ASTEROIDS

6.111 Fall 2011

Final Project Report

Richard Agbeyibor Phillip Mercer

Abstract

Our project will be to recreate the Atari classic video game, Asteroids, using the Xilinx II FPGA board. In the game Asteroids, you navigate a spaceship in a simple square wrap around screen and shoot at incoming asteroids and space debris in order to protect your ship. The ship is able to rotate and continuously accelerate based on user input. The difficulties in this project are the number of sprites, the object interactions, and the ship acceleration.

We will separate this game into fundamental modules of game logic, physics, audio, and visual display. Additional modules would be associated with multiplayer game play, such as any wireless transmissions and communications protocols

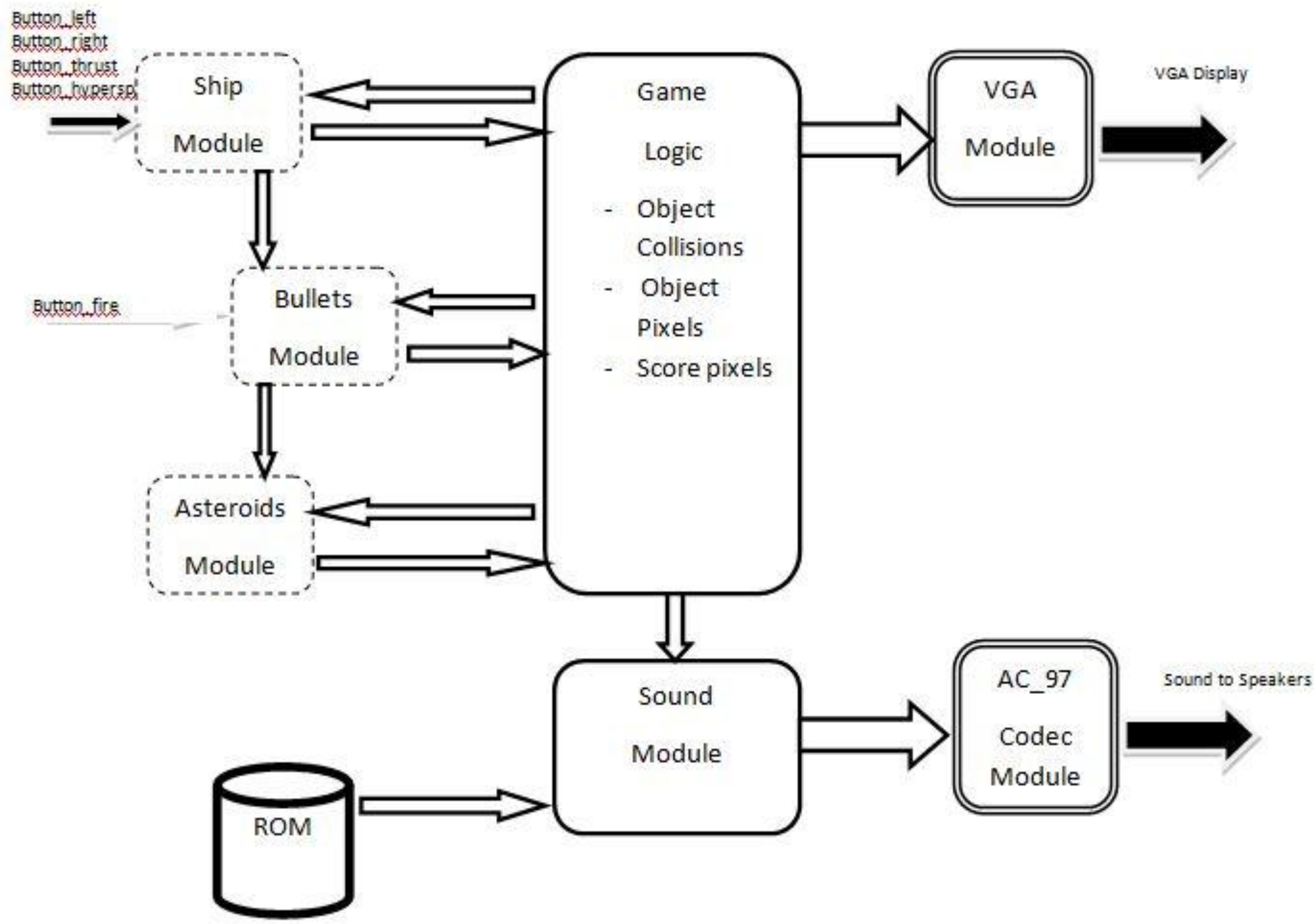
Table of Contents:

Introduction

The objective of this project is to recreate the classic video game Asteroids, which was originally released to the Atari 2600, on the Xilinx II FPGA. In the game Asteroids, the player navigates a spaceship in outer space and shoots at incoming asteroids and other space debris in order to protect the ship. The ship has thrusters and a gun, which allow it to rotate, accelerate, and shoot incoming asteroids.

The game design is divided into the physical design of the objects: the ship, the bullets, the asteroids, and their physical behavior on screen. There are multiple instances of these objects on screen and they all interact with each other. A sound module provides a musical score to the game events.

Block Diagram



Modules

Ship

The ship module controls the position, velocity and heading of the spaceship in the game Asteroids. This object moves freely in 2-Dimensional space. A gun is mounted to the ship which keeps the bearing of the ship, and determines the azimuth of the bullets.

The ship module takes as input user-controlled buttons for right and left rotate, and thrust to control the movement of the ship.

The ship module outputs the center coordinates of the ship, its velocity and heading. The module also outputs to the bullet module information about the gun, such as its center coordinates, and velocity.

In the project proposal, the specifications describe a triangular ship, as in the original Asteroids game. An equilateral triangle would be drawn given center coordinates using the Bresenham line drawing algorithm.

In the original game, the ship reacts to drag and thus slows down and eventually stops when the user releases the thrust button.

As the ship can be drawn directly from its center coordinates, it is possible to represent it as a point object and calculate its movement using Newtonian equations of motion.

The following formulas can be used to calculate the acceleration, velocity and position of the ship:

```
acc <=  (thrust)?  (thrust_coeff - drag_coeff) : (vx>0 or vy>0) ? (- drag_coeff) : 0
vx <= vx + acc*arccos(heading)*delta_t
vy <= vy + acc*arcsin(heading)*delta_t
x <= x + vx* delta_t
y <= y + vy* delta_t
```

delta_t is an experimentally determined change coefficient.

If the user turns on the thrusters, the ship will accelerate with a magnitude obtained by subtracting the drag coefficient from the thrust coefficient. If there is no user input to thrust, the ships continuously decreases its velocity and stops after some time depending on its initial speed. The acceleration during this period is negative and equivalent in magnitude to the drag coefficient.

The calculation of the acceleration of the ship involves a feedback loop as it uses the current velocity of the ship as input.

In order to calculate the x and y components of the velocity, sines and cosines of the current heading are required. To calculate the sine and cosine values, the module trig stores a look up table for all possible angles the ship heading can be and returns the previously calculated sine and cosine values. To minimize the size of the trig look up table, the heading of the ship is quantized by 15 degree increments.

The final implemented design of the ship varies in parts from the proposed solution. The proposed design involved the drawing of complex geometrical shapes and a motion control system with a feedback loop.

In the first couple weeks of the implementation of the ship module, further research into the Bresenham line drawing algorithm revealed emerging complexities with the plan to render the three sides of a triangle for every frame.

A decision was made to implement a prototype version of the ship and gun as circles. Therefore the ship module is composed of two sub-modules, one called saucer and the other gun. As circles are symmetric on all axes, a rotate move does not change the appearance of the saucer. The gun however is mounted in a fixed position on the ship; therefore it rotates with the ship.

The gun module takes as input the center coordinates of the ship, its radius, as well as its current heading. Using these input values, the gun module calculates its center coordinates on the perimeter of the ship, and draws itself at that position.

The proposed motion control system for the ship involved an acceleration variable with a feedback loop. This control mechanism was implemented as a proportional controller with a threshold absolute velocity below which the ship stopped slowing down. The drawback of a proportional controller is the oscillatory behavior that emerges. Such a behavior in addition to an accumulated error term appeared in the first implementation of the ship module. The ship would oscillate when directed to move in a straight line and at high speeds the magnitude of the oscillation diverged and created an unstable system.

After several failed attempts to fine-tune the coefficients, and stabilize the system, the decision was made to abandon this control system and devise a simpler mechanism.

The second implementation of a control mechanism for the ship dejected the feedback system for a simple feed-forward system with no acceleration or drag. The user command to thrust simply increases the velocity of the ship in the appropriate heading, by a delta_vel parameter which was experimentally derived. This parameter was selected to be high enough to give the user the ability to increase velocity from rest to avoid incoming asteroids, and low enough to allow the user to slow down and change directions for evasive action when on the move.

Trigonometry

The Trig module stores a look-up table with sine and cosine values for angle values allowed in the game. The sine and cosine values were calculated in python. In hexadecimal it is not possible to store decimal values between 0 and 1, therefore the real values were multiplied by a scaling factor of 1024. The resulting scale values were converted to a signed hexadecimal value by using the two's complement method.

To recover the original sine and cosine values, all modules which use the trig module must divide their results by 1024 or shift right by 10 bits. This normalization move must be done after the desired calculation has been done.

The test setup for this module involved manual selection of the heading using the switches on the lab kit. A visual inspection of the drawn ship was used to debug errors in the module.

Bullet

The Bullet module controls the movement of an individual bullet and is closely related to the Ship. As this module only controls an individual bullet, there is a higher level module that wraps multiple Bullet modules.

The Ship module passes to the bullet its heading as well as the X and Y positions of its gun. This module also takes in the user input of when the fire button is pressed and a signal from the Game Logic representing whether or not the bullet has been hit.

The life of the bullet is controlled through a two state state-machine. The default state is a Not Visible state. In this state, nothing occurs until the fire button has been pressed. Once this has occurred, the location and heading of the bullet are set and the state changes to a Visible state.

While in the Visible state, the position of the bullet is incremented according to its X and Y velocities upon completion of the frame, as signaled by the horizontal and vertical pixel counts of the VGA. Should the bullet go off of the screen, or receive a high hit signal, the bullet will cease to be drawn and the state will change back to the Not Visible state.

In order to moderate which bullets have been shot and which are to be shot, there is a wrapper module. This module has four instances of the bullet module and passes along the relevant information to the appropriate instances. This module also maintains an index of which bullet is to be fired next, which is updated upon the release of the fire button.

The basic Bullet module was tested thoroughly. The first step was a test module ran in ModelSim to assure that the logic functioned as desired and all the variables updated as expected. Once this passed, the module was tested visually; making sure that the bullet began in the expected position and could move in all of the directions that were currently permissible. Upon integration of the trigonometric look-up table, the module was again tested to be sure it could move correctly in all of the possible directions.

Initially the bullets were to move with a relative velocity of the ship. This feature worked when tested individually; however, when it was integrated with the ship it stopped functioning. Unfortunately, we could not remedy the cause of the undesired behavior and thus the feature was removed.

Asteroid

The Asteroid module is responsible for the creation, size, movement and destruction of a single asteroid. In order to create multiple asteroids, we create multiple Asteroid modules. There are three different sizes of asteroids that can be seen, each with a specific speed.

The randomness of the asteroid is determined by a 12 bit seed variable that is passed to the module. This variable is squared every cycle and used as a pseudorandom number generator. The initial X and Y locations are determined by selecting a range of bits from this variable, 11 bits and 10 bits respectively. The remaining relevant variables that are determined from this variable are as follows: heading (5 bits), size (2 bits), spawn time (4 bits). The heading is passed into a lookup table of trigonometric functions which returns two 12 bit results corresponding to a signed cosine and signed sine result. The spawn time is passed to a timer which will return a signal of one when the designated time has elapsed and a new asteroid will be created. The size is used to determine the size of the asteroid as well as its speed. The larger the asteroid is, the slower it moves. The appropriate speed is multiplied by the results of the trigonometric table to determine the velocities in the X and Y directions.

The actual creation and termination of the asteroid is controlled by a three state state machine. The initial state is a Check state where it assures that the randomness variable is greater than one, as a variable of zero or one will not produce interesting behavior. Should the variable be a zero or one, the variable is arbitrarily set as 17. Here the spawn time is determined from the randomness variable and is arbitrarily set to 4 if the spawn time would be zero. Finally, a start signal is sent to the timer and the state is changed to a Not Visible state.

In this Not Visible state, the start signal is returned to a low value and the initial X and Y positions, heading, and size are determined from the randomness variable. The state then transitions to a Visible state upon an expire signal from the timer going high, stating that the asteroid should now be spawned.

While visible, the position of the asteroid needs to be updated as it moves across the screen. This is done by updating the X and Y positions of the asteroid with the corresponding velocities at the end of each frame, as signaled by the horizontal and vertical counts of the VGA. The position of the asteroid is also monitored to determine if the asteroid has gone off of the screen. This, as well as a high hit signal, will terminate the asteroid, square the randomness variable and change state back to the Check state.

This module was tested visually using the VGA to display the asteroids. The 16 digit display on the FPGA was used to show the current value of the randomness variables, as well as other test variables, and they were monitored to be correct over several generations of asteroids.

Initially, we had intended on implementing asteroids such that the larger asteroids would break off into smaller pieces as opposed to all asteroids being terminated on collision. However, after a base implementation of the asteroid as described would have required a large restructuring of the code. Similarly, the implementation of an animation when the asteroid was destroyed would have been tedious. The number of asteroids was also narrowed down from ten to five as it was more reasonable during game play.

Game Logic

The Game Logic module keeps track of any and all object interactions before sending out information to be displayed. While there are many potential interactions, the only interactions that we are concerned about in our game are those between asteroids and bullets or asteroids and the ship. In the case that an asteroid were to hit a bullet, both are destroyed and the user's score increments. If, instead, the asteroid hits the ship, the game ends. Alternatively, if there are no interactions at the current stage, the game proceeds.

Due to all of the interactions, this module requires an input signal from each of the objects in the game. In order to support this, each object reports a reduction OR of its pixel information for the current positioning of the horizontal and vertical synchronous signals. The colors are designed such that the least significant bit of the color will be a one, thus if the object is currently being drawn the module will receive a one else it will receive a zero. These bits are then used in collision detection.

Internally, the module maintains, for each asteroid, a list that has the same number of bits as there are bullets and ship so that each bit may represent a collision. On the first clock cycle, the bits for the relevant collisions are compared using AND gates and the bit representing the respective collision is updated based on this result. At this point there are multiple bits signaling whether or not each object has been a part of a collision, and since this is a binary possibility, this is inadequate. In order to narrow it down to one bit, all of the bits referring to a specific object are compared using OR gates, thus giving the final result as to whether the object itself has been involved in a collision. These signals are then sent to the appropriate objects.

The module also contains an 8 bit register that maintains the player's score. This is incremented by one when any of the bullets register a hit and is limited to a maximum score of

99. Since the process to determine a hit and remove both the bullet and asteroid from play takes multiple cycles, the hit signal can be high for multiple cycles thus incrementing the score by up to three points. This encourages better accuracy so that the objects may overlap for longer before being eliminated. The score is then sent to the VGA module where it is converted to a string and then displayed on the screen.

In order to test the functionality of this module we wrote a ModelSim test module. In this module we created a wire large enough to accommodate each of the object inputs. Initially this was set to zero, then incremented by one. Upon each increment, we looked to see what inputs were high, and if any collisions occurred, we assured that the appropriate hit signals represented this.

Sound

To implement the sound module, there were two existing solutions that could be adapted for the Asteroids game.

The first was to use a Matlab script written by Yuta Kuboyama which converted .wav files into a .coe containing an audio signal the ac97 can interpret. With this approach, a .wav file of the desired sound effects would be converted and stored on a ROM on the FPGA and played on command. The advantage of this method is the ability to create any sound effect desired, and the potential to replicate the original game sound effects. The drawback is the introduced time investment associated with learning how to use unfamiliar technology and to integrate it seamlessly into the existing system.

The second option was to use a sound effect module written by Eric Fellheimer in 2005 which produced 4 simple sound tones: slash, ramp, triangle and boing. This module worked out of the box and was selected for its ease of use. The drawback is of course the limited number of sound effects.

The sound effect module plays a ramp sound every time the fire button is depressed. There was not enough time to implement a sound effect for collisions or any other interesting interaction.

The sound effect module drives the ac97 components of the lab kit, according to their specs, and restarts the sound for each press of the fire button.

Integration

The individual object modules were implemented and tested independently.

Once the ship and bullet modules were complete, the two were tested together. During this testing phase it was discovered that giving the bullet a relative velocity to the velocity of the ship sometimes caused a register overflow and other unexplainable behaviors. The bullets were then implemented with fixed velocity only.

Once the ship could reliably fire a bullet at stand still, in any direction, and while moving, a module called manybullets was implement to manage multiple bullets. In the proposed game design, 10 bullets were planned. However, 10 bullets turned out to be too many as it would allow the user a continuous stream of bullets which they could shoot while spinning and effectively create a shield. For game play design purposes, it was decided to limit the bullet clip to 4 bullets.

The module manybullets keeps a count of the current bullet_index and fires the next one available in the clip. If all bullets are on screen and have not hit the edge, the player is unable to shoot again until one of the bullets has hit a target or reached the edge of the screen.

Once Asteroids had passed independent testing, it was then added to the game. As game logic had not been integrated all objects could traverse each other.

Game Logic was added in and collisions were tested. A freeze motion switch was added to allow for analysis of a fixed frame using the logic analyzer. The various object pixels and output of the game logic were sent to the logic analyzer to compare inputs, expected outcomes and actual outcomes.

The system did not work at once, but after a rigorous round of methodological testing, the problem was discovered to be one of timing in the collision calculations.

Conclusion

This project was to recreate the game Asteroids on the FPGA board. Initially we were to replicate the game as it was produced, with: a triangle ship, asteroids that splinter, hyperspace, and intricate asteroid shapes. However we did not realize the difficulties that were faced due to the line drawing to form the asteroids and ship. Atari developed a Line Drawing Machine that handled these intricacies. The original game also provided a universe where momentum was not conserved and there was an inherent drag force. We saw this system as a feedback loop to determine the velocity of the using the drag force. Unfortunately, the inclusion of this drag made the system unstable.

The resulting project is a more basic implementation of the game Asteroids and still is amusing to play. However, with a higher analysis of the feedback systems and line drawing techniques, the project could be more similar to the original game. Despite this we still faced interesting design issues such as the randomness of the asteroids and the angular movement of the objects and were able to complete a fun game.

Appendix A

Top Module

```
////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//
////////////////////////////////////////////////////////////////////////////////

module debounce (input reset, clock, noisy,
                output reg clean);

   reg [19:0] count;
   reg new;

   always @(posedge clock)
                if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
                else if (noisy != new) begin new <= noisy; count <= 0; end
                else if (count == 650000) clean <= new;
                else count <= count+1;

endmodule

/////////////////////////////////////////////////////////////////////
//
// 10 hz Counter
//
//////////////////////////////////////////////////////////////////////
module counter10hz( input clk, reset, output reg enable);
  reg [26:0] count;

  always @(posedge clk) begin
     if (reset) count <= 0;
                else if (count == 6500000) begin
                enable <= 1;
                count <=0;
                end
                else begin
                 enable <= 0;
                 count <= count + 1;
                end
  end
endmodule
/////////////////////////////////////////////////////////////////////
//
// 1 hz Counter
//
//////////////////////////////////////////////////////////////////////
module counter1hz( input clk, reset, output reg enable);
  reg [26:0] count;

  always @(posedge clk) begin
     if (reset) count <= 0;
                else if (count == 65000000) begin
                enable <= 1;
```

```
          count <=0;
          end
          else begin
           enable <= 0;
           count <= count + 1;
          end
 end
endmodule


  ////////////////////////////////////////////////////////////////////////////////
  //
  // Asteroid Top Module
  //
  ////////////////////////////////////////////////////////////////////////////////

  // use FPGA's digital clock manager to produce a
  // 65MHz clock (actually 64.8MHz)
  wire clock_65mhz_unbuf,clock_65mhz;
  DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
  // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
  // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
  // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
  // synthesis attribute CLKIN_PERIOD of vclk1 is 37
  BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

  // power-on reset generation
  wire power_on_reset;        // remain high for first 16 clocks
  SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .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 db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button3),.clean(user_reset));
  assign reset = user_reset | power_on_reset;

  // LEFT and RIGHT buttons for rotate
  wire left,right;
  debounce db2(.reset(reset),.clock(clock_65mhz),.noisy(~button_left),.clean(left));
  debounce db3(.reset(reset),.clock(clock_65mhz),.noisy(~button_right),.clean(right));

   // FIRE, THRUST, BRAKE buttons for ship
   wire fire, thrust, brake;
   debounce db4(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(thrust));
   debounce db5(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(brake));
  debounce db6(.reset(reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(fire));

   //play
   wire play_fx;
   debounce db7(.reset(reset),.clock(clock_65mhz),.noisy(~button2),.clean(play_fx));

  // generate basic XVGA video signals
  wire [10:0] hcount;
  wire [9:0]  vcount;
  wire hsync,vsync,blank;
```

```
xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
           .hsync(hsync),.vsync(vsync),.blank(blank));



// feed XVGA signals to ship module
wire [2:0] pixel;
 reg [2:0] frame_pixel;
wire phsync,pvsync,pblank;


 // ship
 wire [2:0] ship_pixel;
 wire [10:0] ship_x, gun_x;
 wire [9:0] ship_y, gun_y;
 wire [11:0] ship_vel_x, ship_vel_y;
 wire [4:0] ship_heading;
 wire ship_hit;
 wire ship_freeze;
ship enterprise(.left_rot(left),.right_rot(right),
                                                                        .fire(fire),.thrust(thrust),.brake(brake),
                                                                        .ship_hit(ship_hit),
                                                                        .vclock(clock_65mhz),.reset(reset),
                                                                        .hcount(hcount),.vcount(vcount),
                                                                        .hsync(hsync),.vsync(vsync),.blank(blank),
                                                                        .phsync(phsync),.pvsync(pvsync),.pblank(pblank),
                                                                        .pixel(ship_pixel),.center_x(ship_x),.center_y(ship_y),
                                                                        .vel_x(ship_vel_x),.vel_y(ship_vel_y),
                                                                        .gun_x(gun_x), .gun_y(gun_y),
                                                                        .heading(ship_heading),
                                                                        .ship_freeze(ship_freeze)
                                                            );



 // bullets
 wire [2:0] bullet0_pixel,bullet1_pixel;
 wire [2:0] bullet2_pixel,bullet3_pixel;
 reg [2:0] bullets_pixel;
 wire [3:0] bullets_hit;
 wire [1:0] bullet_index;
 manybullets ak47(.fire(fire && ~ship_freeze),.ship_heading(ship_heading),
                                                .lead_vertex_x(gun_x),.lead_vertex_y(gun_y),
                                                .clock(clock_65mhz),.reset(reset),.bullets_hit(bullets_hit),
                                                .ship_x_speed(ship_vel_x),.ship_y_speed(ship_vel_y),
                                                .hcount(hcount),.vcount(vcount),
                                                .bullet0_pixel(bullet0_pixel),
                                                .bullet1_pixel(bullet1_pixel),
                                                .bullet2_pixel(bullet2_pixel),
                                                .bullet3_pixel(bullet3_pixel),
                                                .switch(switch[2])),
                                                .bullet_index(bullet_index)
                                                );

// asteroids
 reg [2:0] asteroids_pixel;
 wire [2:0] a0_pixel;
 wire a0_hit;
 reg [11:0] seed0;
asteroid a0(.hit(a0_hit),.reset(reset),.clock(clock_65mhz),
                                         .hcount(hcount),.vcount(vcount),.square_in(seed0),
                                         .pixel(a0_pixel), .switch(switch[3]));
 wire [2:0] a1_pixel;
 wire a1_hit;
 reg [11:0] seed1;
asteroid a1(.hit(a1_hit),.reset(reset),.clock(clock_65mhz),
                                         .hcount(hcount),.vcount(vcount),.square_in(seed1),
                                         .pixel(a1_pixel), .switch(switch[3]));
 wire [2:0] a2_pixel;
 wire a2_hit;
 reg [11:0] seed2;
asteroid a2(.hit(a2_hit),.reset(reset),.clock(clock_65mhz),
                                         .hcount(hcount),.vcount(vcount),.square_in(seed2),
                                         .pixel(a2_pixel), .switch(switch[3]));
 wire [2:0] a3_pixel;
 wire a3_hit;
 reg [11:0] seed3;
asteroid a3(.hit(a3_hit),.reset(reset),.clock(clock_65mhz),
                                         .hcount(hcount),.vcount(vcount),.square_in(seed3),
                                         .pixel(a3_pixel), .switch(switch[3]));
 wire [2:0] a4_pixel;
 wire a4_hit;
 reg [11:0] seed4;
asteroid a4(.hit(a4_hit),.reset(reset),.clock(clock_65mhz),
                                         .hcount(hcount),.vcount(vcount),.square_in(seed4),
                                         .pixel(a4_pixel), .switch(switch[3]));

 // sound
 reg [1:0] sound_fx;
 sfx sound(.reset(reset),.clock_27mhz(clock_27mhz),
                               .audio_reset_b(audio_reset_b),
                               .ac97_sdata_out(ac97_sdata_out),
                               .ac97_sdata_in(ac97_sdata_in),
                               .ac97_synch(ac97_synch),
                               .ac97_bit_clock(ac97_bit_clock),
                               .mode(2'b01),.play(fire));


 // game logic
 wire as1, as2, as3, as4, as0, b1, b2, b3, b0, s;
 wire[7:0] points;
 Logic collision(.clock(clock_65mhz), .reset(reset),
                                                    .asteroid1(|a0_pixel),.asteroid2(|a1_pixel),
                                                    .asteroid3(|a2_pixel),.asteroid4(|a3_pixel),
                                                    .asteroid5(|a4_pixel),
                                                    .bullet1(|bullet0_pixel),.bullet2(|bullet1_pixel),
                                                    .bullet3(|bullet2_pixel),.bullet4(|bullet3_pixel),
                                                    .ship(|ship_pixel),
                                                    .hitA1(a0_hit),.hitA2(a1_hit),
                                                    .hitA3(a2_hit),.hitA4(a3_hit), .hitA5(a4_hit),
                                                .hitB1(bullets_hit[0]),.hitB2(bullets_hit[1]),
                                                .hitB3(bullets_hit[2]),.hitB4(bullets_hit[3])
                                                .hitS(ship_hit), .score(points));

 //char string displays
 reg[2:0] char_pixel;
 wire [2:0] score_pixel,points_pixel, gameover_pixel;
 wire [15:0] points_str;
 bin2str pointsascii (.clock(clock_65mhz),.binary(points),.string(points_str));
 char_string_display pointsdisp(.vclock(clock_65mhz), .hcount(hcount), .vcount(vcount), .cstring(points_str),
                                                          .cx(11'd202), .cy(10'd20), .pixel(points_pixel));
 wire [151:0] score_str = "ASTEROIDS - score: ";
 char_string_display scoredisp(.vclock(clock_65mhz), .hcount(hcount), .vcount(vcount), .cstring(score_str),
                                                          .cx(11'd10), .cy(10'd20), .pixel(score_pixel));
 defparam          scoredisp.NCHAR = 19;
defparam           scoredisp.NCHAR_BITS = 5;

 wire [71:0] gameover_str = "GAME OVER";
 wire [71:0] not_gameover_str = "         ";
 wire [71:0] game_str = (ship_freeze)?gameover_str:not_gameover_str;
 char_string_display gameoverdisp(.vclock(clock_65mhz), .hcount(hcount), .vcount(vcount), .cstring(game_str),
                                                          .cx(11'd412), .cy(10'd343), .pixel(gameover_pixel));
 defparam          gameoverdisp.NCHAR = 9;
defparam           gameoverdisp.NCHAR_BITS = 4;


 assign pixel = frame_pixel;
 display_16hex debug1(.reset(reset), .clock_27mhz(clock_27mhz),
                                                .data({57'b0, points}),
                                                .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));

 assign analyzer1_data ={|(a0_pixel),|(a1_pixel),|(a2_pixel),
                                                |(a3_pixel),|(a4_pixel),|(bullet0_pixel),
                                                |bullet1_pixel,|(bullet2_pixel),
                                                |bullet3_pixel,|ship_pixel,
                                                a0_hit,a1_hit,a2_hit,
                                                bullets_hit[0],bullets_hit[1],
                                                ship_hit} ;
assign analyzer1_clock = clock_65mhz;

 assign analyzer3_data = {phsync, pvsync,pblank, 5'b0};
 assign analyzer3_clock = clock_65mhz;

 //assign led = ~{a0_hit, a1_hit,a2_hit,bullets_hit[0], bullets_hit[1],
        //                        bullets_hit[2],bullets_hit[3],ship_hit};
 reg a0hitled, a1hitled,a2hitled, b0hitled,b1hitled,b2hitled,b3hitled,shiphitled;
 assign led = ~{a0hitled, a1hitled,a2hitled, b0hitled,b1hitled,b2hitled,b3hitled,shiphitled};

 initial begin
 seed0 = 6;
 seed1 = 8;
 seed2 = 12;
 seed3 = 14;
 seed4 = 18;
 sound_fx = 3;
 end

 // switch[1:0] selects which video generator to use:
//  00: user's game
//  01: 1 pixel outline of active video area (adjust screen controls)
//  10: color bars
reg [2:0] rgb;
reg b,hs,vs;
always @(posedge clock_65mhz) 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==1023 | vcount==0 | vcount==767) ? 7 : 0;
             end else if (switch[1:0] == 2'b10) begin
  // color bars
  hs <= hsync;
  vs <= vsync;
  b <= blank;
  rgb <= hcount[8:6];
             end else begin
             // default: Asteroids
 if (reset) begin
              a0hitled <= 0;
              a1hitled <= 0;
              a2hitled <= 0;
              b0hitled <= 0;
              b1hitled <= 0;
              b2hitled <= 0;
              b3hitled <= 0;
              shiphitled <= 0;
              end
 else if (a0_hit) a0hitled <= 1;
 else if (a1_hit) a1hitled <= 1;
 else if (a2_hit) a2hitled <= 1;
 else if (bullets_hit[0]) b0hitled <= 1;
 else if (bullets_hit[1]) b1hitled <= 1;
 else if (bullets_hit[2]) b2hitled <= 1;
 else if (bullets_hit[3]) b3hitled <= 1;
 else if (ship_hit) shiphitled <= 1;
 else begin
              a0hitled <= a0hitled;
              a1hitled <= a1hitled;
              a2hitled <= a2hitled;
              b0hitled <= b0hitled;
              b1hitled <= b1hitled;
              b2hitled <= b2hitled;
              b3hitled <= b3hitled;
              shiphitled <= shiphitled;
              end


  sound_fx[1:0] <= switch[7:6];
  char_pixel <= score_pixel|points_pixel|gameover_pixel;
```

```
    bullets_pixel <= bullet0_pixel|bullet1_pixel|bullet2_pixel|bullet3_pixel;
    asteroids_pixel <= a0_pixel|a1_pixel|a2_pixel|a3_pixel|a4_pixel;
    frame_pixel <= ship_pixel | bullets_pixel | asteroids_pixel | char_pixel;

    hs <= phsync;
    vs <= pvsync;
    b <= pblank;
    rgb <= pixel;
          end
  end


  // VGA Output.  In order to meet the setup and hold times of the
  // AD7125, we send it ~clock_65mhz.
  assign vga_out_red = {8{rgb[2]}};
  assign vga_out_green = {8{rgb[1]}};
  assign vga_out_blue = {8{rgb[0]}};
  assign vga_out_sync_b = 1'b1;          // not used
  assign vga_out_blank_b = ~b;
  assign vga_out_pixel_clock = ~clock_65mhz;
  assign vga_out_hsync = hs;
  assign vga_out_vsync = vs;


endmodule

////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
////////////////////////////////////////////////////////////////////////////////

module xvga(input vclock,
            output reg [10:0] hcount,        // pixel number on current line
            output reg [9:0] vcount,     // line number
            output reg vsync,hsync,blank);

  // horizontal: 1344 pixels total
  // display 1024 pixels per line
  reg hblank,vblank;
  wire hsyncon,hsyncoff,hreset,hblankon;
  assign hblankon = (hcount == 1023);
  assign hsyncon = (hcount == 1047);
  assign hsyncoff = (hcount == 1183);
  assign hreset = (hcount == 1343);

  // vertical: 806 lines total
  // display 768 lines
  wire vsyncon,vsyncoff,vreset,vblankon;
  assign vblankon = hreset & (vcount == 767);
  assign vsyncon = hreset & (vcount == 776);
  assign vsyncoff = hreset & (vcount == 782);
  assign vreset = hreset & (vcount == 805);

  // sync and blanking
  wire next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
          hcount <= hreset ? 0 : hcount + 1;
          hblank <= next_hblank;
          hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

          vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
          vblank <= next_vblank;
          vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

          blank <= next_vblank | (next_hblank & ~hreset);
  end
endmodule
```

Ship Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: 
// Engineer: 
// 
// Create Date:    11:39:32 11/10/2011 
// Design Name: 
// Module Name:    ship 
// Project Name:    Asteroids
//
//////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
//
// gun: draws a circle on the outer edge of ship to mark gun turret
//
////////////////////////////////////////////////////////////////////////////////
module gun
   #(parameter RADIUS = 6,
                                                           COLOR = 3'b011)
   (input [10:0] ship_x_center, hcount,
    input [9:0] ship_y_center,vcount,
    input [4:0] ship_radius,heading,
    input clk,
    output reg [2:0] pixel,
    output [10:0] gun_x,
    output [9:0] gun_y);

    reg [21:0] r_sq;
    reg signed [11:0] x_dist, x_center, y_center;
    reg signed [10:0] y_dist;
    reg signed [23:0] x_delta;
    reg signed [23:0] y_delta;
    reg [21:0] x_sq;
    reg [21:0] y_sq;
    reg [2:0] pixelbuff1;

    assign gun_x = x_center[10:0];
    assign gun_y = y_center [9:0];

    wire signed [11:0] cos, sin;
    trig gun_trig(.clock(clk),.heading(heading),.signedcos(cos),.signedsin(sin));

    always @(*) begin
    r_sq = RADIUS*RADIUS;
    x_delta = ship_radius*cos;
    y_delta = ship_radius*sin;
    //divide by 10 bits to remove cos and sin scaling
    x_center = ship_x_center + x_delta[23:10];
    y_center = ship_y_center + y_delta[23:10];
    end

    always @(posedge clk) begin
            pixel <= pixelbuff1;
```

```
                x_dist <= hcount - x_center;
                y_dist <= vcount - y_center;
                x_sq <= x_dist*x_dist;
                y_sq <= y_dist*y_dist;
                if ((x_sq+y_sq) <= r_sq) pixelbuff1 <= COLOR;
                else pixelbuff1 <= 0;
                end

endmodule

//////////////////////////////////////////////////////////////////////
//
// saucer: generate pixels for circle on screen
//              parameters: center x,y coordinates, radius and color
//
//////////////////////////////////////////////////////////////////////

module saucer
    #(parameter RADIUS = 18)
    (input [10:0] x_center, hcount,
     input [9:0] y_center,vcount,
     input clk,
     output reg [2:0] pixel);

     wire [21:0] r_sq;
     reg signed [11:0] x_dist;
     reg signed [10:0] y_dist;
     reg [21:0] x_sq;
     reg [21:0] y_sq;
     reg [2:0] pixelbuff1;

    assign r_sq = RADIUS*RADIUS;

    always @(posedge clk) begin
                pixel <= pixelbuff1;
                x_dist <= hcount - x_center;
                y_dist <= vcount - y_center;
                x_sq <= x_dist*x_dist;
                y_sq <= y_dist*y_dist;
                if ((x_sq+y_sq) <= r_sq) pixelbuff1 <= 3'b001;
                else pixelbuff1 <= 0;
    end
endmodule



/////////////////////////////////////////////////////////
//
// Ship
//
/////////////////////////////////////////////////////////
module ship(
    input left_rot,
    input right_rot,
    input thrust,
    input brake,
    input fire,
    input ship_hit,

    //input clock27mhz,
    input vclock,    // 65MHz 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,          // XVGA horizontal sync signal (active low)
    input vsync,          // XVGA vertical sync signal (active low)
    input blank,          // XVGA blanking (1 means output black pixel)
    output phsync,    // ship horizontal sync
    output pvsync,    // ship vertical sync
    output pblank,    // ship blanking
    output [2:0] pixel,                // ship  pixel

    output reg [10:0] center_x,
    output reg [9:0] center_y,
    output reg signed [11:0]  vel_x,
    output reg signed [11:0]  vel_y,
    output [10:0] gun_x,
    output [9:0] gun_y,
    output [4:0] heading,
    output ship_freeze
              );

    //reg signed [5:0] acc, thrust_coeff, drag_coeff;
    reg signed [5:0] delta_t;
    wire signed [11:0] shipcos, shipsin;
    reg signed [23:0] vel_xdelta, vel_ydelta;
    reg [2:0] frame_pixel;
    reg [4:0] curr_heading;

    reg thrust_en, brake_en,fire_en;

    reg tenhzenbuff1, tenhzenbuff2;
    reg tenhzenbuff3, tenhzenbuff4;
    reg tenhzenbuff5, tenhzenbuff6;

    //reg [7:0] vel_xbuff1, vel_ybuff1;

    reg hsyncbuff1, vsyncbuff1, blankbuff1;
    reg hsyncbuff2, vsyncbuff2, blankbuff2;

    reg [12:0] velsq;
    reg [12:0] delta_tsq;



    wire [2:0] ship_pixel;
    saucer ufo(.x_center(center_x),.hcount(hcount),.y_center(center_y),.vcount(vcount),
                                              .clk(vclock),.pixel(ship_pixel));
    wire [2:0] gun_pixel;
    gun turret(.ship_x_center(center_x),.hcount(hcount),.ship_y_center(center_y),.vcount(vcount),
                                              .ship_radius(5'd18),.heading(curr_heading),.clk(vclock),.pixel(gun_pixel),
                                              .gun_x(gun_x),.gun_y(gun_y));
```

```
trig ship_trig(.clock(vclock),.heading(curr_heading),.signedcos(shipcos),.signedsin(shipsin));
wire tenhzen;
counter10hz tenhz(.clk(vclock),.reset(reset),.enable(tenhzen));

assign phsync = hsyncbuff2;
assign pvsync = vsyncbuff2;
assign pblank = blankbuff2;
assign pixel = frame_pixel;

assign heading = curr_heading;

parameter S_FREEZE = 1;
parameter S_MOVE = 0;
reg state;

assign ship_freeze = state;

initial begin
center_x = 500;
center_y = 300;
delta_t =8;
vel_x = 0;
vel_y = 0;
end

always @(*) begin
velsq = (vel_x*vel_x)+(vel_y*vel_y);
delta_tsq = delta_t*delta_t;

end

always @(posedge vclock) begin
        // pipeline hsync,vsync and blank
        hsyncbuff2 <= hsyncbuff1;
        hsyncbuff1 <= hsync;
        vsyncbuff2 <= vsyncbuff1;
        vsyncbuff1 <= vsync;
        blankbuff2 <= blankbuff1;
        blankbuff1 <= blank;

        // buffer the tenhzen signal
        // to pipeline movement operations
        tenhzenbuff6<=tenhzenbuff5;
        tenhzenbuff5<=tenhzenbuff4;
        tenhzenbuff4<=tenhzenbuff3;
        tenhzenbuff3<=tenhzenbuff2;
        tenhzenbuff2<=tenhzenbuff1;
        tenhzenbuff1<=tenhzen;


        frame_pixel <= ship_pixel | gun_pixel;

        if (reset) begin
                center_x <= 500;
                center_y <= 300;
```

```
    vel_x <= 0;
    vel_y <= 0;
    vel_xdelta <= 0;
    vel_ydelta <= 0;
    state <= S_MOVE;
        end

// transition to Frozen state if ship is hit
if (ship_hit) state <= S_FREEZE;

case(state)
    // Moving the Ship
    S_MOVE: begin
    if (tenhzen) begin
        // poll button at 10hz and
        //increment/decrement heading
        case({left_rot,right_rot})
            2'b00: curr_heading <= curr_heading;
            2'b01:    begin
                if (curr_heading==0) curr_heading <= 23;
                else curr_heading <= curr_heading -1;
                        end
            2'b10:    begin
                if (curr_heading==23) curr_heading <= 0;
                else curr_heading <= curr_heading +1;
                        end
            2'b11: curr_heading <= curr_heading;
            default: curr_heading <= curr_heading;
        endcase
    end
    if (tenhzenbuff2) begin
        // divide by 10 bits to remove trig scaling
        vel_xdelta <=(thrust)? shipcos*delta_t:0;
        vel_ydelta <=(thrust)? (shipsin*delta_t):0;

    end
    if (tenhzenbuff4) begin
        vel_x <= vel_x + vel_xdelta[23:10];
        vel_y <= vel_y + vel_ydelta[23:10];
    end
    if (tenhzenbuff6) begin
        if (center_x>=1000) center_x <= 33;
        else if (center_x<=23) center_x <= 990;
        else center_x <= center_x + vel_x*delta_t;
        if (center_y>=744) center_y <= 33;
        else if (center_y<=23) center_y <= 734;
        else center_y <= center_y + vel_y*delta_t;
            end
    end
    // Freeze the ship
    S_FREEZE: begin
        if (tenhzenbuff6) begin
        center_x <= center_x;
        center_y <= center_y;
        end
            end     endcase    end endmodule
```

Game Logic Module

```
module Logic(
        input clock,
        input reset,
        //lots of inputs, all of which are 1 bit
        input asteroid1,
        input asteroid2,
        input asteroid3,
        input asteroid4,
        input asteroid5,

        input bullet1,
        input bullet2,
        input bullet3,
        input bullet4,

        input ship,

        //lots of outs too
        output reg hitA1,
        output reg hitA2,
        output reg hitA3,
        output reg hitA4,
        output reg hitA5,

        output reg hitB1,
        output reg hitB2,
        output reg hitB3,
        output reg hitB4,

        output reg hitS,
        output [7:0] score
);

        //an array of collisions for each asteroid
        reg[4:0] collision1;
        reg[4:0] collision2;
        reg[4:0] collision3;
        reg[4:0] collision4;
        reg[4:0] collision5;

        //score_0 is ones digit, score_1 is tens digit
        reg [3:0] score_0, score_1;

        initial begin
                collision1 = 0;
                collision2 = 0;
                collision3 = 0;
                collision4 = 0;
                collision5 = 0;
                score_0 = 0;
                score_1 = 0;
                end

        always @(posedge clock) begin
                if(reset) begin
                    score_0 <= 0;
                    score_1 <= 0;
                    end

                //check asteroid 1 with everything
                collision1[0] <= (asteroid1 && bullet1) ? 1 : 0;
                collision1[1] <= (asteroid1 && bullet2) ? 1 : 0;
                collision1[2] <= (asteroid1 && bullet3) ? 1 : 0;
                collision1[3] <= (asteroid1 && bullet4) ? 1 : 0;
                collision1[4] <= (asteroid1 && ship) ? 1 : 0;

                //asteroid 2
                collision2[0] <= (asteroid2 && bullet1) ? 1 : 0;
                collision2[1] <= (asteroid2 && bullet2) ? 1 : 0;
                collision2[2] <= (asteroid2 && bullet3) ? 1 : 0;
                collision2[3] <= (asteroid2 && bullet4) ? 1 : 0;
                collision2[4] <= (asteroid2 && ship) ? 1 : 0;

                //asteroid 3
                collision3[0] <= (asteroid3 && bullet1) ? 1 : 0;
                collision3[1] <= (asteroid3 && bullet2) ? 1 : 0;
                collision3[2] <= (asteroid3 && bullet3) ? 1 : 0;
                collision3[3] <= (asteroid3 && bullet4) ? 1 : 0;
                collision3[4] <= (asteroid3 && ship) ? 1 : 0;

                //...4
                collision4[0] <= (asteroid4 && bullet1) ? 1 : 0;
                collision4[1] <= (asteroid4 && bullet2) ? 1 : 0;
                collision4[2] <= (asteroid4 && bullet3) ? 1 : 0;
                collision4[3] <= (asteroid4 && bullet4) ? 1 : 0;
                collision4[4] <= (asteroid4 && ship) ? 1 : 0;

                //...5
                collision5[0] <= (asteroid5 && bullet1) ? 1 : 0;
                collision5[1] <= (asteroid5 && bullet2) ? 1 : 0;
                collision5[2] <= (asteroid5 && bullet3) ? 1 : 0;
                collision5[3] <= (asteroid5 && bullet4) ? 1 : 0;
                collision5[4] <= (asteroid5 && ship) ? 1 : 0;

                //cool now do the specific hits

                //hitting asteroids
                hitA1 <= |(collision1);
                hitA2 <= |(collision2);
                hitA3 <= |(collision3);
                hitA4 <= |(collision4);
                hitA5 <= |(collision5);

                //hitting bullets
                hitB1  <= (collision1[0] | collision2[0] | collision3[0] | collision4[0] | collision5[0]);// |  collision6[0] | collision7[0] |
collision8[0] | collision9[0] | collision10[0]);
                hitB2  <= (collision1[1] | collision2[1] | collision3[1] | collision4[1] | collision5[1]);// | collision6[1] | collision7[1] |
collision8[1] | collision9[1] | collision10[1]);
```

```
    hitB3  <= (collision1[2] | collision2[2] | collision3[2] | collision4[2] | collision5[2]);// | collision6[2] | collision7[2] |
collision8[2] | collision9[2] | collision10[2]);
    hitB4  <= (collision1[3] | collision2[3] | collision3[3] | collision4[3] | collision5[3]);// | collision6[3] | collision7[3] |
collision8[3] | collision9[3] | collision10[3]);

    //hitting the ship
    hitS <= (collision1[4] | collision2[4] | collision3[4] | collision4[4] | collision5[4]);

    //update the score on collisions
    if(hitB1 | hitB2 | hitB3 | hitB4) begin
        //check to see if next digit should be 0
        if(score_0 == 9) begin

            //check to see if you are in the 90s already
            if(score_1 == 9) begin

                //if you are here the score is 99...stay
                score_1 <= score_1;
                score_0 <= score_0;
                end

            //if you are here you are incrementing the next tens digit
            else begin
                score_0 <= 0;
                score_1 <= score_1 + 1;
                end
            end

        //here just increment the ones digit
        else score_0 <= score_0 + 1;
        end
    end

    //concat score to be tens and ones
    assign score = {score_1,score_0};
endmodule
```

Bullet Module

```
module bullet #(parameter Color = 3'b001)(
   input button_fire,        //fire!!
   input[4:0] ship_heading,    //heading of the ship
   input[10:0] lead_vertex_x,    //leading x of the ship
   input[9:0] lead_vertex_y,    //leading y of the ship
   input clock,            //...clock
   input reset,            //...reset
   input hit,            //lets you know if the bullet got hit
   input signed[11:0] ship_x_speed,    //x speed of ship
   input signed[11:0] ship_y_speed,    //y speed of ship
   input[10:0] hcount,        //the hcount appears again
   input[9:0] vcount,        //as does mr vcount
   input switch,                                                   //killswitch!

   output reg[2:0] pixel        //pixel
   );

   parameter Screen_Height = 767;
   parameter Screen_Width = 1023;
   parameter Width = 20;
   parameter S_Not = 0; parameter S_Visible = 1;
   reg state;
   reg[10:0] x;
   reg[9:0] y;

   //speeds are internal copies of the ship speed so if u speed up it doesnt effect the bullet
   reg signed[11:0] x_speed, y_speed;

   //the deltas represent the difference in position when considering the TRIG
   reg signed[23:0] x_delta, y_delta;

   reg signed[3:0] Speed;

   //outputs from lookup table
   wire signed[11:0] cos, sin;

   initial begin
         Speed = 4;
         state = 0;
         x = 0;
         y = 0;
         x_delta = 0;
         y_delta = 0;
         end

   //give the table the clock and the heading, receive cos and sin values
   trig t(.clock(clock), .heading(ship_heading), .signedcos(cos), .signedsin(sin));

 //state transitions
always @(posedge clock) begin
            case(state)
                        //when you get a fire button, set the relevant variables
                        S_Not:     if(button_fire) begin
                                                //start point of bullet
```

```
                                x <= lead_vertex_x;
                                y <= lead_vertex_y;

                                //heading info
                                x_delta <= Speed * cos;
                                y_delta <= Speed * sin;

                                //switch states
                                state <= S_Visible;
                                end

                S_Visible:      //if you get hit, or the bullet goes off the screen then go to not visible

                                        if((hit) || (x >= Screen_Width) || (y >= Screen_Height) ) state <= S_Not;
                        default:        state <= S_Not;
                        endcase

                if (reset) begin
                        //defaults
                        state <= 0;
                        x <= 0;
                        y <= 0;
                        x_speed <= 0;
                        y_speed <= 0;
                        x_delta <= 0;
                        y_delta <= 0;
                        end

                //update the position of the bullet if it is visible and after the last frame finished
                else if((hcount == Screen_Width) && (vcount == Screen_Height) && (state == S_Visible)) begin

                        //due to the fact that the speeds and deltas are signed, simply add them to the current position
                        if(~switch) begin
                                x <= x + x_delta[23:10];// + x_speed;
                                y <= y + y_delta[23:10];// + y_speed;
                                end
                        end
                end

  //handle shaping
  always @(*) begin
    //if you are INSIDE the square make it yellow
    if (((hcount >= x) && (hcount <= (x + Width))) &&
      ((vcount >= y) && (vcount <= (y + Width))) && (state == S_Visible))
      pixel = Color;
    else pixel = 0;
  end
endmodule
```

Asteroid Module

```
module asteroid(
            input hit,                                                  //letting the Asteroid know if it has been hit
            input reset,
            input clock,                                                //clock
            input[10:0] hcount,  //count of horizontal pixels
            input[9:0] vcount,              //count of vertical pixels
            input[11:0] square_in,          //term to be squared
            input switch,                                               //debug tool to set velocity to zero

            output reg[2:0] pixel                       //color of pixel
   );

            parameter Screen_Width = 1023; parameter Screen_Height = 767;
            parameter S_Not = 0; parameter S_Visible = 1; parameter S_Check = 2;
            parameter Small = 0; parameter Medium = 1; parameter Big = 2;
            parameter Color = 7;

            //have signed speed so that you can go either direction
            reg signed[13:0] x_speed;
            reg signed[13:0] y_speed;
            reg[2:0] size;
            reg[5:0] asteroid;
            reg[10:0] x_pos;
            reg[9:0] y_pos;
            reg signed[23:0] x_delta, y_delta;
            reg signed[2:0] speed_s;
            reg signed[3:0] speed_m;
            reg signed[4:0] speed_f;
            reg[4:0] heading;
            reg[3:0] value;
            reg start;
            wire enable, expire;
            wire signed[11:0] cos, sin;
            reg[11:0] square;
            reg [1:0] state;

            initial begin
                        //def don't want these signed!
                        x_pos = 0;
                        y_pos = 0;
                        state =0;
                        speed_s = 3;
                        speed_m = 4;
                        speed_f = 6;
                        start = 0;
                        end

            trig t(.clock(clock), .heading(heading), .signedcos(cos), .signedsin(sin));

            //use counter and timer to count up some second value until the asteroid respawns
            counter c(.clk(clock), .reset(reset), .enable(enable));
            timer timer(.clock(clock), .enable(enable), .start(start), .reset(reset), .value(value), .expire(expire));

            always @(posedge clock) begin
```

```
		case(state)
				//if you aren't visible you will be right here
				S_Not:			begin
								start <= 0;
								//create some new value of parameters from z
								x_pos <= square[10:0];
								y_pos <= square[11:2];
								heading <= square[5:1];
								size <= {square[8],square[4]};

								//if it is time to be visible again go to visible
								if (expire) begin
									state <= S_Visible;
									end
								end

				//if you are visible then go here
				S_Visible:				//if you are hit...you are done, also if u go off the screen
								if((hit) || (x_pos >= Screen_Width) ||  (y_pos >=
Screen_Height)) begin
								//change state
								state <= S_Check;
								square <= square * square;
								end

				//this state assures that the square term and the spawn time are not zero
				S_Check:begin
						if(square <= 1) square <= 17;
								if(square[3:0] == 0) value <= 4;
								else value <= square[3:0];
								start <= 1;
								state <= S_Not;
								end
				endcase

		if(reset) begin
			//if reset then go to defaults
			state <= 2;
			square <= square_in;
			if(square_in[3:0] == 0) value <= 4;
			else value <= square_in[3:0];
			end

		else if((hcount == Screen_Width) && (vcount == Screen_Height) && (state == S_Visible)) begin
			//update the position with the velocity
			//switch is used in debugging to prevent movement
			if(~switch) begin
				x_pos <= x_pos + x_speed;
				y_pos <= y_pos + y_speed;
				end
			end
		end

	//handle shaping
	always @(*) begin
		//if you are INSIDE the square make it white
```

```
                if (((hcount >= x_pos) && (hcount <= (x_pos + asteroid))) &&
                        ((vcount >= y_pos) && (vcount <= (y_pos + asteroid))) && (state == S_Visible))
                        pixel = Color;
                else pixel = 0;
        end


        //whenever you get a new size you will be here
        always @(posedge clock) begin
                case(size)
                        //set the speed based on the size of the asteroid
                        Small:  begin
                                //use the cosine and sine to get the effective change of x and y
                                x_delta <= speed_f * cos;
                                y_delta <= speed_f * sin;

                                //divide delta by 1024
                                x_speed <= x_delta[23:10];
                                y_speed <= y_delta[23:10];

                                //set the asteroid size
                                asteroid <= 15;
                                end

                        Medium: begin
                                x_delta <= speed_m * cos;
                                y_delta <= speed_m * sin;

                                x_speed <= x_delta[23:10];
                                y_speed <= y_delta[23:10];

                                asteroid <= 31;
                                end

                        Big:    begin
                                x_delta <= speed_s * cos;
                                y_delta <= speed_s * sin;

                                x_speed <= x_delta[23:10];
                                y_speed <= y_delta[23:10];

                                asteroid <= 63;
                                end

                        default: begin
                                x_delta <= speed_m * cos;
                                y_delta <= speed_m * sin;

                                x_speed <= x_delta[23:10];
                                y_speed <= y_delta[23:10];

                                asteroid <= 31;
                                end
                        endcase
                end
endmodule
```

Appendix B

References:

Sound Module:
http://web.mit.edu/6.111/www/f2005/code/sfx.v

Character Display:
http://web.mit.edu/6.111/www/f2005/code/cstringdisp.v
http://web.mit.edu/6.111/www/f2005/code/font_rom.v
http://web.mit.edu/6.111/www/f2005/code/font_rom.ngo