Vertex

Don Goldin
Mark Sullivan, III

Abstract

The goal for this project was to create a game loosely inspired by *Asteroids*, or, more recently, *Geometry Wars*. The player's avatar is a ship, with which the player has two primary ways of interacting. The ship is able to move and shoot, but is able to do so in independent directions. Movement and shooting will each be accomplished through joysticks communicating with the labkit. The player's goal is to survive as long as possible while a variety of enemy ships, each type having its own simple AI, spawn and try to destroy the avatar.

Graphically, the goal is to create visuals reminiscent of vector graphics. That is, all in-game objects would be treated as 2D wireframes. This should permit an interesting aesthetic experience as well as permitting transformations such as scaling and rotation of game objects with a fair amount of ease compared to sprite based graphics.

Table of Contents

List of Figures

1 Overview

Asteroids is name many people associate with the dawn of gaming. The player has the ability to spin, shoot, and accelerate forward. The player's goal is to avoid colliding with the asteroids on the playing field. Shooting an asteroid caused it to break apart into several smaller ones. Video games have evolved a lot since then, and the ideas of *Asteroids* have been all but abandoned.

As with most classic games, *Asteroids* remakes are in no short supply. However, one comparatively recent game has successfully expanded on this formula. This game is called *Geometry Wars*, appropriately subtitled *Retro Evolved*. Similar to *Asteroids*, the player controls a ship, is able to shoot at any angle, and must avoid being hit by on-screen obstacles. Also, a 2D wireframe art style mimics the true vector graphics of the original. There are several differences, however, which distinguish *Geometry Wars.* The player is able to move and shoot at independent angles. The player's input controls the player's velocity, instead of in the original where the input controlled the force and the ship kept its momentum. All enemies have different AI, whereas in the original most "enemies" were merely floating asteroids. This fresh take on an old formula was greeted by respectably high sales and reviews. Many of our ideas mirror the implementation of *Geometry Wars*, so that is the best existing game to use to get an idea of what this project aimed to become.

On a high level, three main modules were developed. The first stage is the input stage; this module implements the serial peripheral interface (SPI) protocol to communicate with each of two joysticks. The input is updated every game clock, which runs at 30 frames per second. Based on information received from the joysticks, it passes on several pieces of information. The first is the x and y position of the left joystick; this is to be used for movement. The second piece of information is the angle of the right joystick. Since this module receives x and y from the joystick, it uses an additional arctangent module to deduce the angle from this information. Finally, this module also reports the state of the right joystick button (that is, the button triggered by pushing in the stick slightly).

The second stage of information flow is the game module. This is where all of the game logic occurs. It maintains a memory of the positions, orientations, and states of all units (or 'entities') in the game. This module typically goes through several states every frame. First, it detects if the player is trying to use their 'bomb' item; if they are able to, it destroys all enemies. Next, it decides based on the output of the random number generator if it should spawn a new enemy somewhere on screen. Then, it creates three bullets at the avatar's position every eighth frame, and these bullets travel in the direction dictated by the second joystick. The next stage is the movement stage. Each entity gets its position, orientation, and state updated in this stage. Finally, collision detection takes place. Using distance based collisions, relevant collisions are checked for, and colliding objects are then each destroyed. The relevant information about each entity is made available to the graphics module via a block RAM.

The graphics module is also divided into several pipeline stage. The first stage, the shape module, looks up each entity's data; for each entity, it looks up the line segments comprising the entity's model and applies translation and rotation to the endpoints to get them to the right position. Each line segment is then passed onto the Bresenham module, which uses Jack Bresenham's line algorithm to determine which pixels are on that line. Those pixels are then sent to the buffer module, which utilizes double-buffering with the labkit's two ZBT RAMs to allow random-access drawing to the screen without flickering.

A few other minor modules included the ramclock module used to keep the two ZBTs synchronized, and the SVGA module which controlled the timing of the video output signals.

2 Description

<u>2.1 Interface Module</u> (Mark Sullivan)

The purpose of the interface module is to provide input and output to the user's controller. The game logic requires the x and y coordinates of one joystick for movement, the angle of the second joystick for shooting, and a button press for when a bomb is deployed. This module is also responsible for rumble output to the controller.

2.1.1 Input Module

The input module communicates with the joysticks using the serial peripheral interface mode 0 protocol. The joysticks used are the Digilent PmodJSTK, shown below.

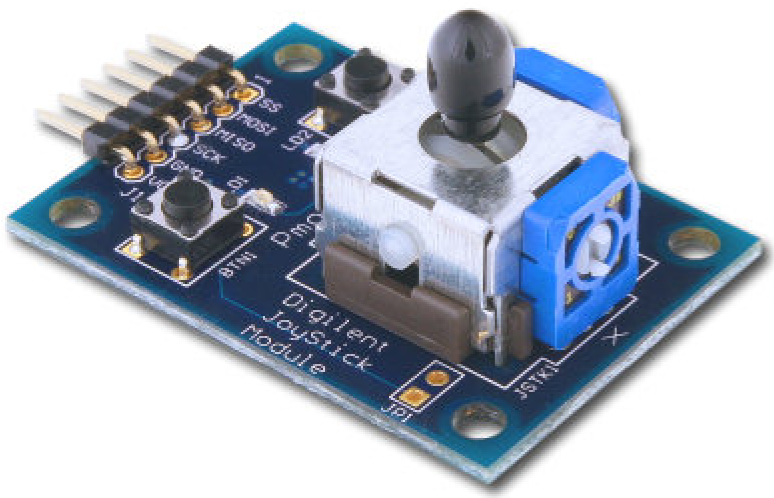

Figure 1: PmodJSTK used for user input

This particular joystick has several additional constraints which required consideration. Information was transmitted in a sequence of eight bytes. Positional information is sent as a 10 bit value for each axis, zeroed at the bottom-left corner. The first byte is the lower order x bits, while the high order x bits are the two low order bits of the second byte. For transmitting the y position, similarly the low order bits are in the third byte while the high order bits are in the fourth byte. The fifth byte communicates the state of the buttons. Between each byte, the joystick expects a 10 microsecond delay before transmitting the next byte. It also expects a 15 microsecond delay between when the slave select signal goes low and when the first byte is transmitted. Also, it can be clocked at a maximum of 1MHz.

Both joysticks are communicated with in parallel. Communication begins on the positive edge of the vsync signal. Since game logic starts at the negative edge, the game is done using the arctangent module by this point, so the input module will have access to it. The protocol was implemented using a state machine with three states, S_IDLE, S_TRANSMIT, S_PAUSE. In the idle state, the module waits for the next clock edge. In the transmit state, a set of registers counts the clock cycles to produce a 1MHz clock, and data is read in at every positive edge. Finally, in the pause state, the module waits 15 microseconds before transitioning to transmit. Typical communication is shown below, taken from ModelSim.

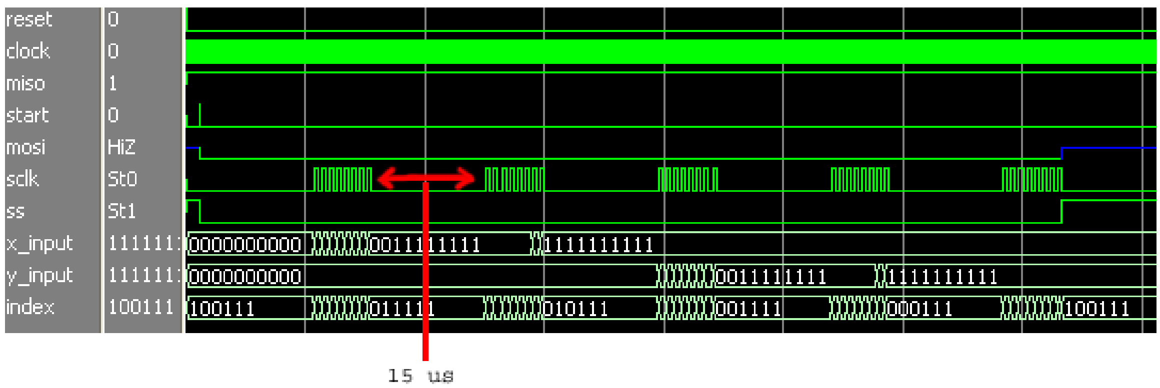

Figure 2: Implementation of SPI protocol

The block diagram specifying the inputs and outputs of this module are shown below.

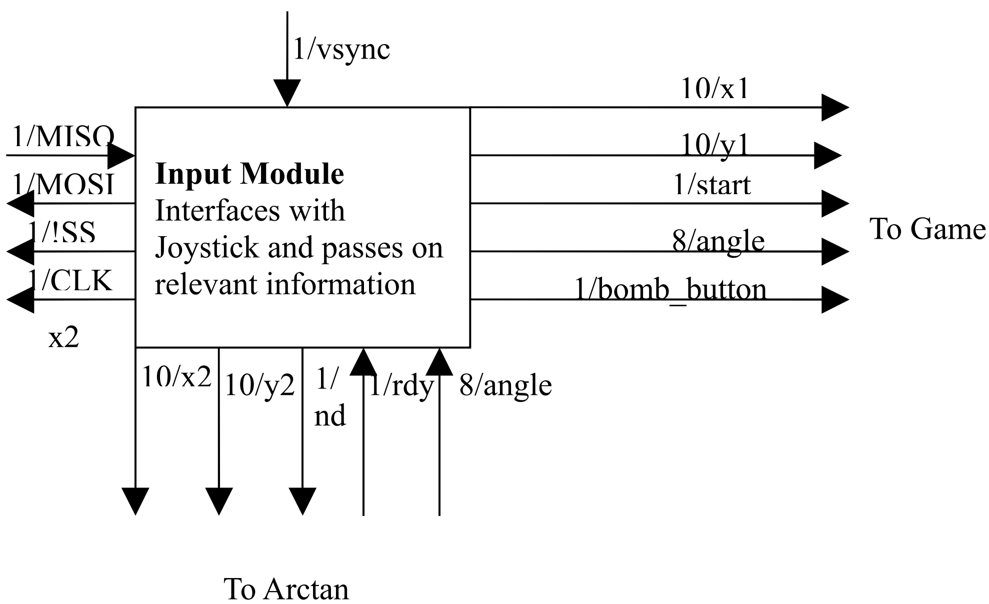

Figure 3: Block Diagram of Input Module

Since the joysticks are fairly uncomfortable to hold in one's hand, they were mounted inside the shell Nintendo Gamecube controller. The buttons are purely aesthetic, although the rumble functionality was preserved, and is used in game. Since we require a button press to activate a bomb in the game, we use the button activated by a joystick press on the rightmost joystick.

Figure 4: Modified Nintendo Gamecube controller, with new joysticks and working rumble

2.1.2 Rumble Module

This is a very simple module. It is responsible for receiving a pulse from the game module, and generating a high signal for one second. This signal is sufficient to drive a small motor which was put inside the controller. The motor swings around an uneven mass, which causes the shaking effect. It is used to give feedback when the player is hit by an enemy, or when the player deploys a bomb.

2.2 Game Module (Mark Sullivan)

This module is responsible for maintaining data of all of the units in the game. The game supports up to 256 simultaneous units. The information for each unit is stored in several BRAMs. The unit information is stored in a standardized way: each has "public" and "private" information. Public information is shared with the game module. This information is stored in 32 bits and has the following form:

0001	0000000001	0000000001	00000001
[31:28] ID	[27:18] x	[17:8] y	[7:0] angle

The ID is the objects type. This is used by the game module to determine its behavior, and by the graphics module to determine its graphical representation. X and Y are integer representations of the center of the object. The angle represents the rotation of the object. We adopted the convention that zero meant right facing, and positive angles rotated clockwise. This represents a scaled angle, such that an angle of theta corresponds to theta/256 * 2 * pi radians.

“Private” information is not shared with the graphics module. There are also 32 bits reserved for each unit’s private information.

000001_000001_00000000000000000001
[31:26] x_precision [25:20] y_precision [19:0] state

Precision bits are fixed point extensions of the integer x and y values and are used to permit smoother movement. Without this, non integer speeds would not be as straightforward, and this is often the case with angular movement. The state bits aren’t as strictly defined as everything else. The function often varies by unit. For instance, some units use these bits as a timer, while others use them to store velocity.

Since the graphics module needs access to the BRAM, a way was needed to coordinate these memory accesses, as we had been using single port BRAMs. In our implementation, we used four BRAMs total. There was one which stored private information. One stored public information, and was only used by the game. The other two also contained the public information, and they were used as so: One BRAM was read from by the graphics module on even game frames, and written to by the game module on the odd ones. The same information is written to it as is written to the game module’s personal public information BRAM. The other BRAM behaves symmetrically on opposite cycles. This effectively lags visual feedback by one frame, but it was found to be imperceptible during game play.

The main hub of the game module is the game logic module. Two of the more complicated tasks, movement and collision detection, have dedicated modules to take care of that processing. Also, the arctangent module is shared with the input module. The arctangent module communicates with the movement module when vsync is low, and this module will have plenty of time to complete before it rises. A block diagram of the game module follows.

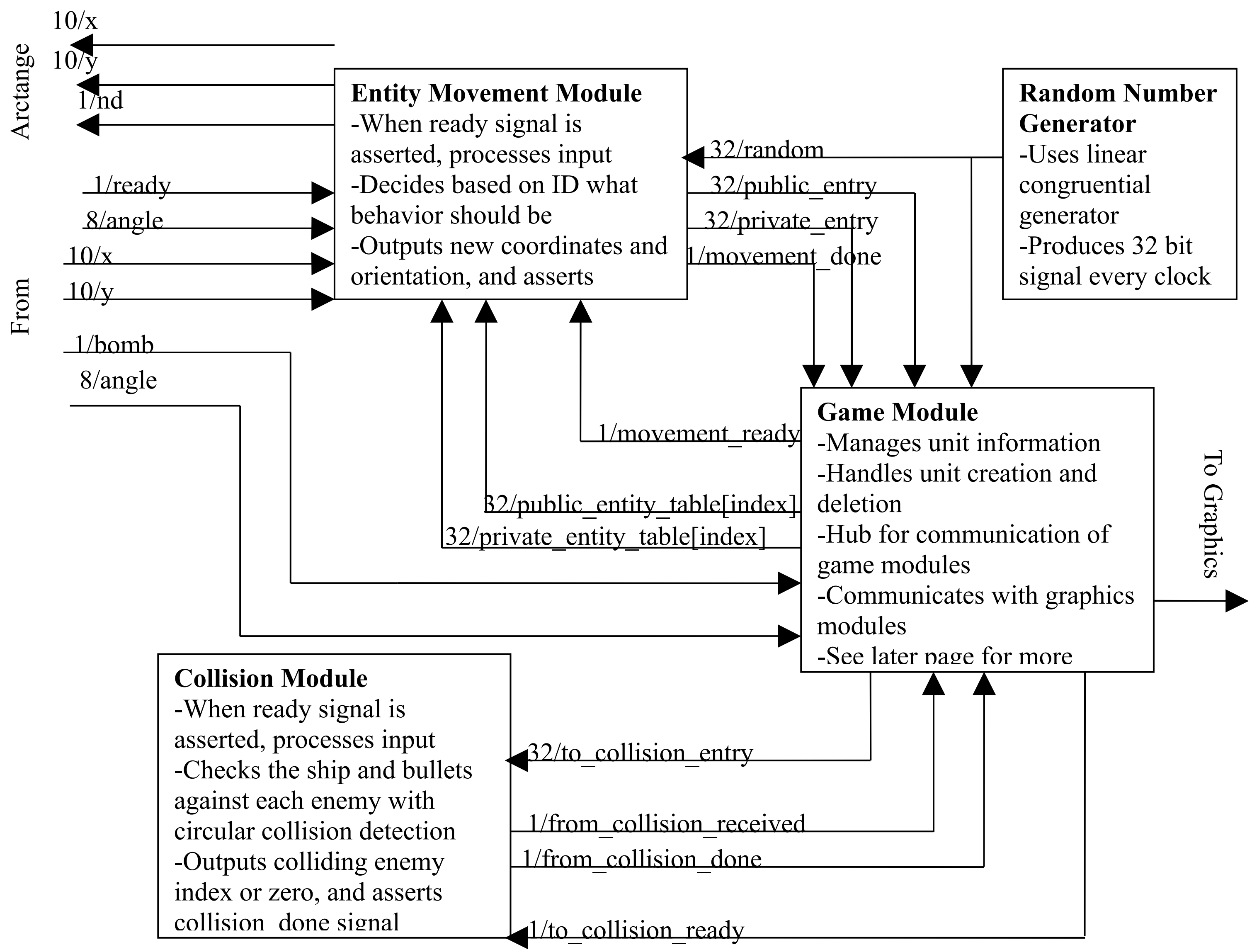

Figure 5: Block Diagram of Game Module

2.2.1 Game Logic Module

The game logic module is the main state machine of the game module as a whole. It is the only module with direct read/write access to the BRAMs containing the entity data. Usually, it linearly goes through several major states: Bombing, enemy spawning, bullet spawning, moving, and colliding. For simplicity, especially in collision checks, this module follows the standard that the first 128 entities must not be enemies, while the last 128 entities may only be enemies.

When the game is reset, the memory is initialized. The avatar is created at memory location 0. The next 6 elements are part of the HUD (heads-up display, the static part of the screen dedicated to conveying information to the player). The first three objects added are the representation of bombs, put in the top left. The next three are remaining lives, put in the top right.

Every frame, the first step is to see if the player has used a bomb. They are permitted to do so if they have not used all of their lives, and if they have remaining bombs. The bomb signal received from player input is a pulse. If this did happen, we enter the S_BOMB_REMOVE state. When this happens, the graphical representation of the bomb is erased, and the bomb count is decremented. The address of the bomb to be removed is already known, since the three bombs are in addresses 8'd0, 1, 2. The next objective of the bomb is to turn any enemies with non-zero ID into explosions. Therefore, the game must alternate between reading and writing. Since only enemies need to be checked, the game starts searching at address 8'd128. Due to the fact that BRAMs are clocked, and we are alternating between S_BOMB_READ, and S_BOMB_WRITE, a read queued up in the read state will be available on the following read. Therefore, the goal of the read state is to queue up the next address and to decide what will get written on the next step. If there is an enemy there, it loads an explosion at that position onto the BRAM's memory in registers. Otherwise, it just loads zero, which means that nothing is there. Either way, the write state will write what is currently on memory in into memory. Because of the delay between read and write, the addresses need to be changed a lot. In particular, the address increases by two on every read, and decrements on every write. This creates an efficient flow of reads and writes so that there doesn't need to be any waiting time in between.

The next set of states corresponds to spawning. The first state is S_DECIDE_SPAWN. This state consults the random number generator to decide if it should create an enemy or not. It will attempt to create a new enemy every second on average. If it does decide to spawn, the module enters the S_SPAWNING state. Here, it searches through the enemy indices until it finds an unoccupied slot. It then creates a 'spawn' unit at a random location. The spawn unit can't be collided with, so it can't kill you unfairly by appearing at your location. The state bits of this unit dictate which unit it will turn into after some amount of time has passed.

The next set of states corresponds to shooting, and they are very similar to the spawning states. The major difference is that the bullet firing occurs at a deterministic rate, with no random influence. The number of frames since a bullet has been fired is checked, and every fourth frame a bullet is spawned; this is checked in the S_DECIDE_SHOOT state. Then, should the game need to shoot, it enters the S_SHOOTING state. The first 128 addresses are iterated through until an open spot is found, and then the bullet is created at the player's position and at the orientation received from the aiming joystick by the input module. There are no checks for if the player has been destroyed. This is because once the player is destroyed, this null object is moved off screen, so any created bullets will be instantly annihilated.

The next step is the movement stage. This term is perhaps not completely indicative of what this stage is capable of. While units will be moved, they also update their state and may even transform into different units. In any case, most of that logic occurs in the movement module, and these states are dedicated to communicated with it and updating the memory based on information received. First, address initialization is done in the S_PREMOVING state. The next state, S_MOVING, iterates through all in game units and sends them one by one to the movement module. First, an address is loaded up for transmission to the movement module. Once this address appears, it is accompanied by a "movement_ready" pulse. The game logic module then waits until it received a "movement_done" pulse. The arrival of the movement_done signal indicates that the public and private information from the last transmission are available. This information is then stored in memory, and this process then repeats with the next address. A sample of typical communication is shown below.

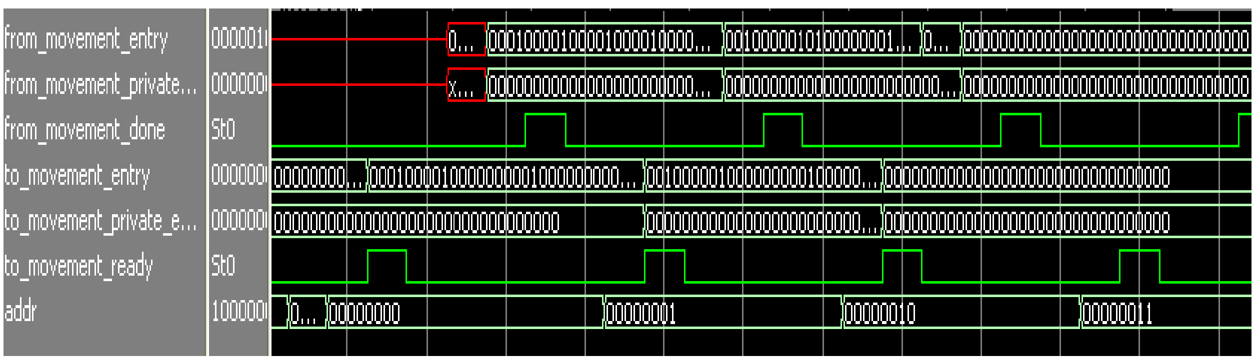

Figure 6: Communication with movement module

The final step is collision processing, and it is the most complex part of the game logic module. There are two types of collisions in the game, player into enemy and bullet into enemy. Since we have partitioned the address space into non-enemies and enemies, we check those entries in the first 128 memory positions against those in the next 128, doing a maximum of 128x128 collision checks. To establish a terminology convention, the non-enemy is referred to as the primary object, and the enemy is referred to as the secondary object. The communication protocol in the S_COLLIDING state is as follows: First, a primary object is sent to the collision module, along with a "ready" signal. The collision module will need access to all of the enemies in the game. To this end, after the ready signal is sent, the first enemy is put on the to_collision_entry wires. They will remain there until the collision module asserts the from_collision_received signal, indicating that it has received the currently sent enemy and the game is free to start loading the next one. If the collision module asserts its done signal, the S_COLLISION_PROCESSING state is entered. This signal can be received for any one of several reasons. If the primary entity was a null object, then no further collisions needed to be checked. Sometimes it reports invalid collisions, such as those with spawner objects discussed before, for which no collision should take place. Otherwise, it's usually a valid collision, and both objects are then destroyed. If one of the collided objects is the player, then the player's lives are decremented. The screen is then cleared of enemies in the S_AVATAR_COLLIDED state, unless all of the player's lives are spent, in which case the player is destroyed. After processing a collision, the next primary object is sent until all have been processed. A sample of typical communication is shown below; in this case, no collision occurs.

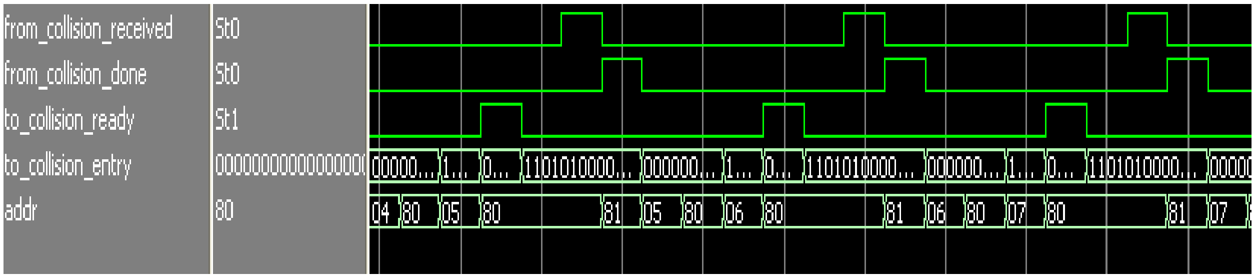

Figure 7: Communication with collision module

2.2.2 Movement Module

The movement module is responsible for updating each unit once per frame. It handles one unit at a time, and the processing has several stages. Upon receiving the 'ready' signal from the game logic module, it stores the public and private entity data. From there, many of the decisions made depend on the ID of the entity. Each one, however, has several options. The entity may choose whether it wants to attempt Cartesian or polar movement. If it chooses Cartesian, it specifies a (signed) delta x and delta y by which it wants to change. This delta x and delta y are fixed point, with the implied decimal following the 6 low order bits. If it chooses polar, it must specify a magnitude and it will attempt to move along its current angle. The object is also free to specify a new ID, a new angle, and a new state. It may also request use of the arctangent module, specifying atan_x and atan_y, which will then determine its new orientation. After the proposed position is established, it is checked to see if it falls within the boundaries of the game (a 600x800 area). If it extends too far in either direction, a corrected_x or corrected_y signal will be asserted, and the new x or y value will be overridden by this. Finally, if arctangent use was requested, the module waits for the arctan module's ready signal, and then the movement module asserts its own 'done' signal with the new public and private entries ready.

Angles were all scaled radians. Several trigonometric modules were used here. The arctangent module was relied upon, as were sine and cosine modules. Sine and cosine were look-up tables generated by CoreGen. Angles are 8 bit values, as are sine and cosine values, although these are signed. The sine and cosine values were used any time a unit attempted polar movement, and then the appropriate fixed point multiplication took place.

Each unit has its own unique behavior. The enumeration for IDs is as follows:

NO_ID	4'h0
AVATAR	4'h1
A_BULLET	4'h2
BOMB	4'h3
EXTRA_LIFE	4'h4
SPAWN	4'h5
EXPLOSION_0	4'h6
EXPLOSION_1	4'h7
MIMIC	4'h8
SATELLITE	4'h9
BROWN	4'hA
PONG	4'hB
CHASER	4'hC
SPIKE	4'hD
CHEVRON	4'hE
FISH	4'hF

Figure 8: ID Convention

Objects of type NO_ID don't move at all. They are null objects, and can be treated as available memory. Two other units are entirely idle, the BOMB and EXTRA_LIFE. These objects exist purely for the HUD, so they have no need to move. As an additional note, all of these objects are ignored for collisions.

Objects of type AVATAR, of which there should only be one, move about using Cartesian movement. The delta x and delta y are specified by the joystick input. It uses arctangent to have a rotation which matches is current delta x and delta y. The player unit is special, so the movement module remembers the public entry of the player.

Objects of type A_BULLET are bullets the avatar has fired. They have purely polar movement, and move in a straight line, whose angle is set when the bullet is spawned. The state bits are used as a timer, although at the final speed parameters chosen make it such that the timer doesn't expire, but the code was left in if plans changed. If it ever reaches any of the screen borders, the bullet destroys itself by changing its id to NO_ID.

Objects of type SPAWN don't move at all. They spin in place by decreasing their angle by two every frame. The six lower order state bits are a countdown timer. Once this timer expires, this unit becomes whatever its top four state bits are, randomly selected upon its creation.

EXPLOSION_0 and EXPLOSION_1 are frames of an explosion effect. The objects themselves are stationary and do not rotate. The state bits are used as a count down timer. EXPLOSION_0 is created upon a bullet's collision with the enemy. The second frame, EXPLOSION_1, appears after some time, before it itself destroys itself after some amount of time.

Objects of type MIMIC respond inversely to the player input. That is, if the player moves up, they move down, and the same holds horizontally. They also get the reverse orientation of the player; they don't use arctangent themselves for this, they rely on the movement module's stored value. They have no state.

Objects of type SATELLITE circle around the avatar. Using the arctangent module, they move in a polar fashion around the avatar counter clockwise. They have a fixed speed and no state.

Objects of type BROWN move around randomly. Using output from the random number generator, they sometimes decide to change their velocity. Every step, the BROWN sets its delta x and delta y to the velocity values it has stored in its 10 lower order bits, 5 for each component. This velocity is sign extended to permit omnidirectional movement.

Objects of type PONG move diagonally and bounce off of the walls. The two lower order state bits dictate the direction the unit travels in. A value of 1 for the lowest order bit moves the unit rightwards, while a value of one for the next bit moves the unit downwards. Zeros move them up or left, respectively.

Objects of type CHASER actively pursue the avatar. They move using polar coordinates towards the avatar. To accomplish this, use of the arctangent module is required. They have no additional state.

Objects of type SPIKE move in small circles. They don't actively pursue the player. They use polar coordinates to move, and increase their angle linearly every frame to achieve the circular motion.

Objects of type CHEVRON have two states. In the first state, they aim. They use arctangent to always face the player. Once some time has passed, they enter the charging state. They no longer change their angle, but they move quickly towards the spot where the player was when they finished aiming.

Objects of type FISH move in an apparently sinusoidal manner across the screen. They move using polar coordinates. They start by decrementing their angle every frame. Any time a 45 degree angle is reached in some quadrant, they change between incrementing and decrementing. The net result if the implementation is a wavy sequence of arcs horizontally across the screen. If the fish reaches the edge of the screen, it changes direction.

2.2.3 Collision Module

The collision module reports collision detections to the game module. To restate the terminology, the non enemy object is the 'primary' object, which is then checked against all of the enemies, 'secondary' objects. The collision detection process begins when the module receives a ready signal from the game. The module latches onto the primary object which has just been transferred. After this, the module receives the secondary object and asserts its 'received' signal. This gives the game module enough time to load the next object by the time the collision module needs it. If either a collision is found or the primary object is null, the done signal is asserted. The game logic module will be able decide how to handle it.

The collision checks themselves are simply distance based. The module has a table of the radius-squared of all of the entities. It then goes through several pipelined stages, with the goal of computing if $(x_{primary} - x_{secondary})^2 + (y_{primary} - y_{secondary})^2 < (r_{primary} + r_{secondary})^2$.

2.2.4 Random Number Module

The random number generator is used for all random events in the game, most notably spawning. The random number generator itself is a linear congruential generator. For a given starting value, the next value it outputs is:

$$X_{t+1} = AX_t + C \pmod{M}$$

This generator, while simple, was sufficient for our purposes. Default values are those used by Borland C/C++ rand(). In particular, M in this case is 2^{32}, so the modulus is accomplished by discarding the higher order bits, and thus a 32 bit value is produced.

2.2.5 Arctangent Module

The arctangent computation itself is done in a CoreGen module. This module was designed to take in an 11 bit x and y value, and produce a 10 bit angle. However, we only need to feed it 10 bit x and y values, and require an 8 bit angle. The arctangent module was restricted to input values between -1 and 1, so this needed to be padded with an extra bit. The angle, similarly, became what was expected if we discard the top two order bits.

The arctangent module is actually shared between the movement module and the input module. When vsync is low, it interacts with the movement module. When it is high, it does so with the input module. The movement module has plenty of time to complete before vsync rises, so sharing in this way was not an issue.

2.3 Graphics Module (Don Goldin)

The graphics module behaves more or less as a linear pipeline. On one end it communicates with the game module, sending requests for the public data (see above) and receiving the data back. On the other end, it produces a video signal which goes directly out to the VGA interface. It also interfaces with the FPGA's ZBT chips and produces the 60hz "game clock" signal that synchronizes the framerate. It is broken into a few submodules which are described below.

2.3.1 Shape Module

The shape module is the first module in the graphics pipeline. For each of the 256 possible entities, it sends a requested index to the game module and receives the public data back. This data is split into four variables: `entity_id`, `entity_x`, `entity_y`, and `entity_theta`. With that data, it iterates through the entity's line segments (of which there are up to 16), looking up the start and end points. Some entity types and segments return an ignore signal, in which case the shape module skips to the next segment. The lookup table contains data something like this:

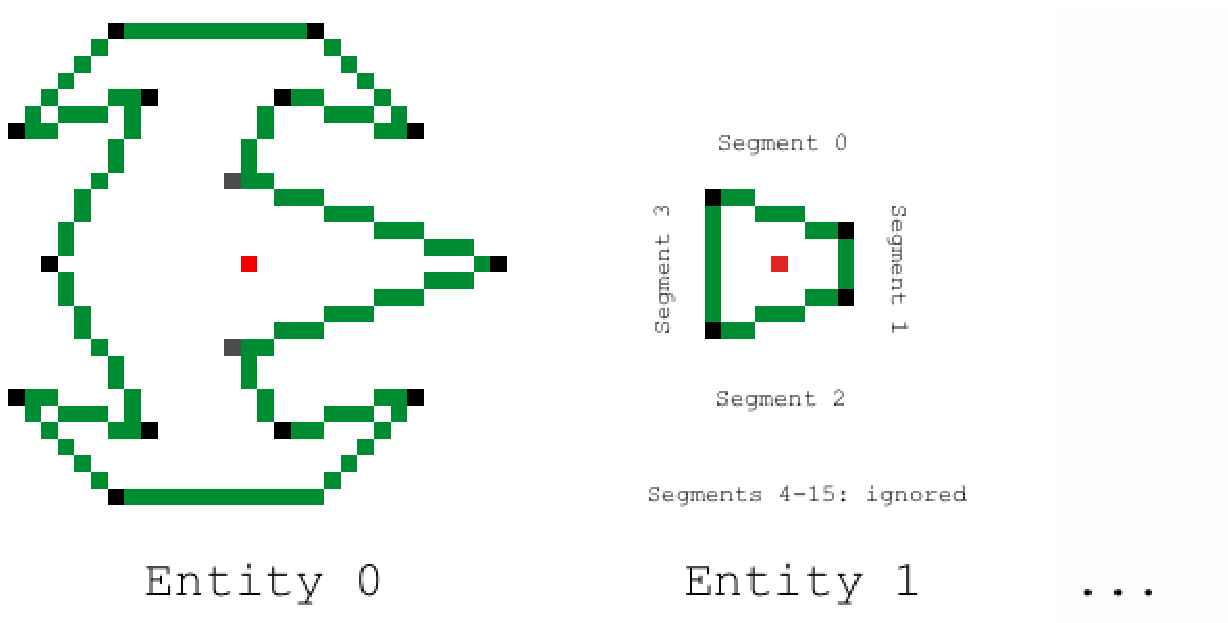

Figure 9: Illustration of shape module's lookup table contents.
NOTE: not an exact or complete representation

If the segment is not to be ignored, then the shape module rotates its endpoints about the local origin (represented by the red dot) by `entity_theta` using the sine and cosine outputs of a CoreGen-provided trigonometry module. The rotated endpoints are added to `entity_x` and `entity_y` and the result is output to the Bresenham module along with a signal indicating that the shape module is ready. The table also contains color data for each segment, which is passed along through the Bresenham module as well.

2.3.2 Bresenham Module

The Bresenham module is so named because it implements Jack Bresenham's line drawing algorithm, described at http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm. This algorithm draws a straight line between two points; it is a particularly good choice because its implementation does not require any division. The module's inputs are the two endpoints and color of the line segment it will draw, as well as a ready signal from the shape module. Its outputs are the colored pixels to be written to the offscreen buffer, a write_enable signal to the buffer, and a ready signal back to the shape module.

The timing of the ready signals is as follows. When the Bresenham module is not in the process of drawing a segment, it asserts its ready signal. When the shape module sees this, it proceeds to iterate through entities and segments until it finds one that is not ignored. Once it finds one, it sends out the segment's data along with a one-cycle long ready signal. Upon receiving this, the Bresenham module takes down its ready signal until it finishes processing the segment.

2.3.3 Double-buffer module

The double-buffer module allows random-access pixels to be displayed to the screen without needing huge arrays of combinational logic and without causing flicker due to erased pixels. It physically interfaces with the Virtex2's two ZBT memories, clearing and then writing to one while the other is displayed on-screen. This module is synchronized with the SVGA module's hcount, vcount, and vsync signals and operates on a two-frame period, bounded by vsync. During the first frame of the period, hcount and vcount are used as indices to the off-screen buffer and the value at each address is set to zero. During the second frame, the module accepts pixel coordinates and color values from the Bresenham module and writes them to the off-screen buffer. During both frames, hcount and vcount also index the on-screen buffer, the contents of which are sent to the VGA interface. Upon the rising edge of the vsync signal, the off-screen buffer becomes on-screen and vice versa.

2.4 Timing Modules (Don Goldin)

Two small extra modules were required in order to keep everything on the labkit synchronized. The first is the ramclock module, which is a slight modification of the module of the same name available from the course website. Due to the different locations of the ZBT memories, the timings of the clock signals required to operate them are slightly different. This module uses clock feedback to compensate for that problem. The other is the SVGA module, which is a modified version of lab 5's XVGA module. Instead of taking a 65mhz clock and producing 1024x768x60hz video timing, it uses a 40mhz clock to produce 800x600x60hz video.

3 Testing and Debugging

(Mark Sullivan)

Nearly all initial development was done in ModelSim. Given the nature of our project, this tool was convenient, since testing of the game couldn't be done well graphically for a large portion of the project. The game module had about three weeks of development. One week was dedicated to each of the three major constituent modules: the game logic module, movement module, and collision module. In general, one of these modules was developed and debugged per week, and then tested in lab. Test benches were written to evaluate these modules. One test bench was used to test the three major modules, since often their integration was the key. Another test bench was written for the input module, and another for the random number generator.

For testing in lab, a simple graphics module was developed to evaluate performance of these modules on the labkit. This module was a modified version of code from lab five. It was capable of drawing one unit at a time, and it was represented as a white square. Since there were 256 possible entities, the address could be selected using the labkit switches. The LEDs were used to display the binary representation of the unit type. While simple, it was sufficient for evaluation of nearly every component of the game. The most difficult part to check with this module was the collision detection. To do this, random spawning was deactivated, and the player's ship collided into pre-positioned stationary objects.

There was only one major inconsistency between simulation and practice. The collision module seemed to destroy everything when on the labkit, despite working perfectly in simulation. It turned out that this wasn't a problem with the labkit. It was a problem with the test bench. In simulation, the collision module was being prematurely terminated by the start of the next game cycle. However, the bug appeared when the collision module was allowed to finish. Once the simulation was made to accurately reflect what was happening, the problem was easily fixed.

The testing for the input module was much more readily done in lab. For one thing, the communication protocol was tested in ModelSim, but that was contingent on whether or not the SPI protocol was properly understood, so it could only really be tested in lab. Also, there was some trouble getting arctangent working in simulation, so the angular tests associated with input were better done in lab.

(Don Goldin)

Unlike the game logic and input modules, the graphics modules had a direct and visible output which could be verified by eye. All of the graphics testing was done in lab using the labkit's own video display.

The first module built and tested was the SVGA module. As in lab 5, a couple different test signals were prepared: a box outlining the 800x600 viewable area, a colored horizontal gradient, and a two-by-two pixel checkerboard pattern. Once these signals were working properly, they were reproduced in the double-buffer module in order to test it.

The video produced by the double-buffer module was somewhat glitchy at first: there were bands of color visible as well as flickering video. To determine the cause of the problem, the alternating buffer functionality was temporarily replaced by a switch which controlled which was displayed. It soon became evident that one of the buffers was producing the correct output while the other was not. Gim and Alex eventually suggested that this was due to a timing problem with the ZBT's clocks. Inverting those clock signals worked as a temporary fix, but further timing issues appeared down the line.

Testing the Bresenham and shape modules was relatively simple. For Bresenham, a stub was created to give it the endpoints of a segment to draw, moving each frame. Once the module was properly implemented, the video showed the expected segment. For the shape module, some of the labkit's switches were used to select which type of entity to draw and another stub controlled its position and rotation. At this point, some confusing discrepancies appeared between the expected and actual output. After much head-scratching and rebuilding, we figured out that once again the problem was caused by the ZBT clock timing. This prompted us to add in the ramclock module, which removed that bug once and for all.

One final bug appeared during the final integration of the project: a single line segment in one of the entities was not drawing to the screen. This turned out to be because the combinational delay of the multiple lookup tables and arithmetic operations between the game logic and shape modules had pushed the ignore bit past the threshold for the next cycle's setup time. A one-cycle delay was added in the shape module to allow the signals to stabilize.

4 Conclusion

All initial goals were met. Joystick support was successfully added. The game module keeps a memory of all in game units, and provides them in a meaningful format to the graphics module. Units randomly spawn in the game. Every frame, the positions and rotations of all of the units are updated. Colliding objects are detected, and collisions are handled by creating an explosion at the site of the collision. Many objects, each composed of multiple rotated line segments, can be displayed to screen at a time. The video outputs at a smooth 30hz, with no visible glitches.

A few extra features were added as well. The bomb item, which destroys all enemies on screen, was not originally planned for. The accompanying rumble feature was not planned for as well. Multiple lives were added, as was a graphical representation of lives and bombs. Score, which appears on the hex display, was added.

Due to the slew of features implemented, this project can be considered successful. However, there is always room for improvement. From the game end of things, there were two extra ideas for which there was no time to implement. One of these was the scaling of difficulty with time. Another unimplemented feature is the ability for spawning to happen in waves. That is, instead of enemies randomly spawning one at a time, several enemies might spawn in a coordinated fashion. For graphics, anti-aliasing and alpha blending were considered but never implemented. Anti-aliasing turned out to be impractical with the Bresenham algorithm, and introducing alpha blending would have significantly reduced the available throughput of the ZBTs, since it would take an additional two cycles to read the old value and further time to compute the blending.

Aside from the technical features called for in our design, Vertex also met our original goal, which was to create a game similar to *Geometry Wars*. The end result was fun to implement, fun to play, and well received.

Figure 10: Vertex in action!

A Appendices

A.1 Top Level Verilog

```
`default_nettype none

///////////////////////////////////////////////////////////////////////////////
//
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes, 6.111 staff
//
///////////////////////////////////////////////////////////////////////////////
//

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
);

  ///////////////////////////////////////////////////////////////////////////
/
  //
  // 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.
  //
  ///////////////////////////////////////////////////////////////////////////
/


   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;

   // 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;
```

```
      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));

   // 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_r
eset));
   assign reset = user_reset | power_on_reset | !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;

   // 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;

      // clock enable (should be synchronous and one cycle high at a time)
   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]   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;

      wire ss;
      wire sclk;
      wire mosi;
      wire miso1;
      wire miso2;
      wire rumble_voltage;

      //Interface with joysticks
      assign user3[31] = ss;
      assign user3[30] = mosi;
      assign miso1 = user3[29];
      assign user3[28] = sclk;
      assign user3[27] = ss;
      assign user3[26] = mosi;
      assign miso2 = user3[25];
      assign user3[24] = sclk;
      assign user3[23] = rumble_voltage;

      //Score
      wire [15:0] score;

      game_module
gm0(.clock(vclock),.reset(reset),.vsync(vga_out_vsync),.miso1(miso1),.miso2(m
iso2),.ss(ss),.mosi(mosi),
            .rumble_voltage(rumble_voltage),
.graphics_addr(entity_index[7:0]),.sclk(sclk),.score(score),.to_graphics_entr
y(entity_data[31:0]));


      vertex_graphics vg (.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));


      assign analyzer3_clock = vclock;
      assign analyzer3_data =
{entity_index[7:0],entity_data[31:28],2'b1,vga_out_vsync,vvsync};

      display_16hex
hd0(.reset(reset),.clock_27mhz(clock_27mhz),.data({48'b0,score[15:0]}),
            .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));


endmodule
```

A.2 Game Verilog

```
`timescale 1ns / 1ps
`default_nettype none
///////////////////////////////////////////////////////////////////////////////
/////
// Company:
// Engineer: Mark Sullivan
//
// Create Date:    16:20:06 12/03/2008
// Design Name:
// Module Name:    game_module
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
//////////////////////////////////////////////////////////////////////////
/////
module game_module(input clock, reset, vsync, miso1, miso2, input [7:0]
graphics_addr, output sclk, mosi, ss, rumble_voltage, output [15:0] score,
output [31:0] to_graphics_entry);

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

   //Input variables and atan
      wire [9:0] x_input,y_input,angle_x,angle_y;

      wire [10:0] atan_x2, atan_y2;
      wire atan_nd1, atan_nd2;
      wire atan_ready;
      wire [7:0] atan_angle;

      wire [7:0] angle;
      wire bomb_button;
      wire rumble_on;

      //Input module
      input_module im0(reset,clock,miso1,miso2,vsync,mosi,sclk,ss,


x_input,y_input,angle_x,angle_y,atan_nd1,angle,atan_ready,atan_angle,bomb_but
ton);

      //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_e
ntry,


from_movement_private_entry,from_movement_done,graphics_addr,to_graphics_entr
y,
            from_collision_received, from_collision_done, to_collision_ready,
to_collision_entry, angle, bomb_button, rumble_on, score,
            state, addr);

      movement_module
mm0(reset,clock,x_input,y_input,to_movement_entry,to_movement_private_entry,
```

```
random,to_movement_ready,from_movement_entry,from_movement_private_entry,from
_movement_done,
            atan_x2, atan_y2, atan_nd2, atan_ready, atan_angle);

      arctan_module at0(clock, reset, vsync, {angle_x[9],angle_x[9:0]},
{angle_y[9],angle_y[9:0]},atan_nd1,atan_x2, atan_y2, atan_nd2, atan_ready,
atan_angle);

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

      //Controller Rumble
      rumble_module rm0(clock, vsync, reset, rumble_on, rumble_voltage);

endmodule
```

A.3 Input Module

```
`timescale 1ns / 1ns
`default_nettype none
//////////////////////////////////////////////////////////////////////////////
/////
// Author: Mark Sullivan
//
// Create Date:    14:49:50 11/04/2008
// Design Name:
// Module Name:    input_module
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
//    Takes user input and generates signals meaningful for the game
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
/////
module input_module(input reset,clock,miso1,miso2,vsync,
                                          output reg mosi, sclk, ss, output
[9:0] x_input,y_input, angle_x,angle_y, output reg atan_nd, output reg [7:0]
angle, input atan_ready, input [7:0] atan_angle, output bomb_button_out);//,
output reg [5:0] index);

      reg [5:0] clock_counter;
      reg [2:0] state;
      reg [3:0] s_clock_counter;
      reg [39:0] bit_array1,bit_array2;
      reg start;
```

```
	reg last_vsync;
	reg [5:0] index;

	//Rotates since joysticks mounted sideways
	assign y_input = {bit_array1[25:24], bit_array1[39:36], 4'b0} +
10'd576;
	assign x_input = 10'd528 + {bit_array1[9:8], bit_array1[23:20], 4'b0} ;

	assign angle_y = 10'd512 - {bit_array2[25:24], bit_array2[39:36],
4'b0};
	assign angle_x = 10'd500 - {bit_array2[9:8], bit_array2[23:20], 4'b0} ;

	wire bomb_button = bit_array2[0];
	reg last_bomb_button;
	//We want a pulse
	assign bomb_button_out = bomb_button && !last_bomb_button;

	//SPI interface
	parameter S_IDLE = 3'b000;
	parameter S_TRANSMIT = 3'b001;
	parameter S_PAUSE = 3'b010;

	always @(posedge clock) begin
	  if (reset) begin
	    sclk <= 0;
	    ss <= 1;
	    mosi <= 1'bZ;
	    clock_counter <= 0;
	    s_clock_counter <= 0;
	    state <= S_IDLE;
	    bit_array1 <= 0;
	    bit_array2 <= 0;
	    index <= 6'd39;
		last_vsync <= vsync;
		start <= 0;
		angle <= 0;
		atan_nd <= 0;
		last_bomb_button <= 0;
	  end
	  else begin
		if (vsync && ~last_vsync) begin
			start <= 1;
			last_bomb_button <= bomb_button;
		end
		else
			start <= 0;
		last_vsync <= vsync;

		if (atan_ready & vsync) begin
			angle <= atan_angle;
		end

		//We're not communicating
	    if (state == S_IDLE) begin
	      ss <= 1;
	      mosi <= 1'bZ;
	      sclk <= 0;
```

```
          clock_counter <= 6'b11_1111;
          s_clock_counter <= 0;
          index <= 6'd39;
                atan_nd <= 0;
          if (start) begin
            state <= S_TRANSMIT;
          end
        end

           //We wait 15us between bytes
        else if (state == S_PAUSE) begin
          ss <= 0;
          sclk <= 0;
          clock_counter <= clock_counter + 1;
          if (s_clock_counter == 4'b1111) begin
            state <= S_TRANSMIT;
            s_clock_counter <= 0;
            clock_counter <= 0;
          end
          if (clock_counter == 6'b11_1111) begin
            s_clock_counter <= s_clock_counter + 1;
          end
        end

           //Send and receive at 1 MHz clock
        else if (state == S_TRANSMIT) begin
          ss <= 0;
          sclk <= ~clock_counter[5];
          mosi <= 1;
          clock_counter <= clock_counter + 1;
          if (clock_counter == 6'b00_0000) begin
            bit_array1[index] <= miso1;
            bit_array2[index] <= miso2;
            index <= index - 1;
          end
          else if (index[2:0] == 3'b111 & sclk == 0) begin
            clock_counter <= 0;
            if (index == 6'd63) begin
              state <= S_IDLE;
                       atan_nd <= 1;
                   end
            else
              state <= S_PAUSE;
          end
        end
      end
    end

endmodule
```

A.4 Rumble Verilog

```
`timescale 1ns / 1ps
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
/////
// Company:
// Engineer: Mark Sullivan
//
// Create Date:    17:48:06 12/05/2008
// Design Name:
// Module Name:    rumble_module
// Project Name:
// Target Devices:
// Tool versions:
// Description: Gets a pulse and generates a high signal for some amount of
time
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
/////
module rumble_module(input clock, game_clock, reset, rumble_on, output reg
rumble_voltage);

	reg [5:0] count;
	reg last_game_clock;

	always @(posedge clock) begin
		last_game_clock <= game_clock;
		if (reset) begin
			count <= 0;
			rumble_voltage <= 0;
		end
		else if (rumble_on) begin
			count <= 6'b11_1111;
			rumble_voltage <= 1;
		end
		else if (~game_clock && last_game_clock) begin
			if (count != 0) begin
				count <= count - 1;
				rumble_voltage <= 1;
			end
			else begin
				count <= 0;
				rumble_voltage <= 0;
			end
		end
	end

endmodule
```

A.5 Game Logic Module

```
`timescale 1ns / 1ns
////////////////////////////////////////////////////////////////////////////////
/////
// Author: Mark Sullivan
//
// Create Date:    17:41:50 11/02/2008
// Design Name:    game_logic_module
// Module Name:    game_logic_module
// Project Name:   Vertex
//
// Description:
// The majority of the in-game logic will take place in this module.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
/////
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 bomb_button,
      output reg rumble_on,
      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] bombs;
      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_BOMB_READ = 4'b1011;
      parameter S_BOMB_WRITE = 4'b1100;
      parameter S_BOMB_REMOVE = 4'b1101;
      parameter S_AVATAR_COLLIDED = 4'b1110;

      //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(pub
lic_mem_in),.dout(table0_mem_out));
   mybram #(.LOGSIZE(8), .WIDTH(32))
graphics_entity_table1(.addr(table1_addr),.clk(clock),.we(table1_we),.din(pub
lic_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),.do
ut(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;
                  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;
                  rumble_on <= 0;
                  bombs <= 2'b11;
                  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;
                    //Should we bomb?
                    if (bomb_button && (bombs != 0) && !(game_over)) begin
                         state <= S_BOMB_REMOVE;
                         addr_reg <= {6'b0, bombs};
                         we_reg <= 1;
                         public_mem_in <= 0;
                         private_mem_in <= 0;
```

```
                rumble_on <= 1;
                bombs <= bombs - 1;
          end
          else begin
                state <= S_DECIDE_SPAWN;
                addr_reg <= 8'b1000_0000;
                rumble_on <= 0;
          end
        end

        //REMOVE BOMB STATE
        //Removes bomb object
        else if (state == S_BOMB_REMOVE) begin
              addr_reg <= 8'b0111_1110;
              we_reg <= 0;
              state <= S_BOMB_READ;
        end
        //BOMB READ STATE
        //Requests a read, so we only create explosions on objects
which have nonzero ID
        else if (state == S_BOMB_READ) begin
              addr_reg <= addr + 2;
              we_reg <= 0;
              state <= S_BOMB_WRITE;
              if (public_mem_out[31:28] != 4'b0) begin
                    public_mem_in <=
{4'h6,public_mem_out[27:8],8'b0};
                    private_mem_in <= 32'd10;
              end
              else begin
                    public_mem_in <= 0;
                    private_mem_in <= 0;
              end
        end

        //BOMB WRITE STATE
        //Writes if we should
        else if (state == S_BOMB_WRITE) begin
              addr_reg <= addr - 1;
              state <= S_BOMB_READ;
              if (addr_reg == 8'b1000_0000) begin
                    we_reg <= 0;
              end
              else if (addr_reg == 8'b0000_0001) begin
                    state <= S_DECIDE_SPAWN;
                    addr_reg <= 8'b1000_0000;
                    we_reg <= 0;
           end
              else
                    we_reg <= 1;
        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;
          if (random[31:28] == 5'b00_0000)
```

```
                      state <= S_SPAWNING;//SPAWNING;
                    else
                      state <= S_DECIDE_SHOOT;
                  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 == 1) 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'd5,random[27:0]};
                      private_mem_in <=
{12'b0,1'd1,random[30:28],16'd60};//32'd1;//0;
                      state <= S_DECIDE_SHOOT;
                    end
                    else begin //Nope, try next
                      addr_reg <= addr_reg + 1;
                      state <= S_SPAWNING;
                    end
                  end

                  //DECIDE SHOOT STATE
                  //We shoot every 8th frame
                  else if (state == S_DECIDE_SHOOT) begin
                    we_reg <= 0;
                    shot_cycle_counter <= shot_cycle_counter + 1;
                    if (shot_cycle_counter == 0) begin
                      if (addr_reg != 1) begin
                        addr_reg <= 8'b0000_0001;
                        shot_cycle_counter <= 0;
                        state <= S_DECIDE_SHOOT;
                      end
                      else begin
                        addr_reg <= 8'd2;
                        state <= S_SHOOTING;
                              bullet_pause <= 0;
                              shots <= 3;
                      end
                    end
                    else begin
                      state <= S_PREMOVING;
                      addr_reg <= 0;
                    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
```

```
                      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
                         private_mem_in <= 32'd120;//0;
                         shots <= shots - 1;
                         if (shots == 1) state <= S_PREMOVING;
                    end
                    else begin //Nope, try next
                      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 entity 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) //Are we done?
                        state <= S_PRECOLLIDING;
                      else if (addr_reg == 0)
                        player_data <= from_movement_entry;
                    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;
        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;
            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 //We haven't finished
            addr_reg <= addr_reg + 1;
        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;
                            rumble_on <= 1;
                            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, bomb HUD, and lives HUD
else if (state == S_RESET) begin
  case (addr_reg) //CASE INDICES LAG BY ONE!!!!!!!
```

```
                     8'b1111_1111: begin public_mem_in <=
32'b0001_0110010000_0100101100_00000000;
                              private_mem_in <= 0;
                              player_data <=
32'b0001_0000100000_0000100000_00000000; end
                     0: begin public_mem_in <=
32'b0011_0000010000_0000010000_00000000;
                              private_mem_in <= 0; end
                     1: begin public_mem_in <=
32'b0011_0000100000_0000010000_00000000;
                              private_mem_in <= 0; end
                     2: begin public_mem_in <=
32'b0011_0000110000_0000010000_00000000;
                              private_mem_in <= 0; end
                        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
                    // 127: begin public_mem_in <=
32'b1100_1000000000_1000000000_00000000;
                    //         private_mem_in <=
32'b000000_000000_0000_0000_0000_0000_0000; 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)
                     state <= S_DONE;
                  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
```

A.6 Movement Module

```
`timescale 1ns / 1ns
`default_nettype none
////////////////////////////////////////////////////////////////////////////
/////
// Author: Mark Sullivan
//
// Create Date:    23:20:05 11/10/2008
// Design Name:
// Module Name:    movement_module
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// The purpose of this module is to handle the movement of the avatar and
NPCs.
// It should receive one unit at a time, and based on some facts about the
unit and
// possibly user input or the avatar's information, decide what to do.
// The player will be solely governed by the player input.
// Bullets will continue their trajectory
// Enemies will move according to their corresponding AI
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////
/////
module movement_module(input reset, clock, input signed [9:0] x_input,
y_input,
                                          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,
                                          output reg [10:0] atan_x, atan_y,
output reg atan_nd, input atan_ready, input [7:0] atan_angle);

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

            parameter NO_ID = 4'h0;
            parameter AVATAR = 4'h1;
            parameter A_BULLET = 4'h2;
            parameter SPIKELET = 4'h3;
            parameter TURRETTE = 4'h4;
```

```
parameter SPAWN = 4'h5;
parameter EXPLOSION_0 = 4'h6;
parameter EXPLOSION_1 = 4'h7;
parameter MIMIC = 4'h8;
parameter TURRET = 4'h9;
parameter BROWN = 4'hA;
parameter PONG = 4'hB;
parameter CHASER = 4'hC;
parameter SPIKE = 4'hD;
parameter CHEVRON = 4'hE;
parameter FISH = 4'hF;

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_ATAN_WAITING = 3'd3;
parameter S_PRE_BOUND = 3'd4;
parameter S_POST_BOUND = 3'd5;

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

            //If requires atan, next angle is that supplied by atan
            reg requires_atan;

            wire [7:0] final_new_angle = requires_atan ? atan_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;
```

```
                        requires_atan <= 0;
                  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
                            delta_x <= x_input;
                            delta_y <= y_input;
                            movement_mode <= CARTESIAN;
                            new_id <= AVATAR;
                            new_angle <= angle;
                            new_state <= state;
                            player_data <= current_entry_public;
                               requires_atan <= 1;
                               atan_x <= {x_input[9],x_input[9:0]};
                               atan_y <= {y_input[9],y_input[9:0]};
                          end
                          //Move in a line at a fixed angle, destroy if hit
edge or time up (state)
                          else if (id == A_BULLET) begin
                            movement_mode <= POLAR;
                            mag_reg <= 4'sb1000;
                            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)
                                new_id <= NO_ID;
                            else
                                new_id <= A_BULLET;
                               requires_atan <= 0;
                          end
                          //Do nothing, this is the bomb hud
                          else if (id == SPIKELET) begin
                            movement_mode <= CARTESIAN;
                               delta_x <= 0;
                               delta_y <= 0;
                            new_state <= state;
                            new_angle <= angle;
                            new_id <= SPIKELET;
                               requires_atan <= 0;
                          end
                          //Do nothing, lives hud
                          else if (id == TURRETTE) begin
                            movement_mode <= CARTESIAN;
                               delta_x <= 0;
                               delta_y <= 0;
                            new_state <= state;
                            new_angle <= angle;
```

```
                          new_id <= TURRETTE;
                             requires_atan <= 0;
                        end
                        //Move opposite the player
                        else if (id == MIMIC) begin
                          delta_x <= -1*x_input;
                          delta_y <= -1*y_input;
                          movement_mode <= CARTESIAN;
                          new_id <= MIMIC;
                          new_angle <= player_angle + 8'b10000000;
                          new_state <= state;
                             requires_atan <= 0;
                        end
                        //Not really a turret.  Orbits player
                        else if (id == TURRET) begin
                          mag_reg <= 4'b0110;
                          movement_mode <= POLAR;
                          new_id <= TURRET;
                          new_angle <= angle;
                          new_state <= state;
                             requires_atan <= 1;
                             atan_y <= player_x - x;
                             atan_x <= y - player_y;
                        end
                        //Moves randomly
                        else if (id == BROWN) begin
                          delta_x <=
{state[9],state[9],state[9],state[9],state[9],state[9],state[9],state[9:5],4'
b0};
                          delta_y <=
{state[4],state[4],state[4],state[4],state[4],state[4],state[4],state[4:0],4'
b0};
                          movement_mode <= CARTESIAN;
                          new_id <= BROWN;
                          new_angle <= angle + 2;
                          if (random[4:0] == 0)
                            new_state <= {state[19:10],random[31:22]};
                          else
                            new_state <= state;
                             requires_atan <= 0;
                        end
                        //Bounces off sides of screen
                        else if (id == PONG) begin
                          delta_x <= state[0] ? 16'b0000_0000_1100_0000 :
16'b1111_1111_0100_0000;
                          delta_y <= state[1] ? 16'b0000_0000_1100_0000 :
16'b1111_1111_0100_0000;
                          movement_mode <= CARTESIAN;
                          new_id <= PONG;
                          new_angle <= angle;
                          new_state[19:2] <= state[19:2];
                             if (x == radius_int)
                                  new_state[0] <= 1;
                             else if (x == SC_WIDTH - radius_int)
                                  new_state[0] <= 0;
                             else
                                  new_state[0] <= state[0];
```

```
     if (y == radius_int)
          new_state[1] <= 1;
     else if (y == SC_HEIGHT - radius_int)
          new_state[1] <= 0;
     else
       new_state[1] <= state[1];
     requires_atan <= 0;
end
//Pursues player
else if (id == CHASER) begin
  movement_mode <= POLAR;
  mag_reg <= 4'sb0010;
  new_id <= CHASER;
  new_angle <= angle;
  new_state <= state;
     requires_atan <= 1;
     atan_x <= player_x - x;
     atan_y <= player_y - y;
end
//Moves in circles
else if (id == SPIKE) begin
  movement_mode <= POLAR;
  mag_reg <= 4'sb0100;
  new_state <= state;
  new_angle <= angle+4;
  new_id <= SPIKE;
     requires_atan <= 0;
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;
  requires_atan <= 0;
  if (state[5:0] == 0) begin
    new_id <= state[19:16];
    new_state <= 0;
  end
  else begin
    new_id <= SPAWN;
    new_state <= state - 1;
  end
end
//Aims at player, then charges
else if (id == CHEVRON) begin
     movement_mode <= POLAR;
     if (state[7] == 0) begin //Stationary
          mag_reg <= 4'sb0000;
          new_id <= CHEVRON;
          new_angle <= angle;
          if (state[6:0] == 0) begin
                new_state <= {12'b0,1'b1,7'd120};
          end
          else begin
                new_state <= state - 1;
          end
```

```
                requires_atan <= 1;
                atan_x <= player_x - x;
                atan_y <= player_y - y;
          end
          else begin //Charge
                mag_reg <= 4'sb0110;
                new_id <= id;
                new_angle <= angle;
                if (state[6:0] == 0) begin
                      new_state <= {12'b0,1'b0,7'd60};
                end
                else begin
                      new_state <= state - 1;
                end
                requires_atan <= 0;
          end
     end
     //First explision sprite
     else if (id == EXPLOSION_0) begin
          delta_x <= 0;
          delta_y <= 0;
          movement_mode <= CARTESIAN;
          new_angle <= angle;
          new_state <= state - 1;
          requires_atan <= 0;
          if (state == 0) begin
                new_id <= EXPLOSION_1;
                new_state <= 20'd10;
          end
          else
                new_id <= EXPLOSION_0;
     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;
          requires_atan <= 0;
          if (state == 0)
                new_id <= NO_ID;
          else
                new_id <= EXPLOSION_1;
     end
     //Moves sinusiodally-ish horizontally
     else if (id == FISH) begin
       movement_mode <= POLAR;
       mag_reg <= 4'sb0100;
          new_state <= state;
          new_id <= FISH;
          if (angle[5:0] == 6'b10_0000) begin
                new_state[0] <= angle[6];
          end
       if (x == radius_int) begin
                new_angle <= {!angle[7],angle[6:0]};
                movement_mode <= CARTESIAN;
```

```
                                   delta_x <= 128;
                              end
                              else if (x == SC_WIDTH - radius_int) begin
                                   new_angle <= {!angle[7],angle[6:0]};
                                   movement_mode <= CARTESIAN;
                                   delta_x <= -128;
                              end
                              else if (state[0])
                                   new_angle <= angle+1;
                              else
                                new_angle <= angle-1;
                              requires_atan <= 0;
                         end
                         else begin
                           delta_x <= 0;
                           delta_y <= 0;
                           movement_mode <= CARTESIAN;
                           new_id <= id;
                           new_angle <= angle;
                           new_state <= state;
                              requires_atan <= 0;
                         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] < radius_int |
unchecked_full_x[15:6] > 950) //Magic number, assuming you can't reach there
from right
                           begin corrected_x <= 1; fixed_x <= radius;end
                         //Too far right
                         else if (unchecked_full_x[15:6] > SC_WIDTH -
radius_int)
                           begin corrected_x <= 1; fixed_x <= SC_WIDTH_FP -
radius; end
                         //Too far up
                         if (unchecked_full_y[15:6] < radius_int |
unchecked_full_y[15:6] > 896) //Magic number, assuming you can't reach there
from bottom
                           begin corrected_y <= 1; fixed_y <= radius;end
                         //Too far down
                         else if (unchecked_full_y[15:6] > SC_HEIGHT -
radius_int)
                           begin corrected_y <= 1; fixed_y <= SC_HEIGHT_FP -
radius; end

                         if (requires_atan) begin
                              module_state <= S_ATAN_WAITING;
                              atan_nd <= 1;
                         end
                         else begin
                              movement_done <= 1;
```

```
                                module_state <= S_IDLE;
                          end

                        end

                        //Waiting for atan to finish
                        else if (module_state == S_ATAN_WAITING) begin
                              atan_nd <= 0;
                              if (atan_ready) begin
                               movement_done <= 1;
                               module_state <= S_IDLE;
                              end
                        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 = 17;
      A_BULLET: radius_int = 8;
      SPIKELET: radius_int = 16;
      TURRETTE: radius_int = 16;
      MIMIC: radius_int = 17;
      TURRET: radius_int = 16;
      BROWN: radius_int = 16;
      PONG: radius_int = 16;
      CHASER: radius_int = 16;
      SPIKE: radius_int = 16;
      SPAWN: radius_int = 16;
            CHEVRON: radius_int = 16;
            EXPLOSION_0: radius_int = 16;
            EXPLOSION_1: radius_int = 16;
      default: radius_int = 16;
    endcase
  end


endmodule
```

A.7 Collision Module

```
`timescale 1ns / 1ns
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
/////
// Author: Mark Sullivan
//
// Create Date:    21:20:05 11/23/2008
```

```
// Design Name:
// Module Name:    collision_module
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// The purpose of this module is to check for colliding objects.
// Collision checks will be entirely distance based, so it's as though all
objects are circles.
// If there is no collision, a new entity should be acquired from the game
module
// If collision_ready is asserted, a new object is ready to be checked
against the enemies
// If there is a collision, collision_done should be asserted
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
/////

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 A_BULLET = 4'h2;
      parameter SPIKELET = 4'h3;
      parameter TURRETTE = 4'h4;
      parameter SPAWN = 4'h5;
      parameter EXPLOSION_0 = 4'h6;
      parameter EXPLOSION_1 = 4'h7;
      parameter MIMIC = 4'h8;
      parameter TURRET = 4'h9;
      parameter BROWN = 4'hA;
      parameter PONG = 4'hB;
      parameter CHASER = 4'hC;
      parameter SPIKE = 4'hD;
  parameter CHEVRON = 4'hE;

  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;
      A_BULLET: primary_radius_sq = 64;
      default: primary_radius_sq = 64;
    endcase
  end

  always @(secondary_id) begin
    case(secondary_id)
      SPIKELET: secondary_radius_sq = 256;
      TURRETTE: secondary_radius_sq = 256;
      MIMIC: secondary_radius_sq = 256;
      TURRET: secondary_radius_sq = 256;
      BROWN: secondary_radius_sq = 256;
      PONG: secondary_radius_sq = 256;
      CHASER: secondary_radius_sq = 256;
      SPIKE: secondary_radius_sq = 256;
      SPAWN: secondary_radius_sq = 256;
              CHEVRON: secondary_radius_sq = 256;
```

```
      default: secondary_radius_sq = 256;
    endcase
  end
endmodule
```

A.8 Random Number Verilog

```
////////////////////////////////////////////////////////////////////////////////
///
// Author: Mark Sullivan
//
// Create Date:   15:06:03 11/02/2008
// Design Name:   random_number_generator
// Module Name:   C:/marks3/Vertex/rng_tb.v
// Project Name:  Vertex
//
// Description:
// Every clock cycle, a new psuedo random number is generated.
//
// Dependencies:
// Random seeding
//
// Revision:
// Revision 0.01 - File Created
//
// Additional Comments:
// Module takes the form of a linear congruential generator, which should be
// sufficient for our purposes.  Basically, this is a linear function,
// X(n+1) = A*X(n) + C, but then the high order bits are truncated, so
// we take mod n, which is assumed here to be a power of 2 for simplicity.
// Default values are those used by Borland C/C++ rand()
//
////////////////////////////////////////////////////////////////////////////////
///

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
```

A.9 Arctangent Module

```
`timescale 1ns / 1ns
////////////////////////////////////////////////////////////////////////////////
/////
// Company:
// Engineer:
//
// Create Date:    16:24:40 12/01/2008
// Design Name:
// Module Name:    arctan_module
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
/////
module arctan_module(input clock, reset, vsync, input[10:0] x1, y1, input
nd1,
            input [10:0] x2, y2, input nd2, output atan_ready, output [7:0]
atan_angle);

      wire [10:0] atan_x = vsync ? x1 : x2;
      wire [10:0] atan_y = vsync ? y1 : y2;
      wire atan_nd = vsync ? nd1 : nd2;
      wire [1:0] atan_garbage;
      arctan atan(atan_x,atan_y,atan_nd,clock,reset,
{atan_garbage,atan_angle},atan_ready);
      //assign atan_angle = 8'b1100_0000;
      //assign atan_ready = 1;

endmodule
```

A.10 Base graphics module

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
/////
// Company:
// Engineer:
//
// Create Date:    16:01:56 12/03/2008
// Design Name:
// Module Name:    vertex_graphics
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
/////
module vertex_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 vertex
   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));

      vertex_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: vertex
   //  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: vertex
       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
```

A.11 SVGA module

```
////////////////////////////////////////////////////////////////////////////
///
//
// svga: Generate SVGA display signals (800 x 600 @ 60Hz)
//
////////////////////////////////////////////////////////////////////////////
///

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

      //800x600, 60Hz   39.8571     800   40    128   87    600   1     4
      23

   // horizontal: 1058 pixels total
   // display 800 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
```

```
   assign hblankon = (hcount == 799);
   assign hsyncon = (hcount == 839);
   assign hsyncoff = (hcount == 967);
   assign hreset = (hcount == 1057);

   // vertical: 628 lines total
   // display 600 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount == 599);
   assign vsyncon = hreset & (vcount == 600);
   assign vsyncoff = hreset & (vcount == 604);
   assign vreset = hreset & (vcount == 627);

   // 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
```

A.12 Double buffer module

```
module vertex_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,         // XVGA horizontal sync signal (active low)
   input vsync,         // XVGA vertical sync signal (active low)
   input blank,         // XVGA 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, // vertex's horizontal sync
   output vvsync, // vertex's vertical sync
   output vblank, // vertex's blanking
   output reg [31:0] pixel    // vertex's 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
            // delay the video sync and blank signals to account for latency
in reading and writing ZBT memory
            /*hsync_delay0 <= hsync && !reset;
            hsync_delay1 <= hsync_delay0 && !reset;
            vhsync <= hsync_delay1 && !reset;
            vsync_delay0 <= vsync && !reset;
            vsync_delay1 <= vsync_delay0 && !reset;
            vvsync <= vsync_delay1 && !reset;
            blank_delay0 <= blank && !reset;
            blank_delay1 <= blank && !reset;
            vblank <= blank_delay1 && !reset;*/

            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
```

A.13 Bresenham line-drawing module

```
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
```

A.14 Shape module

```
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
```

A.15 Shape lookup table

```
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 A_BULLET = 4'h2;
	parameter BOMB = 4'h3;
	parameter LIFE = 4'h4;
	parameter SPAWN = 4'h5;
	parameter EXPLOSION_0 = 4'h6;
	parameter EXPLOSION_1 = 4'h7;
	parameter MIMIC = 4'h8;
	parameter TURRET = 4'h9;
	parameter BROWN = 4'hA;
	parameter PONG = 4'hB;
	parameter CHASER = 4'hC;
	parameter SPIKE = 4'hD;
	parameter CHEVRON = 4'hE;
	parameter FISH = 4'hF;

	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 = 15;
						y0 = 0;
						x1 = -1;
						y1 = 5;
						ignore = 0;
					end
					4'h1: begin
						x0 = -1;
						y0 = 5;
						x1 = 2;
						y1 = 10;
						ignore = 0;
					end
```

```
4'h2: begin
      x0 = 2;
      y0 = 10;
      x1 = 10;
      y1 = 8;
      ignore = 0;
end
4'h3: begin
      x0 = 10;
      y0 = 8;
      x1 = 4;
      y1 = 14;
      ignore = 0;
end
4'h4: begin
      x0 = 4;
      y0 = 14;
      x1 = -8;
      y1 = 14;
      ignore = 0;
end
4'h5: begin
      x0 = -8;
      y0 = 14;
      x1 = -14;
      y1 = 8;
      ignore = 0;
end
4'h6: begin
      x0 = -14;
      y0 = 8;
      x1 = -6;
      y1 = 10;
      ignore = 0;
end
4'h7: begin
      x0 = -6;
      y0 = 10;
      x1 = -12;
      y1 = 0;
      ignore = 0;
end
4'h8: begin
      x0 = -12;
      y0 = 0;
      x1 = -6;
      y1 = -10;
      ignore = 0;
end
4'h9: begin
      x0 = -6;
      y0 = -10;
      x1 = -14;
      y1 = -8;
      ignore = 0;
end
4'hA: begin
```

```
                x0 = -14;
                y0 = -8;
                x1 = -8;
                y1 = -14;
                ignore = 0;
            end
            4'hB: begin
                x0 = -8;
                y0 = -14;
                x1 = 4;
                y1 = -14;
                ignore = 0;
            end
            4'hC: begin
                x0 = 4;
                y0 = -14;
                x1 = 10;
                y1 = -8;
                ignore = 0;
            end
            4'hD: begin
                x0 = 10;
                y0 = -8;
                x1 = 2;
                y1 = -10;
                ignore = 0;
            end
            4'hE: begin
                x0 = 2;
                y0 = -10;
                x1 = -1;
                y1 = -5;
                ignore = 0;
            end
            4'hF: begin
                x0 = -1;
                y0 = -5;
                x1 = 15;
                y1 = 0;
                ignore = 0;
            end
        endcase
        rgba = 32'hFFFFFFFF;
    end
    A_BULLET: begin
        case (segment)
            4'h0: begin
                x0 = -3;
                y0 = -3;
                x1 = 4;
                y1 = -1;
                ignore = 0;
            end
            4'h1: begin
                x0 = 4;
                y0 = -1;
                x1 = 4;
```

```
                    y1 = 1;
                    ignore = 0;
              end
              4'h2: begin
                    x0 = 4;
                    y0 = 1;
                    x1 = -3;
                    y1 = 3;
                    ignore = 0;
              end
              4'h3: begin
                    x0 = -3;
                    y0 = 3;
                    x1 = -3;
                    y1 = -3;
                    ignore = 0;
              end
              default: begin
                    x0 = 0;
                    y0 = 0;
                    x1 = 0;
                    y1 = 0;
                    ignore = 1;
              end
        endcase
        rgba = 32'hFFFFFFFF;
  end
  BOMB: begin
        case (segment)
              4'h0: begin
                    x0 = -2;
                    y0 = -3;
                    x1 = 2;
                    y1 = -3;
                    ignore = 0;
              end
              4'h1: begin
                    x0 = 2;
                    y0 = -3;
                    x1 = 3;
                    y1 = 0;
                    ignore = 0;
              end
              4'h2: begin
                    x0 = 3;
                    y0 = 0;
                    x1 = 2;
                    y1 = 3;
                    ignore = 0;
              end
              4'h3: begin
                    x0 = 2;
                    y0 = 3;
                    x1 = -2;
                    y1 = 3;
                    ignore = 0;
              end
```

```
4'h4: begin
      x0 = -2;
      y0 = 3;
      x1 = -3;
      y1 = 0;
      ignore = 0;
end
4'h5: begin
      x0 = -3;
      y0 = 0;
      x1 = -2;
      y1 = -3;
      ignore = 0;
end
4'h6: begin
      x0 = -2;
      y0 = -6;
      x1 = 2;
      y1 = -6;
      ignore = 0;
end
4'h7: begin
      x0 = 4;
      y0 = -4;
      x1 = 6;
      y1 = -2;
      ignore = 0;
end
4'h8: begin
      x0 = 6;
      y0 = 2;
      x1 = 4;
      y1 = 4;
      ignore = 0;
end
4'h9: begin
      x0 = 2;
      y0 = 6;
      x1 = -2;
      y1 = 6;
      ignore = 0;
end
4'hA: begin
      x0 = -4;
      y0 = 4;
      x1 = -6;
      y1 = 2;
      ignore = 0;
end
4'hB: begin
      x0 = -6;
      y0 = -2;
      x1 = -4;
      y1 = -4;
      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
    LIFE: begin
        ignore = 0;
        case (segment)
            4'h0: begin
                x0 = 7;
                y0 = 0;
                x1 = -1;
                y1 = 3;
            end
            4'h1: begin
                x0 = -1;
                y0 = 3;
                x1 = 1;
                y1 = 5;
            end
            4'h2: begin
                x0 = 1;
                y0 = 5;
                x1 = 5;
                y1 = 4;
            end
            4'h3: begin
                x0 = 5;
                y0 = 4;
                x1 = 2;
                y1 = 7;
            end
            4'h4: begin
                x0 = 2;
                y0 = 7;
                x1 = -4;
                y1 = 7;
            end
            4'h5: begin
                x0 = -4;
                y0 = 7;
                x1 = -7;
                y1 = 4;
            end
            4'h6: begin
                x0 = -7;
                y0 = 4;
                x1 = -3;
                y1 = 5;
            end
            4'h7: begin
                x0 = -3;
```

```
                y0 = 5;
                x1 = -6;
                y1 = 0;
            end
            4'h8: begin
                x0 = -6;
                y0 = 0;
                x1 = -3;
                y1 = -5;
            end
            4'h9: begin
                x0 = -3;
                y0 = -5;
                x1 = -7;
                y1 = -4;
            end
            4'hA: begin
                x0 = -7;
                y0 = -4;
                x1 = -4;
                y1 = -7;
            end
            4'hB: begin
                x0 = -4;
                y0 = -7;
                x1 = 2;
                y1 = -7;
            end
            4'hC: begin
                x0 = 2;
                y0 = -7;
                x1 = 5;
                y1 = -4;
            end
            4'hD: begin
                x0 = 5;
                y0 = -4;
                x1 = 1;
                y1 = -5;
            end
            4'hE: begin
                x0 = 1;
                y0 = -5;
                x1 = -1;
                y1 = -3;
            end
            4'hF: begin
                x0 = -1;
                y0 = -3;
                x1 = 7;
                y1 = 0;
            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
      MIMIC: begin
            case (segment)
                  4'h0: begin
                        x0 = -2;
                        y0 = 0;
                        x1 = 4;
                        y1 = -10;
                        ignore = 0;
                  end
                  4'h1: begin
                        x0 = 4;
                        y0 = -10;
                        x1 = 12;
                        y1 = -8;
                        ignore = 0;
                  end
                  4'h2: begin
                        x0 = 12;
                        y0 = -8;
                        x1 = 6;
                        y1 = -14;
                        ignore = 0;
                  end
                  4'h3: begin
                        x0 = 6;
                        y0 = -14;
                        x1 = -6;
                        y1 = -14;
                        ignore = 0;
                  end
                  4'h4: begin
                        x0 = -6;
                        y0 = -14;
                        x1 = -12;
                        y1 = -8;
                        ignore = 0;
                  end
                  4'h5: begin
                        x0 = -12;
                        y0 = -8;
                        x1 = -4;
```

```
      y1 = -10;
      ignore = 0;
end
4'h6: begin
      x0 = -4;
      y0 = -10;
      x1 = -10;
      y1 = 0;
      ignore = 0;
end
4'h7: begin
      x0 = -10;
      y0 = 0;
      x1 = -4;
      y1 = 10;
      ignore = 0;
end
4'h8: begin
      x0 = -4;
      y0 = 10;
      x1 = -12;
      y1 = 8;
      ignore = 0;
end
4'h9: begin
      x0 = -12;
      y0 = 8;
      x1 = -6;
      y1 = 14;
      ignore = 0;
end
4'hA: begin
      x0 = -6;
      y0 = 14;
      x1 = 6;
      y1 = 14;
      ignore = 0;
end
4'hB: begin
      x0 = 6;
      y0 = 14;
      x1 = 12;
      y1 = 8;
      ignore = 0;
end
4'hC: begin
      x0 = 12;
      y0 = 8;
      x1 = 4;
      y1 = 10;
      ignore = 0;
end
4'hD: begin
      x0 = 4;
      y0 = 10;
      x1 = -2;
      y1 = 0;
```

```
                        ignore = 0;
                end
                default: begin
                        x0 = 0;
                        y0 = 0;
                        x1 = 0;
                        y1 = 0;
                        ignore = 1;
                end
        endcase
        rgba = 32'h00FFFFFF;
end
TURRET: begin
        case (segment)
                4'h0: begin
                        x0 = 10;
                        y0 = 0;
                        x1 = 14;
                        y1 = -4;
                        ignore = 0;
                end
                4'h1: begin
                        x0 = 14;
                        y0 = -4;
                        x1 = 7;
                        y1 = -4;
                        ignore = 0;
                end
                4'h2: begin
                        x0 = 7;
                        y0 = -4;
                        x1 = 0;
                        y1 = -11;
                        ignore = 0;
                end
                4'h3: begin
                        x0 = 0;
                        y0 = -11;
                        x1 = -11;
                        y1 = 0;
                        ignore = 0;
                end
                4'h4: begin
                        x0 = -11;
                        y0 = 0;
                        x1 = 0;
                        y1 = 11;
                        ignore = 0;
                end
                4'h5: begin
                        x0 = 0;
                        y0 = 11;
                        x1 = 7;
                        y1 = 4;
                        ignore = 0;
                end
                4'h6: begin
```

```
                x0 = 7;
                y0 = 4;
                x1 = 14;
                y1 = 4;
                ignore = 0;
            end
            4'h7: begin
                x0 = 14;
                y0 = 4;
                x1 = 10;
                y1 = 0;
                ignore = 0;
            end
            default: begin
                x0 = 0;
                y0 = 0;
                x1 = 0;
                y1 = 0;
                ignore = 1;
            end
        endcase
        rgba = 32'h00FF00FF;
    end
    BROWN: begin
        case (segment)
            4'h0: begin
                x0 = 3;
                y0 = 0;
                x1 = 9;
                y1 = -6;
                ignore = 0;
            end
            4'h1: begin
                x0 = 9;
                y0 = -6;
                x1 = 0;
                y1 = -10;
                ignore = 0;
            end
            4'h2: begin
                x0 = 0;
                y0 = -10;
                x1 = -9;
                y1 = -6;
                ignore = 0;
            end
            4'h3: begin
                x0 = -9;
                y0 = -6;
                x1 = -3;
                y1 = 0;
                ignore = 0;
            end
            4'h4: begin
                x0 = -3;
                y0 = 0;
                x1 = -9;
```

```
                y1 = 6;
                ignore = 0;
            end
            4'h5: begin
                x0 = -9;
                y0 = 6;
                x1 = 0;
                y1 = 10;
                ignore = 0;
            end
            4'h6: begin
                x0 = 0;
                y0 = 10;
                x1 = 9;
                y1 = 6;
                ignore = 0;
            end
            4'h7: begin
                x0 = 9;
                y0 = 6;
                x1 = 3;
                y1 = 0;
                ignore = 0;
            end
            default: begin
                x0 = 0;
                y0 = 0;
                x1 = 0;
                y1 = 0;
                ignore = 1;
            end
        endcase
        rgba = 32'h996633FF;
    end
    PONG: 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'hCC99FFFF;
end
CHASER: begin
        case (segment)
                4'h0: begin
                        x0 = 11;
                        y0 = 0;
                        x1 = -3;
                        y1 = 7;
                        ignore = 0;
                end
                4'h1: begin
                        x0 = -3;
                        y0 = 7;
                        x1 = -10;
                        y1 = 0;
                        ignore = 0;
                end
                4'h2: begin
                        x0 = -10;
                        y0 = 0;
                        x1 = -3;
```

```
                        y1 = -7;
                        ignore = 0;
                end
                4'h3: begin
                        x0 = -3;
                        y0 = -7;
                        x1 = 11;
                        y1 = 0;
                        ignore = 0;
                end
                default: begin
                        x0 = 0;
                        y0 = 0;
                        x1 = 0;
                        y1 = 0;
                        ignore = 1;
                end
        endcase
        rgba = 32'hFFAACCFF;
end
SPIKE: begin
        case (segment)
                4'h0: begin
                        x0 = 3;
                        y0 = 0;
                        x1 = 12;
                        y1 = 0;
                        ignore = 0;
                end
                4'h1: begin
                        x0 = 2;
                        y0 = 2;
                        x1 = 8;
                        y1 = 8;
                        ignore = 0;
                end
                4'h2: begin
                        x0 = 0;
                        y0 = 3;
                        x1 = 0;
                        y1 = 12;
                        ignore = 0;
                end
                4'h3: begin
                        x0 = -2;
                        y0 = 2;
                        x1 = -8;
                        y1 = 8;
                        ignore = 0;
                end
                4'h4: begin
                        x0 = -3;
                        y0 = 0;
                        x1 = -12;
                        y1 = 0;
                        ignore = 0;
                end
```

```
                4'h5: begin
                      x0 = -2;
                      y0 = -2;
                      x1 = -8;
                      y1 = -8;
                      ignore = 0;
                end
                4'h6: begin
                      x0 = 0;
                      y0 = -3;
                      x1 = 0;
                      y1 = -12;
                      ignore = 0;
                end
                4'h7: begin
                      x0 = 2;
                      y0 = -2;
                      x1 = 8;
                      y1 = -8;
                      ignore = 0;
                end
                default: begin
                      x0 = 0;
                      y0 = 0;
                      x1 = 0;
                      y1 = 0;
                      ignore = 1;
                end
          endcase
          rgba = 32'hDDDD22FF;
    end
    CHEVRON: begin
          case (segment)
                4'h0: begin
                      x0 = 11;
                      y0 = 0;
                      x1 = 4;
                      y1 = 7;
                      ignore = 0;
                end
                4'h1: begin
                      x0 = 4;
                      y0 = 7;
                      x1 = -10;
                      y1 = 10;
                      ignore = 0;
                end
                4'h2: begin
                      x0 = -10;
                      y0 = 10;
                      x1 = -4;
                      y1 = 4;
                      ignore = 0;
                end
                4'h3: begin
                      x0 = -4;
                      y0 = 4;
```

```
                        x1 = -8;
                        y1 = 0;
                        ignore = 0;
                end
                4'h4: begin
                        x0 = -8;
                        y0 = 0;
                        x1 = -4;
                        y1 = -4;
                        ignore = 0;
                end
                4'h5: begin
                        x0 = -4;
                        y0 = -4;
                        x1 = -10;
                        y1 = -10;
                        ignore = 0;
                end
                4'h6: begin
                        x0 = -10;
                        y0 = -10;
                        x1 = 4;
                        y1 = -7;
                        ignore = 0;
                end
                4'h7: begin
                        x0 = 4;
                        y0 = -7;
                        x1 = 11;
                        y1 = 0;
                        ignore = 0;
                end
                default: begin
                        x0 = 0;
                        y0 = 0;
                        x1 = 0;
                        y1 = 0;
                        ignore = 1;
                end
        endcase
        rgba = 32'h9922FFFF;
end
FISH: begin
        case (segment)
                4'h0: begin
                        x0 = 12;
                        y0 = 0;
                        x1 = 6;
                        y1 = 6;
                        ignore = 0;
                end
                4'h1: begin
                        x0 = 6;
                        y0 = 6;
                        x1 = -3;
                        y1 = 7;
                        ignore = 0;
```

```
end
4'h2: begin
      x0 = -3;
      y0 = 7;
      x1 = -5;
      y1 = 5;
      ignore = 0;
end
4'h3: begin
      x0 = -5;
      y0 = 5;
      x1 = -9;
      y1 = 9;
      ignore = 0;
end
4'h4: begin
      x0 = -9;
      y0 = 9;
      x1 = -7;
      y1 = 0;
      ignore = 0;
end
4'h5: begin
      x0 = -7;
      y0 = 0;
      x1 = -9;
      y1 = -9;
      ignore = 0;
end
4'h6: begin
      x0 = -9;
      y0 = -9;
      x1 = -5;
      y1 = -5;
      ignore = 0;
end
4'h7: begin
      x0 = -5;
      y0 = -5;
      x1 = -3;
      y1 = -7;
      ignore = 0;
end
4'h8: begin
      x0 = -3;
      y0 = -7;
      x1 = 6;
      y1 = -6;
      ignore = 0;
end
4'h9: begin
      x0 = 6;
      y0 = -6;
      x1 = 12;
      y1 = 0;
      ignore = 0;
end
```

```
                        4'hA: begin
                              x0 = 6;
                              y0 = -2;
                              x1 = 6;
                              y1 = -2;
                              ignore = 0;
                        end
                        default: begin
                              x0 = 0;
                              y0 = 0;
                              x1 = 0;
                              y1 = 0;
                              ignore = 1;
                        end
                  endcase
                  rgba = 32'h22AAEEFF;
            end
            default: begin
                  x0 = 0;
                  y0 = 0;
                  x1 = 0;
                  y1 = 0;
                  ignore = 1;
                  rgba = 32'h00000000;
            end
      endcase
   end
endmodule
```

A.16 Ramclock Module

```
`timescale 1ns / 1ps
`default_nettype none
///////////////////////////////////////////////////////////////////////////////
//
//
// 6.111 FPGA Labkit -- ZBT RAM clock generation
//
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
//
// This module generates deskewed clocks for driving the ZBT SRAMs and FPGA
// registers. A special feedback trace on the labkit PCB (which is length
// matched to the RAM traces) is used to adjust the RAM clock phase so that
// rising clock edges reach the RAMs at exactly the same time as rising clock
// edges reach the registers in the FPGA.
//
// The RAM clock signals are driven by DDR output buffers, which further
// ensures that the clock-to-pad delay is the same for the RAM clocks as it
is
// for any other registered RAM signal.
//
// When the FPGA is configured, the DCMs are enabled before the chip-level I/
O
// drivers are released from tristate. It is therefore necessary to
// artificially hold the DCMs in reset for a few cycles after configuration.
// This is done using a 16-bit shift register. When the DCMs have locked, the
// <lock> output of this mnodule will go high. Until the DCMs are locked, the
// ouput clock timings are not guaranteed, so any logic driven by the
// <fpga_clock> should probably be held inreset until <locked> is high.
//
///////////////////////////////////////////////////////////////////////////////
//

module ramclock(ref_clock, fpga_clock, ram0_clock, ram1_clock,
              clock_feedback_in, clock_feedback_out, locked);

   input ref_clock;                 // Reference clock input
   output fpga_clock;               // Output clock to drive FPGA logic
   output ram0_clock, ram1_clock;   // Output clocks for each RAM chip
   input  clock_feedback_in;        // Output to feedback trace
   output clock_feedback_out;       // Input from feedback trace
   output locked;                   // Indicates that clock outputs are
stable

   wire  ref_clk, fpga_clk, ram_clk, ram_clock, fb_clk, lock1, lock2,
dcm_reset;

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

```
      //IBUFG ref_buf (.O(ref_clk), .I(ref_clock));
   assign ref_clk = ref_clock;

   BUFG int_buf (.O(fpga_clock), .I(fpga_clk));

   DCM int_dcm (.CLKFB(fpga_clock),
            .CLKIN(ref_clk),
            .RST(dcm_reset),
            .CLK0(fpga_clk),
            .LOCKED(lock1));
   // synthesis attribute DLL_FREQUENCY_MODE of int_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of int_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of int_dcm is "FALSE"
   // synthesis attribute DFS_FREQUENCY_MODE of int_dcm is "LOW"
   // synthesis attribute CLK_FEEDBACK of int_dcm  is "1X"
   // synthesis attribute CLKOUT_PHASE_SHIFT of int_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of int_dcm is 0

   BUFG ext_buf (.O(ram_clock), .I(ram_clk));

   IBUFG fb_buf (.O(fb_clk), .I(clock_feedback_in));

   DCM ext_dcm (.CLKFB(fb_clk),
                .CLKIN(ref_clk),
                .RST(dcm_reset),
                .CLK0(ram_clk),
                .LOCKED(lock2));
   // synthesis attribute DLL_FREQUENCY_MODE of ext_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of ext_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of ext_dcm is "FALSE"
   // synthesis attribute DFS_FREQUENCY_MODE of ext_dcm is "LOW"
   // synthesis attribute CLK_FEEDBACK of ext_dcm  is "1X"
   // synthesis attribute CLKOUT_PHASE_SHIFT of ext_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of ext_dcm is 0

   SRL16 dcm_rst_sr (.D(1'b0), .CLK(ref_clk), .Q(dcm_reset),
                 .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   // synthesis attribute init of dcm_rst_sr is "000F";


   OFDDRRSE ddr_reg0 (.Q(ram0_clock), .C0(ram_clock), .C1(~ram_clock),
                  .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));
   OFDDRRSE ddr_reg1 (.Q(ram1_clock), .C0(ram_clock), .C1(~ram_clock),
                  .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));
   OFDDRRSE ddr_reg2 (.Q(clock_feedback_out), .C0(ram_clock),
.C1(~ram_clock),
                  .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));

   assign locked = lock1 && lock2;

endmodule
```