inout wire tv_in_i2c_data,
  */

  // ZBT RAMS
  inout wire [35:0] ram0_data,
  output wire [18:0] ram0_address,
  output wire ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b,
ram0_oe_b, ram0_we_b,
  output wire [3:0] ram0_bwe_b,
  inout wire [35:0]ram1_data,
  output wire [18:0]ram1_address,
  output wire ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b,
ram1_oe_b, ram1_we_b,
  output wire [3:0] ram1_bwe_b,
```

```
  input wire clock_feedback_in,
  output wire clock_feedback_out,

  // FLASH
  /*
  inout wire [15:0] flash_data,
  output wire [23:0] flash_address,
  output wire flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b,
flash_byte_b,
  input  wire flash_sts,

  */
  // RS232
  /*
  output wire rs232_txd, rs232_rts,
  input wire rs232_rxd, rs232_cts,
  */

  // PS2
  //input wire mouse_clock, mouse_data,
  //input wire keyboard_clock, keyboard_data,

  // FLUORESCENT DISPLAY
  output wire disp_blank, disp_clock, disp_rs, disp_ce_b,
disp_reset_b,
  input wire disp_data_in,
  output wire disp_data_out,

  // SYSTEM ACE
  /*
  inout wire [15:0] systemace_data,
  output wire [6:0] systemace_address,
  output wire systemace_ce_b, systemace_we_b, systemace_oe_b,
  input wire systemace_irq, systemace_mpbrdy,
  */

  // BUTTONS, SWITCHES, LEDS
  input wire button0,
  input wire button1,
  input wire button2,
  input wire button3,
  input wire button_enter,
  input wire button_right,
  input wire button_left,
  input wire button_down,
  input wire button_up,
```

```
  input wire [7:0] switch,
  output wire [7:0] led,

  // USER CONNECTORS, DAUGHTER CARD, LOGIC ANALYZER
  //inout wire [31:0] user1,
  //inout wire [31:0] user2,
  inout wire [31:0] user3,
  //inout wire [31:0] user4,
  //inout wire [43:0] daughtercard,
  //output wire [15:0] analyzer1_data, output wire
analyzer1_clock,
  //output wire [15:0] analyzer2_data, output wire
analyzer2_clock,
  //output wire [15:0] analyzer3_data, output wire
analyzer3_clock,
  //output wire [15:0] analyzer4_data, output wire
analyzer4_clock,
  // CLOCKS
  //input wire clock1,
  //input wire clock2,
  input wire clock_27mhz
);

// use FPGA's digital clock manager to produce a
// 40MHz clock
wire clock_40mhz_unbuf,clock_40mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 21
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 31
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));
   wire vclock;
   wire locked;


////////////////////////////////////////////////////////////////
////////////
  //
  // Reset Generation
  //
  // A shift register primitive is used to generate an active-
high reset
  // signal that remains high for 16 clock cycles after
configuration finishes
```

```
  // and the FPGA's internal clocks begin toggling.
  //

////////////////////////////////////////////////////////////////
////////////

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(vclock), .Q
(power_on_reset), .A0(1'b1), .A1(1'b1),
       .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset, user_reset;
   debounce db0(.reset(power_on_reset), .clock(vclock), .noisy
(~button_enter),
       .clean(user_reset));
   assign reset = user_reset | power_on_reset | !locked;

   ///////// 65 KHz clock for accelerometer ////////////
   reg clock_65khz;
   reg [9:0] clock_65_counter;
   always @(posedge clock_40mhz) begin
     if (clock_65_counter == 10'b1111101000) begin
       clock_65khz <= ~clock_65khz;
       clock_65_counter <= 0;
     end
     else begin
      clock_65_counter <= clock_65_counter + 1;
     end
   end
   ////////////////////////////////////

     assign beep= 1'b0;
   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'b0;
   assign ram1_ce_b = 1'b0;
   assign ram1_oe_b = 1'b0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_bwe_b = 4'b0;
```

```
   ramclock rc(.ref_clock(clock_40mhz), .fpga_clock
(vclock), .ram0_clock(ram0_clk),
       .ram1_clock(ram1_clk), .clock_feedback_in
(clock_feedback_in),
       .clock_feedback_out(clock_feedback_out), .locked
(locked));

   wire vvsync;
   wire [31:0] entity_data;
   wire [7:0] entity_index;
   wire [35:0] vram0_write_data;
   wire [35:0] vram0_read_data;
   wire [18:0] vram0_addr;
   wire vram0_we;


   // clock enable (should be synchronous and one cycle high at
a time)
   assign ram0_cen_b = 0;
     assign ram1_cen_b = 0;

   // create delayed ram_we signal: note the delay is by two
cycles!
   // ie we present the data to be written two cycles after we
is raised
   // this means the bus is tri-stated two cycles after we is
raised.
   reg [1:0] we0_delay;
   always @(posedge vclock) we0_delay <= {we0_delay
[0],vram0_we};

   // create two-stage pipeline for write data
   reg [35:0]  write_data0_old1;
   reg [35:0]  write_data0_old2;
   always @(posedge vclock)
     {write_data0_old2, write_data0_old1} <= {write_data0_old1,
vram0_write_data};

   // wire to ZBT RAM signals
   assign ram0_we_b = ~vram0_we;
   assign ram0_address = vram0_addr;
   assign ram0_data = we0_delay[1] ? write_data0_old2 : {36
{1'bZ}};
   assign vram0_read_data = ram0_data;
   wire [35:0] vram1_write_data;
   wire [35:0] vram1_read_data;
```

```
   wire [18:0] vram1_addr;
   wire vram1_we;

   // create delayed ram_we signal: note the delay is two cycles
   // i.e., we present the data to be written two cycles after
we is raised
   // this means the bus is tri-stated two cycles after we is
raised.
   reg [1:0] we1_delay;
   always @(posedge vclock)
     we1_delay <= {we1_delay[0],vram1_we};

   // create two-stage pipeline for write data
   reg [35:0]  write_data1_old1;
   reg [35:0]  write_data1_old2;
   always @(posedge vclock)
       {write_data1_old2, write_data1_old1} <=
{write_data1_old1, vram1_write_data};

   // wire to ZBT RAM signals
   assign ram1_we_b = ~vram1_we;
   assign ram1_address = vram1_addr;
   assign ram1_data = we1_delay[1] ? write_data1_old2 : {36
{1'bZ}};
   assign vram1_read_data = ram1_data;

   //Score
   wire [15:0] score;

   wire turn_left, turn_left_btn, turn_left_adc;
   wire turn_right, turn_right_btn, turn_right_adc;
   wire shooting, shooting_btn, shooting_adc;
   debounce left(.reset(reset), .clock(clock_40mhz), .noisy
(~button_left),
     .clean(turn_left_btn));
   debounce right(.reset(reset), .clock(clock_40mhz), .noisy
(~button_right),
     .clean(turn_right_btn));
   debounce shoot(.reset(reset), .clock(clock_40mhz), .noisy
(~button3),
     .clean(shooting_btn));
   assign turn_left = turn_left_btn | turn_left_adc;
   assign turn_right = turn_right_btn | turn_right_adc;
   assign shooting = shooting_btn | shooting_adc;
   assign shooting_adc = adc_shoot;
```

```
////////////////////////////////////////////////

   wire shoot_button, collision_button, over_button;
   debounce d1(.reset(reset), .clock(clock_40mhz), .noisy
(~button0), .clean(shoot_button));
   debounce d2(.reset(reset), .clock(clock_40mhz), .noisy
(~button1), .clean(collision_button));
   debounce d3(.reset(reset), .clock(clock_40mhz), .noisy
(~button2), .clean(over_button));

   wire [8:0] shooter_dir = {switch[7:0],1'b0};
   wire [10:0] b_x1, b_x2, b_x3, b_x4, b_x5;
   wire [9:0] b_y1, b_y2, b_y3, b_y4, b_y5;
   wire [7:0] hex_dig_data_rl, hex_dig_data_shoot;
   wire adc_shoot;
   wire adc_rd, adc_rdy;
   wire adc_address;
   wire [1:0] adc_state;
   wire [13:0] x_angle, y_angle;
   wire [10:0] shoot_sin;

   reg adc_shooter1, adc_shooter2, adc_shooter3, adc_shooter4,
adc_shooter5;
   wire b_e1, b_e2, b_e3, b_e4, b_e5;

   reg adc_delay;
   //reg adc_delay1, adc_delay2, adc_delay3, adc_delay4;
   reg [2:0] adc_count = 0;
   reg adc_shoot1, adc_shoot2, adc_shoot3, adc_shoot4;
   wire bf1, bf2, bf3, bf4, bf5;
     wire [23:0] pixel, pixel1, pixel2, pixel3, pixel4, pixel5;

always @(posedge clock_40mhz) begin
     adc_delay <= adc_shoot; //adc_shoot shoot_button
     if (adc_shoot == 1 && adc_delay == 0) begin //adc_shoot
          if ((b_e1 == 0) && (~b_e2 || bf2) && (~b_e3 || bf3) &&
(~b_e4 || bf4) && (~b_e5 || bf5)) begin
               adc_shooter1 <= 1;
               adc_shooter2 <= 0;
               adc_shooter3 <= 0;
               adc_shooter4 <= 0;
               adc_shooter5 <= 0;
          end
```

```
          else if ((b_e2 == 0) && (~b_e1 || bf1) && (~b_e3 ||
bf3) && (~b_e4 || bf4) && (~b_e5 || bf5)) begin
               adc_shooter2 <= 1;
               adc_shooter1 <= 0;
               adc_shooter3 <= 0;
               adc_shooter4 <= 0;
               adc_shooter5 <= 0;
          end
          else if ((b_e3 == 0) && (~b_e2 || bf2) && (~b_e1 ||
bf1) && (~b_e4 || bf4) && (~b_e5 || bf5)) begin
               adc_shooter3 <= 1;
               adc_shooter1 <= 0;
               adc_shooter2 <= 0;
               adc_shooter4 <= 0;
               adc_shooter5 <= 0;
          end
          else if ((b_e4 == 0) && (~b_e2 || bf2) && (~b_e3 ||
bf3) && (~b_e1 || bf1) && (~b_e5 || bf5)) begin
               adc_shooter4 <= 1;
               adc_shooter1 <= 0;
               adc_shooter2 <= 0;
               adc_shooter3 <= 0;
               adc_shooter5 <= 0;
          end
          else if ((b_e5 == 0) && (~b_e2 || bf2) && (~b_e3 ||
bf3) && (~b_e4 || bf4) && (~b_e1 || bf1)) begin
               adc_shooter5 <= 1;
               adc_shooter1 <= 0;
               adc_shooter2 <= 0;
               adc_shooter3 <= 0;
               adc_shooter4 <= 0;
          end
     end
     else begin
          adc_shooter1 <= 0;
          adc_shooter2 <= 0;
          adc_shooter3 <= 0;
          adc_shooter4 <= 0;
          adc_shooter5 <= 0;
     end
end

shooter shoot1(.adc_shoot(adc_shooter1), .clock
(clock_40mhz), .reset(1'b0),
                    .hcount(hcount), .vcount(vcount),
```

```
                    .hsync(hsync), .vsync(vsync), .blank(blank),
                    .dir(shooter_dir), .game_over
(1'b0), .collision(collision_button),
                    .pixel(pixel1),
                    .bullet_existence(b_e1), .x_coord
(b_x1), .y_coord(b_y1), .bullet_far(bf1));

shooter shoot2(.adc_shoot(adc_shooter2), .clock
(clock_40mhz), .reset(1'b0),
                    .hcount(hcount), .vcount(vcount),
                    .hsync(hsync), .vsync(vsync), .blank(blank),
                    .dir(shooter_dir), .game_over
(1'b0), .collision(collision_button),
                    .pixel(pixel2),
                    .bullet_existence(b_e2), .x_coord
(b_x2), .y_coord(b_y2), .bullet_far(bf2));

shooter shoot3(.adc_shoot(adc_shooter3), .clock
(clock_40mhz), .reset(1'b0),
                    .hcount(hcount), .vcount(vcount),
                    .hsync(hsync), .vsync(vsync), .blank(blank),
                    .dir(shooter_dir), .game_over
(1'b0), .collision(collision_button),
                    .pixel(pixel3),
                    .bullet_existence(b_e3), .x_coord
(b_x3), .y_coord(b_y3), .bullet_far(bf3));

shooter shoot4(.adc_shoot(adc_shooter4), .clock
(clock_40mhz), .reset(1'b0),
                    .hcount(hcount), .vcount(vcount),
                    .hsync(hsync), .vsync(vsync), .blank(blank),
                    .dir(shooter_dir), .game_over
(1'b0), .collision(collision_button),
                    .pixel(pixel4),
                    .bullet_existence(b_e4), .x_coord
(b_x4), .y_coord(b_y4), .bullet_far(bf4));

shooter shoot5(.adc_shoot(adc_shooter5), .clock
(clock_40mhz), .reset(1'b0),
                    .hcount(hcount), .vcount(vcount),
                    .hsync(hsync), .vsync(vsync), .blank(blank),
                    .dir(shooter_dir), .game_over
(1'b0), .collision(collision_button),
                    .pixel(pixel5),
                    .bullet_existence(b_e5), .x_coord
(b_x5), .y_coord(b_y5), .bullet_far(bf5));
```

```
//
wire [10:0] ship_x;
wire [9:0] ship_y;
wire [23:0] pixels;
//fakeship fship(.clock(clock_40mhz), .right(turn_right), .left
(turn_left),
//             .collision(collision_button), .hsync
(hsync), .vsync(vsync), .blank(blank),
//             .hcount(hcount), .vcount(vcount),
//             .phsync(phsyncs), .pvsync(pvsyncs), .pblank
(pblanks), .pixel(pixels),
//             .x_coord(ship_x), .y_coord(ship_y));

   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;

     wire vup,vdown;
   reg old_vup,old_vdown;
   debounce bup(.reset(reset),.clock(clock_27mhz),.noisy
(~button_up),.clean(vup));
   debounce bdown(.reset(reset),.clock(clock_27mhz),.noisy
(~button_down),.clean(vdown));
   reg [4:0] volume;
   always @ (posedge clock_27mhz) begin
     if (reset) volume <= 5'd8;
     else begin
     if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
     if (vdown & ~old_vdown & volume != 5'd0) volume <=
volume-1;
     end
     old_vup <= vup;
     old_vdown <= vdown;
   end

   lab5audio a(clock_27mhz, reset, volume,
               from_ac97_data, to_ac97_data, //output, input
               ready,audio_reset_b, ac97_sdata_out,
ac97_sdata_in,
            ac97_synch, ac97_bit_clock);



sound_output soundout(.clock(clock_27mhz), .ac97(ac97_synch),
                    .shoot(adc_shoot), .collision
(collision_button),
                    .ready(ready), .ac97_noise(to_ac97_data));
```

```
   assign pixel = pixels + pixel1 + pixel2 + pixel3 + pixel4 +
pixel5;

wire [7:0] data_bus_7bit = user3[31:24];

adc adc_exp(.clock(clock_65khz), .data_bus
(data_bus_7bit), .adc_rdy(adc_rdy),
          .address(adc_address), .state(adc_state), .adc_rd
(adc_rd),
          .turn_right(turn_right_adc), .turn_left
(turn_left_adc), .shoot(adc_shoot),
          .dig_data_rl(hex_dig_data_rl), .dig_data_shoot
(hex_dig_data_shoot));

assign led[7] = ~b_e1;
assign led[6] = ~b_e2;
assign led[5] = ~b_e3;
assign led[4] = ~collision_button;
assign led[3] = ~shoot_button;
assign led[2] = ~bf1;//debug
assign led[1] = ~bf2;
assign led[0] = ~adc_shoot;
assign user3[10] = adc_address;
assign user3[1] = adc_rd;
assign adc_rdy = user3[4];

wire [63:0] data16_state = {5'b0,ship_x,b_x3[10:3],b_y3
[9:2],b_x2[10:3],b_y2[9:2],b_x1[10:3],b_y1[9:2]};
  display_16hex d16hex(.reset(reset), .clock_27mhz(clock_27mhz),
    .data(data16_state), .disp_blank(disp_blank), .disp_clock
(disp_clock),
      .disp_rs(disp_rs), .disp_ce_b(disp_ce_b), .disp_reset_b
(disp_reset_b),
      .disp_data_out(disp_data_out));

///////////////////////////////////////////

     //Game module
   game_module gm0(.clock(vclock), .reset(reset), .vsync
(vga_out_vsync),
     .turn_left(turn_left), .turn_right(turn_right), shooting,
```

```
	.graphics_addr(entity_index[7:0]), .score(score),
	.to_graphics_entry(entity_data[31:0]));

	//Graphics module
   graphics g (.vclock(vclock), .reset(reset), .switch(switch),
	.game_vsync(vvsync), .entity_index
(entity_index), .entity_data(entity_data),
	.vram0_write_data(vram0_write_data), .vram0_read_data
(vram0_read_data),
	.vram0_addr(vram0_addr), .vram0_we
(vram0_we), .vram1_write_data(vram1_write_data),
	.vram1_read_data(vram1_read_data), .vram1_addr
(vram1_addr), .vram1_we(vram1_we),
	.vga_out_red(vga_out_red), .vga_out_green(vga_out_green),
	.vga_out_blue(vga_out_blue), .vga_out_sync_b
(vga_out_sync_b),
	.vga_out_blank_b(vga_out_blank_b), .vga_out_pixel_clock
(vga_out_pixel_clock),
	.vga_out_hsync(vga_out_hsync), .vga_out_vsync
(vga_out_vsync));

endmodule
```