Bellagio Fountain Simulation

Joseph Lane, Allen Yin, George Rossick

December 10, 2009

Abstract

The Bellagio Fountain Simulation project aims to produce an accurate simulation of the Las Vegas Bellagio fountains. Real-time audio processing is used to dynamically create a visually appealing particle fountain show. Accurate physics modeling control each particle to provide collisions between other particles and the bounded simulation world. Finally, graphics render the three dimensional world to realistically display the the visualization on screen for everyone to enjoy.

Contents

List of Figures

List of Tables

1 Overview

The goal of the Bellagio Fountain Simulation project is to create an accurate and visually appealing simulation of the Las Vegas Bellagio fountains. The Bellagio fountains are well known for being an extremely beautiful combination of water, music, and light. The many fountains spray water hundreds of meters in the air, choreographed to music and lighting effects.

To make the simulation of such an event possible, certain abstractions and simplifications were made. First, instead of water, the Bellagio Fountain Simulation renders particles on the screen. The particles can interact with each other and the environment by colliding off of each other and the wall of the Bellagio world. Second, limited by the performance of the FPGA, eight fountains instead of hundreds are used to launch individual particles into the air. Finally, unlike the Bellagio fountains which are pre-choreographed to music, the simulated Bellagio fountains will respond in real time to audio input from the microphone.

The simulation is divided into four main parts; audio, physics, graphics, and a central memory unit. The audio system is responsible for processing an incoming stream of audio and generating particles with features that correspond to different characteristics of the audio signal. For instance, the initial velocity of the particle is proportional the energy of the audio signal. Once the particles have been generated by the audio modules, they are written into the central memory unit. The central memory unit holds all of the on-screen particles, where each particle is a series of bits representing position, velocity, and color. Once the generated balls have been written to the central memory unit, they are updated by the physics modules. The main function of the physics modules is to update the positions and velocities of all on-screen particles. Each particle is able to collide with one another in a perfectly elastic particle collision, as well as bounce off of the walls of the surrounding world. Finally, the graphics modules have the responsibility of displaying each of the balls on the screen at a refresh rate of 30 Hz. Double buffering is implemented to allow for smooth transitions between frames while three dimensional cues aid in the visualization of the three dimensional world.

2 Module Specification

2.1 Audio Design (J. Lane)

The Audio Processing unit was implemented using a series of modules that perform unique functions to quantify and manage incoming audio signals. Figure 1 shows a chart of the separate componenets of the system as well as how they are connected to one another [1]. Each of the modules seen in Figure 1 is described in detail in the following sections.

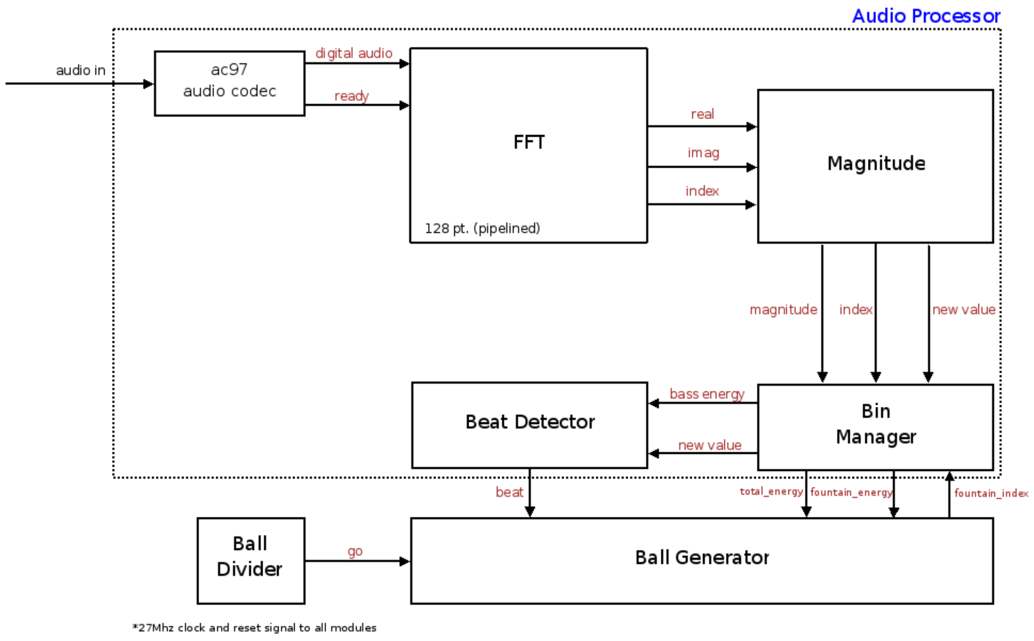

Figure 1: Block Diagram of Audio System

2.1.1 AC97 Codec

Analog audio signals first pass through the AC97 audio codec that is included as part of the lab kit. Inside of this module is an analog signal amplifier and an analog to digital converter. The audio signal is first amplified before entering the analog to digital converter. The analog to digital converter samples the audio signals, producing 18 bits samples at a rate of 48 kHz. A ready signal is given once each sample has been completed. In this design, only the most significant 8 bits of the sampled audio data are passed on to the next module. While it is possible to utilize all 18 bits of the sampled audio data, including this additional resolution is not necessary for computing relative energy levels of the audio signal.

2.1.2 Fast Fourier Transform (FFT)

The FFT module serves to translate the 8-bit digital audio signal from the time domain into the frequency domain. To do so, the FFT efficiently computes an N-point Discrete Fourier Transform. N was initially chosen to ensure high performance of the beat detection module. Audio associated with musical beats typically occurs in the range of 20-400 Hz. As such, the beat detection module should only consider audio produced in this frequency range. An N value of 128 provides a frequency resolution of 375 Hz. This is enough resolution for the beat detection module to operate effectively. While a higher point size would provide additional resolution, selecting a larger point size would increase the space required for the FFT as well as the complexity of the magnitude and bin manager modules. Addtionally, testing revealed minimal increases in beat detection performance as a result of greater low frequency resolution.

The FFT module divides the 48 kHz band limited frequency spectrum into 128 uniform bins. The FFT module was created using Xilinx software. The ready signal from the AC97 Codec was used as the clock enable signal to the module so that computations were performed only when there was a new sample. Additionally, the module was pipelined to be able to produce an output for every 8-bit audio input. The FFT produces a 16-bit unscaled real value, a 16-bit unscaled imaginary value, and a 7-bit index.

The magnitude of the real and imaginary values indicates the level of the audio within frequency band that is determined by the index. The frequency band is related to the index by equations 1 and 2.

$$f_{low} = \frac{samplingrate}{N} * index \tag{1}$$

$$f_{high} = \frac{samplingrate}{N} * (index + 1) \tag{2}$$

2.1.3 Energy

The Energy module computes the energy, or power, of real and imaginary values produced by the FFT module. While it is common practice to calculate the magnitude, as opposed to the energy, of the real and imaginary FFT values, using energy values yielded equal if not better results within the beat detection and ball generator modules. Additionally, computing the magnitude of the real and imaginary values requires a square root operation that adds extra latency to our Energy module and additional hardware in our system. The equation to calculate the energy of the system is shown by equation 3.

$$energy = real^2 + imag^2 \tag{3}$$

Before entering the Energy module, the 16-bit unscaled real and imaginary values are scaled by taking the most significant 8 bits. While this operation reduces the resolution of our data, the decrease in resolution did not appear to have any negative effects during simulation testing. Furthermore, the result of squaring an 8-bit number is a 16-bit number. Within the Virtex II FPGA, each configurable logic block contains 4, 4-bit adders making the summation operation very efficient to implement in hardware.

Computing the energy in this manner requires two clock cycles, one to square the real and imaginary values and another to add them together. As such, this module is piplined to allow it to input new FFT values as they are produced. Given two 8-bit values, the resulting energy is a 17-bit value. Along with real and imaginary FFT values, the Energy module receives the FFT index which it delays by one clock cycle such the new energy value and the correct index associated with that energy are outputted at the same time.

2.1.4 Bin Manager

The function of the Bin Manager module is to combine the energy values from the Energy module and sort them into their corresponding fountains. The module keeps track of the average energy value for each fountain as well as the total energy over all fountains for use by the ball generator and beat detection modules. Since the Bellagio world only contains 8 fountains and there are 128 FFT bins, each fountain is assigned a set of FFT bins. The energy of the fountain is computed as the average energy over all of its bins. The table below shows the distribution of bins to the eight Bellagio fountains.

Table 1: Required Energy Assignments and Bit Shifts Per Fountain

Fountain	Indices	Bitshift
0	0	4
1	1-2	5
2	3-6	6
3	7-10	6
4	11-18	7
5	19-26	7
6	27-34	7
7	35-42	7

Typically, most modern music has a frequency range from approximately 20 Hz up to about 8,000 Hz. Classical music and instrumental music can often extend to about 12,000 Hz. Consequently, only frequency bins 0 to 42 are assigned to a fountain. This equates to a sufficient frequency range of 0 to 15,750 Hz.

In order to make the behavior of the fountains robust to noise within the audio signal, the average energy value for each fountain is computed over 16 samples per bin. Every 16 samples per bin, the average energy for each fountain is recalcuted. This equates to a new average value for each fountain every 0.04 seconds. While a running average would have resulted in a more accurate average at any given instant in time, this simpler method for computing average also provides an accurate estimation due to the short time between updates.

Since bin manager averages over 16 energy values, each energy value is first bit shifted to the right by 4 bits. The sum over 16 such values would then equate to their average energy. For those fountains that contain multiple bins, a further bit shift is needed to compute the average energy. The number of bins per fountain is purposely a power of two. This makes computing the average for each fountain a matter of shifting its sum to the right by the correct number of bits. The number of bit shifts required to compute the average energy is also shown in the table above.

The average energy of the lowest frequency bin, called `bass_energy`, is wired to the beat detection module along with a `bass_ready` signal that is high for one clock cycle each time a new average is calculated. Additionally, Ball Generator can request the average energy of each fountain through the use of the `fountain_index` signal. The output energy from the Bin Manager module corresponds to the average energy for the fountain determined by `fountain_index`.

Along with the individual average energies for each fountain, the Bin Manager module also maintains the total energy across all eight fountains. This signal is also given to the Ball Generator module.

2.1.5 Beat Detection

The Beat Detection module takes in a bass energy value and a ready signal from the Bin Manager module and outputs a signal, `beat`, which is given to the Ball Generator module. The function of the the Beat Detector module is to find peaks within the bass energy values. Each significant peak is classified as a beat.

The beat detection algorithm contained in the Beat Detection module is very robust and accurate for a wide range of beats occuring in the low frequency range. The algorithm compares the instananeous bass energy to the average bass energy over roughly the past second in time. Keeping the average over a time period longer than one second would most likely result in past beat values being contained within the average, making detection of another beat more difficult. Taking the average over a time period significantly shorter than one second would result in more false positive beat detections.

The average energy is computed as a running average. This is performed using a 32 sample FIFO memory architecture. By holding the past 32 bass energy values, the beat detection module is averaging over the past 1.1 seconds, a suitable amount of time that was shown to be very effective for beat detection. If the current bass energy value is greater than the average bass energy multiplied by a scaling factor, a beat is registered. Testing revealed that for songs with a very strong beat, a scaling factor of 4 was appropriate. For songs with more subtle beats, a scaling factor of 2 revealed good results.

The FIFO memory performs the function of keeping track of the oldest bass energy value and inserting the new bass energy value. Only the new and old energy values are needed to update the average energy. Thus, with each new sample, the old energy is popped from the FIFO memory. The calculation of the updated average energy is given by equation 4.

$$avg_i = avg_{i-1} - old + \frac{new}{32} \tag{4}$$

The division by 32 is computed as a right bit shift by 5 bits. In this manner, the updated average energy can be computed in a single clock cycle with the entire beat detection algorithm taking only two clock cycles per input sample.

2.1.6 Ball Generator

The Ball Generator module takes as input individual fountain energy values, the total energy over all eight fountains, as well as a beat signal from the Beat Detection Module. The function of Ball Generator is to compute and write to memory new ball features that are related to the input signal values. Each ball contains a position and velocity in three dimensions as well as a color value. The position values are hard coded within the module as the positions of each of the eight fountains distributed throughout the floor of the Bellagio world. The direction of the fountain is given by the x and y velocities. These values are initiliazed to 0, resulting a launch direction that is straight up into the air. The vertical z velocity is proportional to the fountain energy. Thus, if the audio signal contains a lot of energy within a frequency range, the corresponding fountain will launch ball very high into the air.

The color of each ball is determined by either the `beat` signal calculated by the Beat Detection module or the total fountain energy. As input to the Ball Generator module, the `use_beat` signal controls which value determines the color. If the `use_beat` signal is high, then the color determined by the presence of a beat. If there is currently a beat, the color is given a different value than if there is not a beat. If the `use_beat` signal is low, then, the color is determined by the total fountain energy. In the current implementation, the `use_beat` signal is tied to a switch on the labkit. Thus, for songs with a strong beat, the user can select to change the color of the balls with the beats of the songs. Conversely, for songs without a clear beat, the color of the balls can be selected to be determined by the total energy of the fountains.

Once the position, color, and velocity of each ball has been determined, the Ball Generator module writes the new ball value into main memory. Thus, as output from the module there are write enable, address, and data values corresponding to `bram_wea`, `bram_addra`, and `bram_ina` signals respectively. An additional `write_done` signal is outputted to indicated when the Ball Generator module has finished writing to memory.

2.1.7 Ball Divider

The Ball Divider module performs the function of telling the Ball Generator module when to write new balls into memory. The modules gives a `go` pulse 15 times a second. This is achieved through the use of a counter to keep track of how many 27 MHz clock cycles have passed. Each time the counter reached $\frac{27{,}000{,}000}{15}$, the `go` signal is high.

2.2 Central Memory Unit (J. Lane)

The Audio, Phsyics, and Graphics modules communicate through a central memory constructed using dual-port BRAM. BRAM was chosen to provide enough memory to store a significant number of ball features while also supporting two read or write operations on the same clock cycle.

Each ball consists of a position vector, a velocity vector, and a color value. The position vector is composed of three signed 18-bit values representing x, y, and z positions. Likewise, the velocity vector is composed of three signed 18-bit values representing the x velocity, y velocity, and z velocity. Color is limited to 3-bits, providing 8 different color combinations for each ball.

All of the features combined produces a string that is 111 bits long. Four, 32x512 blocks of BRAM were wired together to produce a single block of size 128x512, providing the system with enough memory to store 512 individual balls. Figure 2 shows the layout of each element inside of the BRAM module [2].

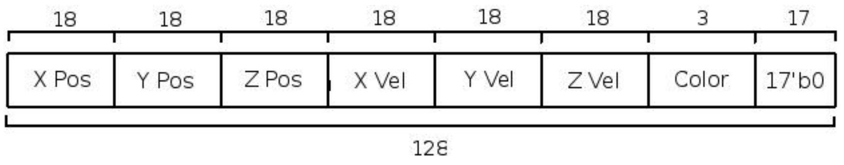

Figure 2: Single BRAM element

Communication to the central memory unit was controlled using timing signals and the dual port capabilities of the BRAM. The ball generator module writes to the BRAM once every 15 frames and otherwise does not interact with the BRAM. Thus, it requires write access to the BRAM but does not require read access. On the other hand, the physics module requires both read and write access to the BRAM. In the case that the ball generator module writes to the BRAM, the phsyics calculations are constrained to operate after the ball generator has successfully written to the BRAM. Finally, the graphics modules need only to read values from the BRAM. They can do so at any time and do not need to wait for either the audio or physics modules to complete their operations.

Port A of the BRAM was assigned to the audio and physics modules for their read and write operations. To enforce the timing constraint described above, the ball generator module outputs an enable signal, `complete`, indicating that it has finished writing to the BRAM and that it is safe to now read and write through Port A. Port B is assigned to the graphics module as a read only port, allowing the graphics modules to read from the BRAM at any time.

2.3 Physics Engine (A. Yin)

The physics engine module handles all of the collision calculations, position and velocity updates of the balls in the simulation. Every frame the physics engine would decide whether any balls are colliding or bouncing off walls and update their position and velocity accordingly. The Physics Engine interacts mininally with the rest of the system and its ports are:

- bram_wea, write enable for the BRAM which stores all of the ball information.
- bram_addra, the address of the BRAM which the Physics Engine needs to read from or write to.
- bram_ina, the inputs of the BRAM to which the Physics Engine writes.
- bram_outa, the outputs of the BRAM from which the Physics Engine reads.
- enable, signal from the graphics module notifying it that the graphics module has finished extracting a frame's information and physics calculation can resume.

The function of the Physics Engine then requires a defined simulation environment, an implementation of each individual ball objects, and a state machine for iterating through all of different balls and stages of calculation.

2.3.1 The World

Because our project aims to produce a simulation of the physical Bellagio fountains, we need to define our simulation environment and coordinate system with similar proportion. Our simulation environment, or world, is scaled from a 100x100x100m^3 cube. When looking into the screen, the horizontal is the x-axis, vertical the z-axis, and the vector coming out of the screen is the y-axis of our world's coordinate system. Further, the coordinate system is signed, meaning the exact center of our cubic world is the origin, and each of the x, y, and z-axis stretches from -50 to 50m. Further, the six sides of our world-cube is modeled as a wall from which the balls can bounce.

Both position and velocity are represented using 18-bit signed values. Values of 18 bits were chosen in order provide enough resolution during the phsyics computations. Notably, the value for gravity was very small in comparison to the maximum velocity. Thus, a large nunber of fractional bits were required to represent gravity and velocity so as not to lose information during physical computations. With sufficients amount of BRAM, position values were also represented using 18-bits. This scheme greatly simplified physics computations while only moderately increasing the amount of memory needed.

2.3.2 Implementation of the Ball Objects

The ball data structure is encapsulated as rows of 128 bits in the dual-port BRAM which include:

- 18 bits each for the x-coordinate, y-coordinate, z-coordinate of the ball.
- 18 bits each for V_x, V_y, and V_z, the velocity of the ball in each direction.
- 3 bits for the color of the ball.
- 17 bits of filler 0's which are not processed in computation.

The BRAM has 500 rows representing the 500 balls that we want to simulate. During simulation, the physics module would read the ball information from the BRAM and update it with the newly calculated values through one port. After the completed computation for one frame, a ready signal would notify the graphics module that a new frame of information is ready for read, which accesses the BRAM from the second port.

2.3.3 The Physics Behind the Simulation

Since our project is a simulation of a physical system, assumptions and calculations about the physics used in the simulation must be worked out. Our simulation models the water of the bellagio fountain with spherical balls, thus the nature of the computation is that of projectile calculation with gravitational effects. Further, to make the simulation more realistic, the balls are able to bounce off walls and each other in perfectly elastic collisions, thus requiring collision detection and calculations. The high level steps required for our physics calculations are then:

1. Detection of balls' collisions with walls.
2. Update balls' velocities with wall collisions.
3. Detection of ball-to-ball collisions.
4. Update balls' velocities with ball-to-ball collisions.
5. Update the balls' positions with their velocities and gravity.

Howver, assumptions can be made about our simulation that can simplify these procedures. First, the balls' size are uniform and small compared to the world, meaning that collisions are rare. This means that when we detect that a ball's involved with either a wall collision or a ball-to-ball collision, we can then end the collision detection stage involving that ball, since most likely it will not be involved in another collision and that we should not waste clock cycles doing most probable futile calculations.

Further, all of the collisions are perfectly elastic, the balls have the same physical attributes and are modeled as point masses. Therefore, the usually complicated collision calculation then reduces to:

- When a ball bounces off from a wall, the component of its velocity parallel to the collision is reversed.
- When two balls collide with each other, their corresponding velocity vectors switch [Ref 1].
- During the velocity update of collision detections, gravity subtracted from the z component of each ball's velocity.

2.3.4 Physics Engine State Machine

With all of the infracstructure of the engine setup, computation then becomes possible. As listed in the previous section, physics calculation requires several step, and since our project requires simulating 500 balls, computational loops are required to iterate through all of them. The work of iterating and calculating is done by the Physics Engine State Machine.

To iterate through and compare the balls, two index are kept: *next_ball_one_index* and *next_ball_two_index*, which represent the index in the BRAM of the ball currently under calculation and the ball which is used to compare against the first ball.

When the simulation starts, the state machine starts in the **DONE** state, in which writing to the BRAM is disabled *next_ball_one_index* is set to 0 while *next_ball_two_index* is set to 1. Further, in this state no calculation can be run unless the module receives the *enable* signal from the graphics module, at which the state transitions to **GET_BALL_1**.

The state will stay in **GET_BALL_1** for two cycles, which is achieved by a counter *ball_one_count*, since the data from the BRAM becomes available two cycles after the address is specified. In the second cycle that the state stays in **GET_BALL_1**, the output from the BRAM is stored in the corresponding registers (i.e. ballx_1 gets ballx_out, bally_1 gets bally_out, etc). Then if there are no more balls left after the ball we just read, the state transitions to **COLLISION_DECTECT_WALL**. Otherwise, *ball_one_count* is reset, bram_addra is set to that of the *next_ball_two_index* and state transitions to **Get_BALL_2**.

Once transitioned into **GET_BALL_2**, the state machine will also stay there for two cycles for the same reason it needs to stay in **GET_BALL_1** for two cycles. This is achieved by keeping a counter *ball_two_count*. In the second cycle in the state, the output from the BRAM is stored in the corresponding registers (i.e. ballx_2 gets ballx_out, bally_1 gets bally_out, etc). Then *next_ball_two_index* is incremented and the state transitions to **COLLISION_DECTECT_WALL**.

In **COLLISION_DETECT_WALL**, the state machine checks whether ball 1 (with index *next_ball_one_index*-1) is colliding with any wall. If it is, bram_wea goes high, bram_outa is set to equal to the updated information of ball 1 and state transitions to **COLLIDE_WALL**. Otherwise, if ball 1 is not the last ball in the BRAM, the state transitions to **COLLISION_DECTECT_BALL**. If ball 1 is the last ball in the BRAM, however, the state transitions to **UPDATE_POSITIONS**.

In **COLLISION_DECTECT_BALL**, the machine checks whether ball 1 and ball 2 are colliding by checking whether the difference between their x, y, and z coordinates are less than 2*radius. If they are colliding, the input to the BRAM takes the updated values of ball 1, bram_wea goes high and state transitions to **COLLIDE_LHC**. Otherwise, bram_addra is set to that of the current ball 1, and the input to the BRAM takes the values of the gravity-updated values of this ball 1 (which is not involved in any collision). Bram_wea is then set to high and state transitions to **GRAVITY_UPDATE**.

In **GRAVITY_UPDATE**, the machine then disables write into the BRAM (after the writing has finished). If the current ball 2 is the last ball in the bram and the current ball 1 is the last ball in the BRAM, then we reset *next_ball_one_index*=0 and *next_ball_two_index*=1, bram_addra to 0 and transitions to **UPDATE_POSITIONS**. If the current ball 2 is the last ball in the BRAM but the current ball 1 is not, then we set the *next_ball_two_index*=*next_ball_one_index*+1 and bram_addra to the *next_ball_one_index* and transitions to **GET_BALL_1**.

In **COLLIDE_WALL**, bram_wea is set to 0. If there are more balls after the current ball 1, *next_ball_two_index*=*next_ball_one_index*+1 and bram_addra is set to the *next_ball_one_index* and state transitions to **GET_BALL_1** again. However, if there are no more balls after the current ball 1, then we must have checked all of the balls and pairs for collision and thus state transitions to **UPDATE_POSITIONS**.

The state stays in **COLLIDE_LHC** for 4 cyles. The first two cycles to finish writing ball 1's updated information the BRAM and the second two cycles to write ball 2's updated information into the BRAM. This is achieved by a counter *write_lhc_count* which incrmenets every cycle. Nothing happens in the first cycle, the machine just waits. In the second cycle, we give the BRAM the post-collision information of the updated ball 2, set bram_addra to that of the current ball 2. In the third cycle, we wait again. In the fourth cycle, write has finished for the second ball; we set disable write to the BRAM, and depending on whether there are more balls after the current ball 1, state either transitions to **GET_BALL_1** after setting *next_ball_two_index*=*next_ball_one_index*+1 and bram_addra set to the *next_ball_one_index*, or **UPDATE_POSITIONS** after setting *next_ball_one_index*=0 and *next_ball_index*=1.

The state machine arrives the **UPDATE_POSITIONS** state when all collision detection and calculation has finished. The only thing left to do to finish the given iteration of physics calcuation is to update each ball's positions. This is done by setting bram_addra to the index of the ball that needs to be modified and the inputs to be the updated ball information. The state then transitions to **WRITE_POSITIONS**.

The state machine stays in **WRITE_POSITIONS** for two cycles because it takes two cycles from the specification of a BRAM address to when data is written into that BRAM address. In the second cycle the machine stays in this cycle, writing into the BRAM is disabled. If the current bram_addra is that of the last ball in the bram, we're done and state transitions into **DONE**. Otherwise, we have more balls to update, thus bram_addra is incremented and state transitions to **UPDATE_POSITONS**.

Note the use of *next_ball_one_index* and *next_ball_two_index* as ways of keeping track of the balls under comparison and calculation. *next_ball_one_index* is incremented in **GET_BALL_1** while *next_ball_two_index* is incremented in **GET_BALL_2**. In this way, the state machine can efficiently iterates through all of the balls.

Further, notice that under each iteration with a given ball 1, only one collision can happen to that ball 1 (wall collision or ball-to-ball collision). This is consistent with our physics assumption that the balls are small compared to the world and a ball involved in two collisions is unlikely.

The behaviors described above are translated into the Physics Engine State Machine state transition diagram, shown below.

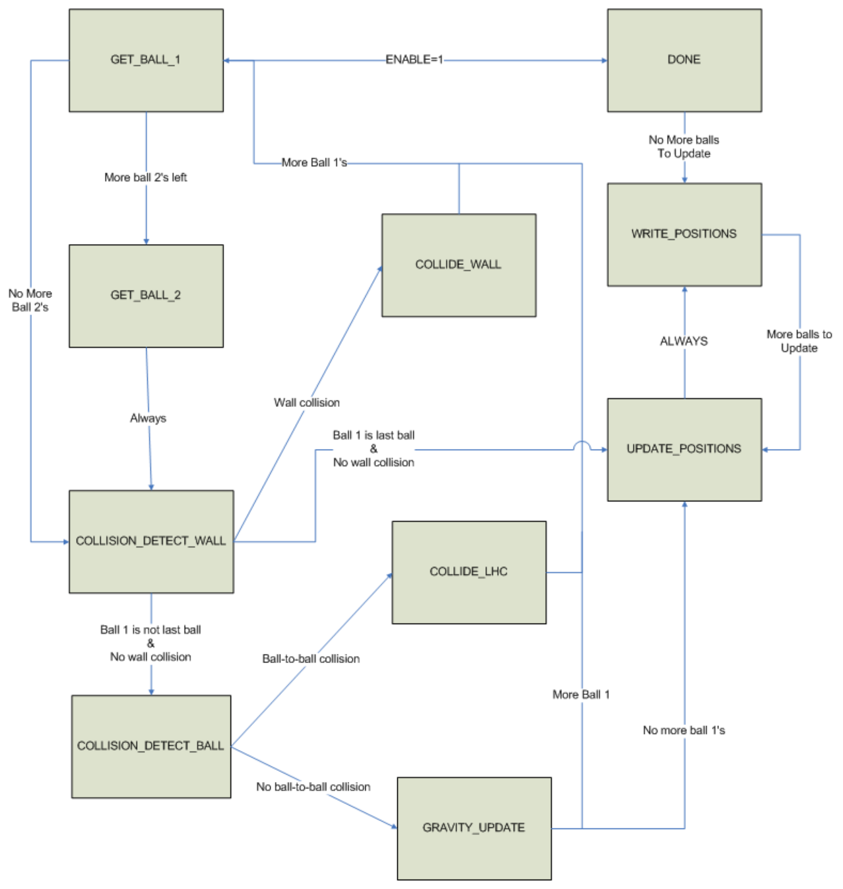

Figure 3: Physics Engine state mahcine transition diagram

2.4 Graphics (G. Rossick)

This module takes the data calculated by the first two modules and creates a graphical representation of the scene. We use a modified version of raster graphics to accomplish our goal. Since a ball just looks like a circle from every angle with proper shading we can create the illusion of a 3D scene without actually taking the processing power to render 3D. The ball data is read by the projection module from the bram. The projection module then converts the 3D representation of a ball into a 2D pixel value on the screen. Next the shading module takes the location of the center of the ball calculated by the projection module and draws a ball around it. Meanwhile the floor module calculates which pixels are part of the floor. All of this data is fed into the displayFSM which writes it to the ZBT using a double buffer technique. One of the ZBT memories is written to with new pixel data while the other is used for displaying from.

2.4.1 DisplayFSM

The displayFSM module writes pixel values into the ZBT memory. The module takes floor and ball positions, depths, and colors in order to output an x and y location as well as a color to write into the memory. It consists of 3 different states: cleaningBuffer, waitForInput, and writing. Every time newFrame goes high, the state instantly switches to cleaningBuffer. In this state, all locations in the writing buffer and the zbuffer are set to initial values. For the writing buffer this means all color locations for each x and y location on the screen are set to

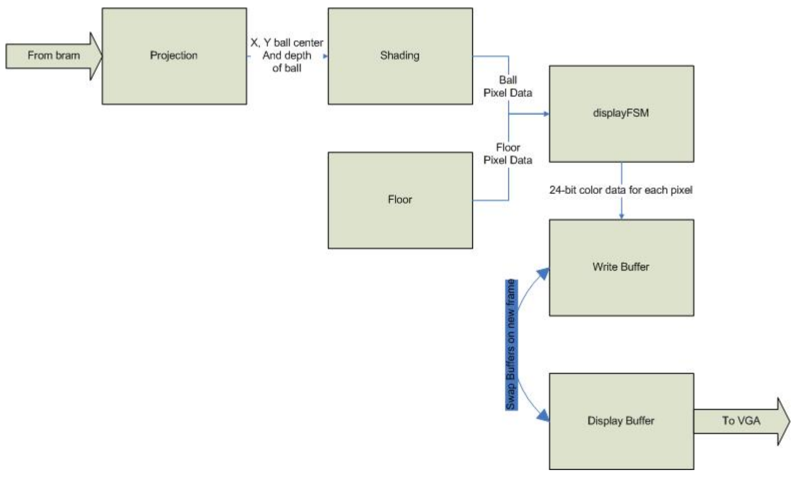

Figure 4: Graphics System Block Diagram

0. In the zbuffer each bit is set to 1 at each address. Once this task is complete the Display FSM switches to waitForInput mode. Here the FSM waits for either the floorValid or ballValid signals to go high. When they do, the data for the valid pixel type is latched into registers, and a floorAccepted or ballAccepted signal is sent as an output to the floor or shading module. The FSM then moves into the writing stage where the current depth value is checked against the value in the zbuffer. If the current value is smaller, the new pixel is written into the ZBT and the new depth is written into the zbuffer. The state then switches back to waitForInput. This cycle continues until there are no more pixels to write or the next frame starts. At that point the state returns back to cleaningBuffer.

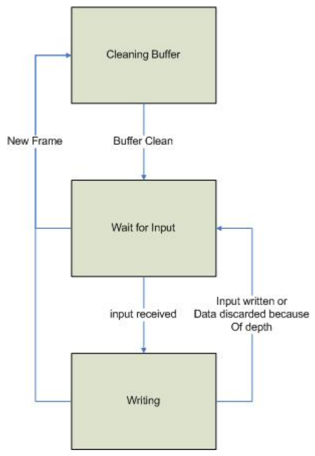

Figure 5: Display FSM State Machine

2.4.2 sramController

The sramController sets up a double buffer system that designates one ZBT memory as the display buffer and one as the write buffer. Every time there is a new frame and the flip signal goes high these memories swap roles. Each pixel has a location in memory equivalent to its y coordinate followed by its x coordinate. In that location 24 bits of color data is stored. Every clock cycle the color data corresponding to the write_x and write_y inputs is written into the write buffer. Since the ZBT writes data into the memory two clock cycles after its write enable is triggered, all writes must be delayed for 2 clock cycles In addition, every clock cycle the values input on the read_x and read_y locations are set as the address of the display buffer. Two clock cycles later the color value for those pixels is available to be sent to the VGA monitor.

2.4.3 Projection

The projection module reads ball data from the central bram that the physics and audio modules update. It then performs a transform on the x,y, and z coordinates to translate that data into x and y screen positions with a certain camera location and angle. The projection module currently uses a simplified version of the projection with the angles all equal to 0. Using 0 for the angles simplifies all of the sine and cosine calculations to either 0 or 1. This greatly simplifies the calculation, but limits the possible camera locations. This is all done using combinational logic and is guaranteed to be completed before the shading module needs the ball values. Every time the shading module is ready to process a new ball it sets the ready signal to be high. At this point the projection module increments by one. This process continues until all of the balls have been processed. When this happens projection module sets the done signal to high and the bram address to zero. It then waits for the newFrame signal before repeating the process.

2.4.4 Shading

There are two different implementations of the shading module. The first is flat shading. With this method a circle is drawn around the point supplied by the projection module in a solid flat color. There was not time to implement the second type of shading using sprites. With this method there would have been a sprite for each of the 16 radius sizes and 8 ball colors The shading module would have accessed these sprites and sent them to the displayFSM module. This would have allowed for much more realistic balls that looked 3D opposed to circles. The flat shading method was used to create the shading module. It is implemented using a state machine with 4 different states: initialize, calculateValid, waitForRam, and increment. It starts in the initialize state and stays in this state until the done signal is low. At this point it latches new ball values into its registers and sends the ready signal to the projection module to let it know to start preparing the next ball for processing. Next the module iterates through each of the pixels within a radius's distance from the center of the ball and determines if they are within the ball. xsquared+ysquared¡radius squared is calculated for the current pixel. If it is pixelValid is set to high and the state changes to waitForRam. In this state the module waits for the displayFSM to accept the pixel before moving to the increment state. If the pixel is not within the ball, the state moves directly from the calculateValid state to the increment state. In the increment state the pixel being examined is moved to the next value. This continues until all pixels within a radius's distance of the ball have been checked. The state then moves back to the calculate valid section to get the next ball's data and repeat the process. This continues until the projection module sends the done signal.

2.4.5 Floor

The floor module calculates all of the floor pixels and sends them to the displayFSM. It starts this process by projecting the corners of the floor onto the screen using the same method as the projection module. The module then interpolates along the line connecting the corners of the floor. If a line is drawn through the screen it will either pass through no interpolated points or exactly two. If it passes through 0 points then there are no floor pixels on this line and the next line is tried. If it passes through two interpolated points then all pixels between those two

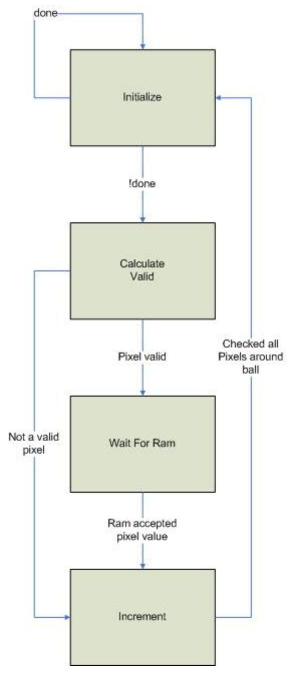

Figure 6: Shading FSM

points are part of the floor and are sent to the display FSM. The next line is then checked. This process continues until all lines on the screen have been checked for floor pixels. The module then waits for the newFrame Signal before starting again.

3 Testing

3.1 Audio Design (J. Lane)

Modules in the audio system design were tested by simulation through the use of ModelSim as well as test frameworks measured using a Digital Logic Analyzer. Additionally, prior testing of audio algorithms was initially conducted through using a sofware implemenation of the module designs. Aside from difficulties experienced concerning the FFT module, all other modules were succesfully tested

3.1.1 FFT

The FFT was constructed by the Xilinx software and tested using simpel test framewords and viewed using a logic analyzer. Tests primarily consisted of using a generated 750 Hz pure tone as input to the FFT. In this case, the clock enable signal was set to a 48 kHz pulsed signal, the real input values were set to the high order eight bits of the sine wave data, and the imaginary input values were set to 0. Outputs from the FFT were viewed using the logic analyzer.

While initial testing of a 128 ans 1024 point FFTs were successful, further testing later on revealed inconsistent and faulty behavior. FFT modules that were previously valided through early testing proved no longer functional later in the design process. While FFTs of varying point sizes were constructed and tested on multiple different lab kits, results were inconsistent and rarely correct.

At one point in the design process, only a 1024 point FFT module was working. The audio modules were then redesigned to use a 1024 point FFT instead of a 128 point FFT. However, the 1024 point FFT also stopped producing valid results after further testing was conducted.

Common issues with FFT outputs were uniform noise distributed throughout the entire frequency spectrum when a pure tone was given as input and the lack of any output at all.

Had early testing revealed the FFT module to be unreliable, there would have been sufficient time to build a FFT module. As a result of the failure of the FFT module, the entire audio system was redesigned. Time constraints forced the redesigned system to a much simpler version. Instead of an FFT, a simple low pass filter was used to extract low frequency audio. Each fountain's energy no longer corresponded to a certain frequency but rather the current bass energy. Balls were launched from consective fountains when the bass audio energy was above a certain theshold.

3.1.2 Magnitude, Ball Generator, and Bin Manager

The Magnitude, Ball Generator, and Bin Manager modules were all tested through simulation and the use of a wide range of input signals. Realistic scenarious and data values were given as inputs to each module and the results were analyzed using ModelSim. ModelSim was unable to accurately simulate the BRAM memory used by the Ball Generator module. As a result, a simulated BRAM was constructed using registers for use during simulation while the actual BRAM was used on the labkit.

3.1.3 Software Implementation

Before decisions were made on the audio system, a software implementation was created and tested in the python and C++ programming languages. A non real-time implementation of the beat detection algorithm was constructed using python and tested on a variety of different music. Energy values from these tests were then used to validate the conversions between fountain energy and velocity.

To assist in determining whether these conversions were appropriate, a three dimensional world was constructed using the C++ programming language and the OpenGL graphical library. Accurate physics were included in the model to give valuable feedback on the behavior of the fountains.

Finally, the Bellagio world was constructed to be proportional to the actual Bellagio fountains. To assist in converting between real world values and Bellagio world values, a converter was constructed using python. This tool prooved to be extremely valuable for determining constant values such as gravity and upper and lower position bounds.

3.2 Physics Engine (A. Yin)

3.2.1 Testing of Proposed Design

As one may notice in the desciption of the state machine of our proposed design, `bram_wea` and `bram_addra` are always set in the state previous to the state in which write read/write is supposed to happen. This is because of the two-clock cycle delays in the input into and the output from the bram. This timing issue was troublesome to get right and often produced unrealistic results in model-sim such as the entries inside BRAM having the same values, the wrong correspondence of an entrie's values with its index, etc.

3.2.2 Testing of Implemented Design

In our alternative project, the physics engine updates the balls' velocities and positions in parallel. The velocities of the balls under simulation are under the effects of gravity (the y-velocity, since in this alternative simulation the world's 2-D, is subtracted in every frame by gravity). Wall-collision detection is implemented and balls' velocities reverse appropriately upon bouncing off walls. However, because of the timing constraint, ball-to-ball collisions detection and calcuations are not implemented.

The biggest difference in the physics engine of our proposed design and our actual finished project is the lack of state machine in our actual implemented project. The implemented project holds the balls in 16 registers, which allow for parallel access. This then effects the state machine's architecture designed for sequential interaction with the BRAM. Thus with a tight timing constraing, we were unable to implement the ball-to-ball collision detection and calculations.

3.3 Graphics (G. Rossick)

The projection, shading and displayFSM modules were all tested in ModelSim to ensure they were functioning properly. In addition the modules were tested graphically using the double buffer display system to ensure that the modules were displaying properly on the screen. The major hurdle that we ran into with testing the graphics module was that the entire system was written in verilog before any testing began. This made it very difficult to test individual parts because of the complexity of the design. It would have been a much better plan to test as we went instead of saving all the testing for the end.

4 Conclusion

Due to the difficulties experienced during testing concerning the FFT module and implementation of the ZBT double buffer, very little time was avialable for complete system testing. Consequently, the resulting final project was dramatically simplified over the proposed system design. The lack of a graphical environment made thorough testing incredibly difficult. The decision to switch to a much simpler graphical environment was made late in the implementation process when there was not enough time to reconstruct the modules as needed.

More thorough research of the available technologies, such as the IPCoreGen generated modules, should have been conducted to ensure valid and consistent performance. Although the FFT module failed to perform correctly, further work to construct a valid FFT module could serve as a replacement in the otherwise successful set of audio modules. Furthermore, testing of the complete system, as opposed to individual modules, is important to confirm propoer communication and compatibility bewteen modules.

5 Appendix A: Audio Verilog Code

```
module energy #(parameter N_BITS = 8,
       parameter INDEX_BITS = 7,
       parameter ENERGY_BITS = (N_BITS*2) + 1)


   /* The energy module computes the energy of the FFT real and
    * imaginary values.
    */

   (input wire clk, reset, new_sample,        // new sample should be a pulse
    input wire [N_BITS-1:0] xk_real, xk_imag, // outputs from FFT module
    input wire [INDEX_BITS-1:0] index,        // output index from FFT module
    output wire [INDEX_BITS-1:0] index_out,   // delayed index value
    output reg [ENERGY_BITS-1:0] energy);     // energy

   reg [N_BITS-1:0]              real_squared;
   reg [N_BITS-1:0]              imag_squared;

   synchronize #(.NSYNC(2), .NBITS(INDEX_BITS)) index_sync(.clk(clk), .in(index),
        .out(index_out));

   always @(posedge clk) begin
      if (reset) begin
   energy <= 0;
   //index_out <= 0;
      end

      if (new_sample) begin

   /* First clock cycle */
   real_squared <= xk_real * xk_real;
   imag_squared <= xk_imag * xk_imag;

   /* Second clock cycle */
   energy <= real_squared + imag_squared;
      end

   end // always @ (posedge clk)

endmodule // energy


module bin_manager #(parameter LOGN_BINS = 7, parameter N_BITS = 17, parameter
    NUM_VALS = 16)
   (input clk, reset,
    input [N_BITS-1:0] energy,                      // energy from FFT at index "
        index"
    input [LOGN_BINS-1:0] index,                    // FFT bin index
    input [2:0] fountain_index,                     // requested fountain index
    output reg [N_BITS-1:0] energy_out,             // energy at the requested
        fountain index
    output reg bass_ready,                          // ready signal for bass
        energy and beat_detection
    output wire [N_BITS-1:0] bass_energy,           // bass energy sent to beat
        detection
    output reg [LOGN_BINS + N_BITS:0] total_energy  // total energy sent to ball
        generator
    );

// Fountain 0 : Indices 0      : Bitshift 4
// Fountain 1 : Indices 1-2    : Bitshift 5
// Fountain 2 : Indices 3-6     : Bitshift 6
// Fountain 3 : Indices 7-10    : Bitshift 6
// Fountain 4 : Indices 11-18   : Bitshift 7
// Fountain 5 : Indices 19-26   : Bitshift 7
// Fountain 6 : Indices 27-34   : Bitshift 7
// Fountain 7 : Indices 35-42   : Bitshift 7

   wire [N_BITS-5:0]          bit_shifted_energy;
   reg [2:0]            my_fountain_index;
   reg [16:0]             sum_array[7:0]; // 8, 17 bit registers
   reg [2:0]            delayed_fountain_index;
   reg [4:0]            counter;        // samples per bin

   /* All values need to be shifted by at least 4 bits */
   assign bit_shifted_energy = energy >>> 4;

   reg [N_BITS-1:0]           fountain0;
   reg [N_BITS-1:0]           fountain1;
   reg [N_BITS-1:0]           fountain2;
   reg [N_BITS-1:0]           fountain3;
   reg [N_BITS-1:0]           fountain4;
   reg [N_BITS-1:0]           fountain5;
   reg [N_BITS-1:0]           fountain6;
   reg [N_BITS-1:0]           fountain7;

   assign bass_energy = fountain0;

   /* 'Ready' to calculat the new averages */
   wire               ready;
   reg                delayed_ready;
   assign ready = (counter == NUM_VALS);

   wire               clear;
   synchronize #(.NSYNC(3), .NBITS(1)) my_updated_samples(.clk(clk), .in(
       delayed_ready), .out(clear));

   always @ * begin
      case(index)
  7'b000_0000 : my_fountain_index = 3'b000;  // 0
  7'b000_0001 : my_fountain_index = 3'b001;  // 1
  7'b000_0011 : my_fountain_index = 3'b010;  // 2,3
  7'b000_0111 : my_fountain_index = 3'b011;  // 4,5,6
  7'b000_1011 : my_fountain_index = 3'b100;  // 7,8,9,10
  7'b001_0011 : my_fountain_index = 3'b101;
  7'b001_1011 : my_fountain_index = 3'b110;
  7'b010_0011 : my_fountain_index = 3'b111;
  7'b010_1011 : my_fountain_index = 3'b000;
  default : my_fountain_index = my_fountain_index;
      endcase // case (index)

      case(fountain_index)
  3'b000 : energy_out = fountain0;
  3'b001 : energy_out = fountain1;
  3'b010 : energy_out = fountain2;
  3'b011 : energy_out = fountain3;
  3'b100 : energy_out = fountain4;
  3'b101 : energy_out = fountain5;
  3'b110 : energy_out = fountain6;
  3'b111 : energy_out = fountain7;
  default energy_out = fountain0;
      endcase // case (fountain_index)
   end
```

```
always @(posedge clk) begin
   delayed_fountain_index <= my_fountain_index;
   delayed_ready <= ready;
   bass_ready <= delayed_ready;
   if(reset) begin
total_energy <= 0;
//energy_out <= 0;
counter <= 0;
sum_array[0] <= 0;
sum_array[1] <= 0;
sum_array[2] <= 0;
sum_array[3] <= 0;
sum_array[4] <= 0;
sum_array[5] <= 0;
sum_array[6] <= 0;
sum_array[7] <= 0;
   end

   /* Consider only values between 0 and 42 */
   else if (index < 43 && index > 0) begin
sum_array[delayed_fountain_index] <= sum_array[delayed_fountain_index] +
    bit_shifted_energy;

   end
   if (counter == NUM_VALS) begin
counter <= 0;
   end
   else if (index == 7'b010_1011) begin // 42
counter <= counter + 1;
   end

   /* Calculate new average values */
   if (ready) begin
sum_array[1] <= sum_array[1] >>> 1;
sum_array[2] <= sum_array[2] >>> 2;
sum_array[3] <= sum_array[3] >>> 2;
sum_array[4] <= sum_array[4] >>> 3;
sum_array[5] <= sum_array[5] >>> 3;
sum_array[6] <= sum_array[6] >>> 3;
sum_array[7] <= sum_array[7] >>> 3;
   end

   /* Set new values to the fountains and reset summation arrays to 0 */
   else if (delayed_ready) begin
fountain0 <= sum_array[0][N_BITS-1:0];
fountain1 <= sum_array[1][N_BITS-1:0];
fountain2 <= sum_array[2][N_BITS-1:0];
fountain3 <= sum_array[3][N_BITS-1:0];
fountain4 <= sum_array[4][N_BITS-1:0];
fountain5 <= sum_array[5][N_BITS-1:0];
fountain6 <= sum_array[6][N_BITS-1:0];
fountain7 <= sum_array[7][N_BITS-1:0];

sum_array[0] <= 0;
sum_array[1] <= 0;
sum_array[2] <= 0;
sum_array[3] <= 0;
sum_array[4] <= 0;
sum_array[5] <= 0;
sum_array[6] <= 0;
sum_array[7] <= 0;
   end // if (delayed_ready)
   end // always @ (posedge clk)

endmodule // divider

module beat_detector #(parameter N_BITS = 17) (input wire clk, reset, ready,
                 input wire [N_BITS-1:0] instant_bass_energy,
                 output reg beat);


   /* The beat detector module compares the instantaneous beat energy
      with the average bass energy over the previous second. If the
      instant bass energy is greater than C * the average bass energy,
      then we have a beat. */

   wire [N_BITS:0]              c_average_energy;
  reg [N_BITS-1:0]              average_energy;
   assign c_average_energy = average_energy <<< 1;

   wire [N_BITS-6:0]               bit_shifted_instant_energy;
   assign bit_shifted_instant_energy = instant_bass_energy >>> 5;

   wire                 delayed_ready;
   synchronize #(.NSYNC(2), .NBITS(1)) index_sync(.clk(clk), .in(ready), .out(
       delayed_ready));

   /* Create a FIFO with 32 samples. The 32 samples equates to the previous 1.3
       seconds of music */
   wire [11:0]                  old_energy;
  wire                  complete;
   wire [11:0]                  minus_energy;
   assign minus_energy = (complete) ? old_energy : 12'b0000_0000_0000;
   fifo #(.LOGSIZE(5), .WIDTH(12)) my_fifo(.clk(clk), .reset(reset), .wr(
       delayed_ready), .rd((ready && complete)), .din(bit_shifted_instant_energy),
            .full(full), .overflow(overflow), .empty(empty), .dout(old_energy));

   reg [4:0]                counter;

   assign complete = (counter == 6'b10_0000);

   always @(posedge clk) begin
      if (reset) begin
   beat <= 0;
   average_energy <= 0;
   counter <= 0;
      end

      if (ready) begin
   counter <= (counter < 6'b10_0000) ? counter + 1 : counter;

   if (complete) begin
      beat <= (instant_bass_energy > c_average_energy) ? 1 : 0;
   end
      end

      if (delayed_ready) begin
      average_energy <= average_energy - minus_energy +
          bit_shifted_instant_energy;
      end
   end // always @ (posedge clk)

endmodule // beat_detector

module ball_generator (input wire clk, reset, ball_enable,
            input use_beat,
            input wire beat,
            input wire [16:0] fountain_energy,
            input wire [23:0] total_energy,
            output reg [2:0]  fountain_index,
            output wire [127:0] bram_ina,
            output reg [8:0] bram_addra,
            output reg bram_wea,
            output reg complete);

/*****************************************************
 * The ball generator writes into the BRAM a new ball
 * considering information gained from beat, fountain
 * energy, and total energy
 *****************************************************/

   parameter signed NEG_WALL = 18'b10_0001_1110_1011_1000;          // -47 meters
   parameter signed [17:0] RADIUS = 18'b00_0000_0101_0001_1110;     // 0.5 meters
   parameter signed [17:0] TWO_RADIUS = 18'b00_0000_1010_0011_1101;// 1.0 meters

   reg signed [17:0]       x_in, y_in;                          // New pos and
       velocity
   wire signed [17:0]       z_in, vx_in, vy_in, vz_in;
   wire [2:0]          color_in;                           // New color
   assign bram_ina = {x_in, y_in, z_in, vx_in, vy_in, vz_in, color_in, 17'b0};
   //reg [127:0]          bram_outa;

/* Each of the fountains has a certain location in the world.
 * The index of the fountain dictates the balls x and y position.
 * All of the balls will start at the same z location.
 */

   /* Assign x and y position based off of index */
   always @(fountain_index) begin
      case(fountain_index)
  3'b000 : begin x_in = 18'b10_0111_0101_1100_0010; y_in = 18'
      b00_0011_0011_0011_0011; end // (-.77, .1)
  3'b001 : begin x_in = 18'b10_1110_0110_0110_0110; y_in = 18'
      b11_1100_1100_1100_1100; end // (-.55, -.1)
  3'b010 : begin x_in = 18'b11_0101_0111_0000_1010; y_in = 18'
      b00_0011_0011_0011_0011; end // (-.33, .1)
  3'b011 : begin x_in = 18'b11_1100_0111_1010_1110; y_in = 18'
      b11_1100_1100_1100_1100; end // (-.11, -.1)
  3'b100 : begin x_in = 18'b00_0011_1000_0101_0001; y_in = 18'
      b00_0011_0011_0011_0011; end // (.11, .1)
  3'b101 : begin x_in = 18'b00_1010_1000_1111_0101; y_in = 18'
      b11_1100_1100_1100_1100; end // (.33, -.1)
  3'b110 : begin x_in = 18'b01_0001_1001_1001_1001; y_in = 18'
      b00_0011_0011_0011_0011; end // (.55, .1)
  3'b111 : begin x_in = 18'b01_1000_1010_0011_1101; y_in = 18'
      b11_1100_1100_1100_1100; end // (.77, -.1)
  default : begin x_in = 18'b10_0111_0101_1100_0010; y_in = 18'
      b00_0011_0011_0011_0011; end// (-.77, .1)

      endcase // case (fountain_index)
   end // always @ (fountain_index)

   /* Assign z_value */
   assign z_in = NEG_WALL + TWO_RADIUS;

   /* For now, make all the balls green */
   assign color_in = 3'b010;

   /* Also for now, make the x and y velocities 0 */
   assign vx_in = 0;
   assign vy_in = 0;

   /* Set the launch velocity */
   assign vz_in = (use_beat) ? {7'b00_0000_0, fountain_energy[16:6]} : {7'
       b00_0000_0, total_energy[23:13]};

   /* Simulated BRAM for modelsim */
   model_bram my_bram(.addra(bram_addra), .clk(clk), .dina(bram_ina), .wea(
       bram_wea), .douta(bram_outa));

   always @(posedge clk) begin

      if (reset) begin
   bram_addra <= 0;
   fountain_index <= 0;
   bram_wea <= 0;
   complete <= 0;
      end

      /* new sample. Stop at 500 balls */
      if (ball_enable) begin
   bram_wea <= 1;
   if (bram_addra > 500) bram_addra <= 0;
      end

      /* done writing all new balls to memory */
      if (fountain_index == 3'b111) begin
   bram_wea <= 0;
   fountain_index <= 0;
   complete <= 1;
      end
      else complete <= 0;

      /* Iterate through the address */
      if (bram_wea) begin
   fountain_index <= fountain_index + 1;
   bram_addra <= bram_addra + 1;
      end

   end // always @ (posedge clk)

endmodule // ball_generator


/*
 * Simulation BRAM
 */
module model_bram #(parameter LOGSIZE=9, WIDTH=128)
   (input wire [LOGSIZE-1:0] addra,
    input wire clk,
    input wire [WIDTH-1:0] dina,
    output reg [WIDTH-1:0] douta,
    input wire          wea);

   reg [127:0]        mem[511:0]; // 512, 128 bit registers
   reg [LOGSIZE-1:0]       saved_addr;

   wire signed [17:0]        z0, z1, z2, z3, z4, z5, z6, z7;
   assign z0 = mem[0][37:20];
   assign z1 = mem[1][37:20];
   assign z2 = mem[2][37:20];
  assign z3 = mem[3][37:20];
  assign z4 = mem[4][37:20];
  assign z5 = mem[5][37:20];
  assign z6 = mem[6][37:20];
  assign z7 = mem[7][37:20];

   always @(posedge clk) begin

      if (wea) mem[addra]  <= dina;
      douta <= mem[addra];
   end

endmodule

`timescale 1ns / 1ps
//
    ////////////////////////////////////////////////////////////////////////////////

// Company: 6.111
// Engineer: Joseph Lane
//
// Create Date:    19:58:07 09/28/2009
// Design Name:
// Module Name:    ball_divider
// Project Name: Bellagio Fountains
// Target Devices:
// Tool versions:
// Description: Creates a 15Hz enable pulse
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
    ////////////////////////////////////////////////////////////////////////////////

module ball_divider #(parameter PERIOD = 1799999) (input clk, reset, output
    ball_enable);

   reg[20:0] counter;
   assign ball_enable = (counter == PERIOD);

   always @(posedge clk) begin
      if(reset) counter <= 0;
      else counter <= (counter == PERIOD) ? 0 : counter + 1;
   end
endmodule // ball_divider
```

6 Appendix B: Physics Verilog Code

```
`timescale 1ns / 1ps
//
    ////////////////////////////////////////////////////////////////////////////////

// Company:
// Engineer:
//
// Create Date:    20:46:36 12/05/2009
// Design Name:
// Module Name:    collision_detection_state
// Project Name:
// Target Devices:
// Tool versions:
// Description: GOOGOL STATE LHC
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
   ////////////////////////////////////////////////////////////////////////////////

module collision_detection_state(input wire clk, reset);
   parameter COLLISION_DETECT = 3'b000;
   parameter GET_BALL_1 = 3'b001;
   parameter GET_BALL_2 = 3'b010;
   parameter UPDATE_POSITIONS = 3'b011;
   parameter COLLIDE_WALL = 3'b100;
   parameter COLLIDE_LHC = 3'b101;
   parameter WRITE_POSITIONS = 3'b110;
   parameter DONE = 3'b111;
   parameter signed [17:0] RADIUS = 18'b00_0000_0101_0001_1110;     // 0.5 meters
   parameter signed [17:0] TWO_RADIUS = 18'b00_0000_1010_0011_1101;// 1.0 meters
   parameter signed [17:0] NEG_WALL = 18'b10_0001_1110_1011_1000;  // -47 meters
   parameter signed [17:0] POS_WALL = 18'b01_1110_0001_0100_0111;  // 47 meters


   /* Input and output from the BRAM */
   reg [127:0] bram_ina;
   wire [127:0] bram_outa;

   /* Assigned values extracted from the BRAM input and output */
   reg signed [17:0]  ballx_in, bally_in, ballz_in, ballvx_in, ballvy_in,
       ballvz_in;
   wire [2:0]    ball_in_color;

   wire [17:0]   ballx_out, bally_out, ballz_out, ballvx_out, ballvy_out,
       ballvz_out, ball_out_color;
   reg [2:0]   ball_out_color;

   assign {ballx_out, bally_out, ballz_out, ballvx_out, ballvy_out, ballvz_out,
       ball_out_color} = bram_outa[127:17];
   assign bram_ina = {ballx_in, bally_in, ballz_in, ballvx_in, ballvy_in,
       ballvz_in, ball_in_color, 17'b0};

   /* Individual balls used for comparison and updates*/
   reg [127:0] ball_1, ball_2;
   wire signed [17:0] ballx_1, bally_1, ballz_1, ballvx_1, ballvy_1, ballvz_1;
   reg signed [17:0]  newx_1, newy_1, newz_1, newvx_1, newvy_1, newvz_1;
   reg signed [17:0]  newx_2, newy_2, newz_2, newvx_2, newvy_2, newvz_2;
   wire signed [17:0] ballx_2, bally_2, ballz_2, ballvx_2, ballvy_2, ballvz_2;
   wire [2:0]  ball_color_1, ball_color_2;
   assign {ballx_1, bally_1, ballz_1, ballvx_1, ballvy_1, ballvz_1, ball_color_1}
       = ball_1[127:17];
   assign {ballx_2, bally_2, ballz_2, ballvx_2, ballvy_2, ballvz_2, ball_color_2}
       = ball_2[127:17];

   reg [2:0] state;
   wire       ball_2_done;
   reg        ball_one_count;
   reg        ball_two_count;
   reg [1:0] lhc_write_count;

   wire       collide_x, collide_y, collide_z, collide_lhc, collide_wall;
   wire [4:0] collide_map;
   wire signed [17:0] x_dif, y_dif, z_dif;

   reg [8:0] next_ball_one_index, next_ball_two_index, bram_addra;

   /* Wall collision conditions */
   assign collide_x = ((ballx_1 - RADIUS <= NEG_WALL) || (ballx_1 + RADIUS >=
       POS_WALL));
   assign collide_y = ((bally_1 - RADIUS <= NEG_WALL) || (bally_1 + RADIUS >=
       POS_WALL));
   assign collide_z = ((ballz_1 - RADIUS <= NEG_WALL) || (ballz_1 + RADIUS >=
       POS_WALL));
   assign collide_wall = (collide_x || collide_y || collide_z);

   /* Ball collision conditions */
   assign x_dif = (ballx_1 >= ballx_2) ? (ballx_1 - ballx_2) : (ballx_2 - ballx_1
       );
   assign y_dif = (bally_1 >= bally_2) ? (bally_1 - bally_2) : (bally_2 - bally_1
       );
   assign z_dif = (ballz_1 >= ballz_2) ? (ballz_1 - ballz_2) : (ballz_2 - ballz_1
       );
   assign collide_lhc = ((x_dif <= TWO_RADIUS) && (y_dif <= TWO_RADIUS) && (z_dif
       <= TWO_RADIUS));

   assign collide_map = {collide_x, collide_y, collide_z, (collide_lhc && !
       collide_wall)};

   reg [8:0] num_active_balls;
   assign ball_2_done = (next_ball_two_index == num_active_balls);

   assign bram_wea = (state == COLLISION_DETECT & (collide_lhc | collide_map !=
       4'b0000)) |
         (state == COLLIDE_LHC & (write_lhc_count != 2'b11)) |
         (state == WRITE_POSITIONS & (position_write_count == 1'b0)) |
         (state == UPDATE_POSITIONS);



   /* Update ball velocities if collision */
   always @* begin
      case (collide_map) begin
   4'b0000 : {newx_1, newy_1, newz_1, newvx_1, newvy_1, newvz_1, new_color_1} = {
       ballx_1, bally_1, ballz_1, ballvx_1, ballvy_1, ballvz_1, ball_color_1};
   {newx_2, newy_2, newz_2, newvx_2, newvy_2, newvz_2, new_color_2} = 111'b0;

   4'b1000 : {newx_1, newy_1, newz_1, newvx_1, newvy_1, newvz_1, new_color_1} = {
       ballx_1, bally_1, ballz_1, -ballvx_1, ballvy_1, ballvz_1, ball_color_1};
   {newx_2, newy_2, newz_2, newvx_2, newvy_2, newvz_2, new_color_2} = 111'b0;

   4'b0100 : {newx_1, newy_1, newz_1, newvx_1, newvy_1, newvz_1, new_color_1} = {
       ballx_1, bally_1, ballz_1, ballvx_1, -ballvy_1, ballvz_1, ball_color_1};
   {newx_2, newy_2, newz_2, newvx_2, newvy_2, newvz_2, new_color_2} = 111'b0;

   4'b0010 : {newx_1, newy_1, newz_1, newvx_1, newvy_1, newvz_1, new_color_1} = {
       ballx_1, bally_1, ballz_1, ballvx_1, ballvy_1, -ballvz_1, ball_color_1};
   {newx_2, newy_2, newz_2, newvx_2, newvy_2, newvz_2, new_color_2} = 111'b0;

   4'b0001 : {newx_1, newy_1, newz_1, newvx_1, newvy_1, newvz_1, new_color_1} = {
       ballx_1, bally_1, ballz_1, ballvx_2, ballvy_2, ballvz_2, ball_color_1};
   {newx_2, newy_2, newz_2, newvx_2, newvy_2, newvz_2, new_color_2} = {ballx_2, bally_2,
       ballz_2 , ballvx_1 , ballvy_1 , ballvz_1 , ball_color_2 };

      endcase // case (collide_map)

   end // always @ *




   /*********************************************************
    *
    * Collision Detection
    * GET_BALL_1 : retrieves mem data for lower index
    * GET_BALL_2 : retrieves mem data for upper index
    * COLLISION_DETECT : checks both ball values to see if they
    *  are colliding with, first, the floor and then each other
    * -> GET_BALL_1 if done checking all pairs for ball 1
    * -> GET_BALL_2 if done checking ball 1 with ball 2
    * -> COLLIDE_WALL if ball 1 collides with wall
    * -> COLLIDE_LHC if ball 1 and ball 2 are colliding
    *
    *********************************************************/


   always @(posedge clk) begin

      if (reset) begin
   state <= GET_BALL_1;
   next_ball_one_index <= 0;
   next_ball_two_index <= 1;
   num_active_balls = 4'b1010;
   bram_addra <= 0;
   ball_one_count <= 0;
   ball_two_count <= 0;
   write_lhc_count <= 0;

      end



      if (state == COLLISION_DETECT) begin

   /* Collision with other ball */
   if (collide_lhc) begin
      bram_addra <= next_ball_one_index - 1;
      //bram_wea <= 1;
      {ballx_in , bally_in , ballz_in , ballvx_in , ballvy_in , ballvz_in , ball_color_in} <=
           {newx_1 , newy_1 , newz_1 , newvx_1 , newvy_1 , newvz_1 , new_color_1 };
      state <= COLLIDE_LHC;
   end

   /* Collision with wall */
   else if (collide_map != 4'b0000) begin
      bram_addra <= next_ball_one_index - 1;
      //bram_wea <= 1;
      {ballx_in , bally_in , ballz_in , ballvx_in , ballvy_in , ballvz_in , ball_color_in} <=
           {newx_1 , newy_1 , newz_1 , newvx_1 , newvy_1 , newvz_1 , new_color_1 };
      state <= COLLIDE_WALL;
   end

   /* No collisions and finished with ball 2 */
   else if (ball_2_done) begin
```

```
      /* No collisions and done testing all balls */
      if (next_ball_one_index == num_active_balls) begin
         next_ball_two_index <= 1;
         next_ball_one_index <= 0;
         bram_addr <= 0;
         state <= UPDATE_POSITIONS;
      end

      /* No collisions but more pairs to test */
      else begin
         next_ball_two_index <= next_ball_one_index + 1;
         bram_addra <= next_ball_one_index;
         state <= GET_BALL_1;
      end
   end

   /* Not done with ball 2 */
   else begin
      bram_addra <= next_ball_two_index;
      state <= GET_BALL_2;
   end
      end // if (state == COLLISION_DETECT)



      if (state == COLLIDE_WALL) begin
   bram_wea <= 0;

   if (next_ball_one_index == num_active_balls) begin
      next_ball_two_index <= 1;
      next_ball_one_index <= 0;
      bram_addr <= 0;
      state <= UPDATE_POSITIONS;
   end
   else begin
      next_ball_two_index <= next_ball_one_index + 1;
      bram_addra <= next_ball_one_index;
      state <= GET_BALL_1;
   end // else: !if(next_ball_one_index == num_active_balls)

      end // if (state == COLLIDE_WALL)



      if (state == COLLIDE_LHC) begin
   case(write_lhc_count) begin

      /* Write begins for first ball */
      2'b00 : begin
    state <= COLLIDE_LHC;
      write_lhc_count <= 3'b001;
    end

      /* Write ends for first ball. Write begins for second ball */
      2'b01 : begin
    state <= COLLIDE_LHC;
      bram_addra <= next_ball_two_index + 1;
      {ballx_in, bally_in, ballz_in, ballvx_in, ballvy_in, ballvz_in, ball_color_in} <=
            {newx_2, newy_2, newz_2, newvx_2, newvy_2, newvz_2, new_color_2};
      write_lhc_count <= 3'b010;
    end
   /* Write finishes for second ball */
   2'b10 : begin
 state <= COLLIDE_LHC;
   write_lhc_count <= 3'b011;
 end

   /* Setup for next state of reads */
   2'b11 : begin
 state <= GET_BALL_1;
   bram_addra <= next_ball_one_index;
   next_ball_two_index <= next_ball_one_index + 1;
   //bram_wea <= 0;
   end

endcase // case (write_lhc_count)

   end


   if (state == GET_BALL_1) begin
if (ball_one_count == 0) begin
   ball_one_count <= 1;
end
else begin

   /* Grab the values for ball 1 */
   ballx_1 <= 3'b000;
   bally_1 <= 3'b001;
   ballz_1 <= 3'b010;
   next_ball_one_index <= next_ball_one_index + 1;

   /* Set dummy  values and continue */
   if (next_ball_one_index == num_active_balls - 1) begin
      ballx_2 <= 3'b011;
      bally_2 <= 3'b100;
      ballz_2 <= 3'b101;
      state <= COLLISION_DETECT;
   end

   /* Else, just increment the index and move to GET_BALL_2 */
   else begin
      ball_one_count <= 0;
      bram_addra <= next_ball_two_index;
      state <= GET_BALL_2;
   end

end // else: !if(ball_one_count == 0)
   end // if (state == GET_BALL_1)



   /* Grab the values from the mem module and store */
   if (state == GET_BALL_2) begin
if (ball_two_count == 0) begin
   ball_two_count <= 1;
end
else begin
   next_ball_two_index <= next_ball_two_index + 1;

   /* Grab values from memory */
   ballx_2 <= 3'b111;
   bally_2 <= 3'b111;
   ballz_2 <= 3'b111;
```

```
        state <= COLLISION_DETECT;
        ball_two_count <= 0;
    end
        end // if (state == GET_BALL_2)


        /* Steps for updating the positions
         * Set the new positions based off of the current balls positions and
             velocities
         * Set write enable to high, go to the write stage
         */
        if (state == UPDATE_POSITIONS) begin
    /* Update all of the values */
    ballx_in <= ballx_out + ballvx_out;
    bally_in <= bally_out + ballvy_out;
    ballz_in <= ballz_out + ballvz_out - GRAVITY;
    ballvx_in <= ballvx_out;
    ballvy_in <= ballvy_out;
    ballvz_in <= ballvz_out - GRAVITY;
    state <= WRITE_POSITIONS;
    //bram_wea <= 1;
        end


        /* Steps for the write stage
         * Write enable is high, wait one cycle for write to finish.
         * On next clock cycle, the write is finished, set we to 0 and change
             address
         * Go back to the UPDATE POSITIONS state on second clock cycle
         */
        if (state == WRITE_POSITIONS) begin
    case(position_write_count) begin
        1'b0 : begin
      position_write_count <= 1;
        state <= WRITE_POSITIONS;
        end

        1'b1 : begin
          //bram_wea <= 0;
        if (ball_addra == num_active_balls - 1) begin
            state <= DONE;
            ball_addra <= 0;
        end
        else begin
            ball_addra <= ball_addra + 1;
            state <= UPDATE_POSITIONS;
        end
     end
    endcase // case (position_write_count)
        end // if (state == WRITE_POSITIONS)


        if (state == DONE) begin
    bram_addr <= 0;
    next_ball_one_index <= 0;
    next_ball_two_index <= 1;
    state <= (enable) ? GET_BALL_1 : DONE;
        end
    end // always @ (posedge clk)
endmodule // collision_detection_state

/*
* This is the modelsim test used to test the Physics Engine State machine.
* It initializes 2 balls in our model bram and run the physicds simulation.
*/
`timescale 1ns / 1ps
module collision_test_v;

  // Inputs
  reg clk;
  reg reset;
  reg enable;
  reg init1;
  reg init2;
  reg init3;
  reg done_init;

  // Instantiate the Unit Under Test (UUT)
  collision_detection_state_test_final uut (
    .clk(clk),
    .reset(reset),
    .enable(enable),
    .init1(init1),
    .init2(init2),
    .init3(init3),
    .done_init(done_init)
  );

  initial clk <= 0;
  always #2 clk <= !clk; // 2ns half-period = 500MHZ

  initial begin
    // Initialize Inputs
    reset = 0;
    enable = 0;
    init1 = 0;
    init2 = 0;
    init3 = 0;
    done_init = 0;
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    reset = 1;
    @(posedge clk);
    @(posedge clk);
    reset = 0;
    @(posedge clk);
    @(posedge clk);
    init1 = 1;
    @(posedge clk);
    @(posedge clk);
    init1 = 0;
    init2 = 1;
    @(posedge clk);
    @(posedge clk);
    init2 = 0;
    done_init = 1;
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
```

```
@(posedge clk);
    enable <= 1;
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
enable <= 0;

#140
enable <= 1;
@(posedge clk);
@(posedge clk);
```

```
    enable <= 0;

    #140
    enable <= 1;
    @(posedge clk);
    @(posedge clk);
    enable <= 0;

    #140
    enable <= 1;
    @(posedge clk);
    @(posedge clk);
    enable <= 0;

    #140
    enable <= 1;
    @(posedge clk);
    @(posedge clk);
    enable <= 0;

    #140
    enable <= 1;
    @(posedge clk);
    @(posedge clk);
    enable <= 0;

    #140
    enable <= 1;
    @(posedge clk);
    @(posedge clk);
    enable <= 0;

    #140
    enable <= 1;
    @(posedge clk);
    @(posedge clk);
    enable <= 0;

    #140
    enable <= 1;
    @(posedge clk);
    @(posedge clk);
    enable <= 0;

    #140
    enable <= 1;
    @(posedge clk);
    @(posedge clk);
    enable <= 0;


    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here

  end

endmodule
```

7 Appendix C: Graphics Verilog Code

```
`timescale 1ns / 1ps
//
   ////////////////////////////////////////////////////////////////////////////////
// Raytracer - SRAM Controller
// Adam Lerer / Sam Gross
//
   ////////////////////////////////////////////////////////////////////////////////
module sram_controller2(clk, reset, flip, write_x, write_y, write_color,
    write_depth,
    read_x, read_y, read_color, read_depth,
ram0_data, ram0_address, ram0_we_b,
ram1_data, ram1_address, ram1_we_b); //RAM IO

  parameter x_bits = 9;
  parameter y_bits = 10;
  parameter color_bits = 24;
   parameter depth_bits = 8;
  parameter width = 640;
  parameter height = 480;


  input reset, clk, flip;
  input [x_bits-1:0] write_x;
  input [y_bits-1:0] write_y;
  input [color_bits-1:0] write_color;
   input [depth_bits-1:0] write_depth;
  input [x_bits-1:0] read_x;
  input [y_bits-1:0] read_y;
  output reg [color_bits-1:0] read_color;
  output reg [depth_bits-1:0] read_depth;

  inout [35:0] ram0_data, ram1_data;
  output [18:0] ram0_address, ram1_address;
  output ram0_we_b, ram1_we_b;
  wire [18:0] read_address;
  reg [18:0] write_address;
  reg [23:0] write_data_int, write_data_int2, write_data;
  reg read_buffer;

  assign ram0_data = read_buffer ? {4'h0,write_depth, write_data} : 36'hz;
  assign ram1_data = ~read_buffer ? {4'h0,write_depth, write_data} : 36'hz;
  assign ram0_address = read_buffer ? write_address : read_address;
  assign ram1_address = ~read_buffer ? write_address : read_address;
  assign ram0_we_b = ~read_buffer ? 1'b1 : 1'b0;
  assign ram1_we_b = read_buffer ? 1'b1 : 1'b0;
  assign read_address = (read_y << x_bits) + read_x;

  always @ (posedge clk)
  begin
    if (reset)
      read_buffer <= 1'b0;
      write_data_int <= write_color;
      write_data_int2 <= write_data_int;
      write_data <= write_data_int2;
    if (flip)
      read_buffer <= ~read_buffer;
      write_address <= (write_y << x_bits) + write_x;
      {read_depth, read_color} <= read_buffer ? ram1_data[depth_bits+color_bits
          -1:0] : ram0_data[depth_bits+color_bits-1:0];
    end
```

```
endmodule

‘timescale 1ns / 1ps
//
   ////////////////////////////////////////////////////////////////////////////////

// Company:
// Engineer:
//
// Create Date:    21:56:29 12/07/2009
// Design Name:
// Module Name:    displayFSM
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
   ////////////////////////////////////////////////////////////////////////////////

module displayFSM(input clock,  reset,ballValid,floorValid,newFrame, input [9:0]
    ballx, input [8:0] bally,
    input [23:0] ballColor,input [7:0] ballDepth, input [9:0] floorx, input [8:0]
         floory, output reg floorAccepted,
    ballAccepted,output reg [9:0] writex, output reg [8:0] writey, output reg
        [23:0] pixelColor);


   reg [9:0]          x;
   reg [8:0]          y;
   reg [24:0]            color;
   reg [7:0]          depth;
   reg [1:0]          state;
   reg [9:0]          xaddress_counter;
   reg [8:0]          yaddress_counter;
   wire [7:0]            zbuffer_data;
   wire [18:0]           zbuffer_addr;
   reg             zbuffer_we;
   wire [7:0]            zbuffer_write;

   zbuffer zbuffer1(.clock(clock), .data(zbuffer_data),.addr(zbuffer_addr),.we(
       zbuffer_we),
         .write(zbuffer_write));

   parameter          floorColor = 0;
   parameter          floorDepth = 8'b1111_1110;
   parameter          cleaningBuffer = 0;
   parameter          waitForInput = 1;
   parameter          writing = 2;
   assign          zbuffer_addr = {x,y};
   assign          zbuffer_write = depth;



   always @ (posedge clock) begin
      case(state)
  cleaningBuffer: begin
```

```
    pixelColor <= 0;
    zbuffer_we <= 1;
    depth <= 8'b1111_1111;
    writex <= xaddress_counter;
    writey <= yaddress_counter;
    xaddress_counter <= xaddress_counter + 1;
    if(xaddress_counter == 639)begin
       yaddress_counter<=yaddress_counter + 1;
       if(yaddress_counter == 479)
   state<=waitForInput;
    end
end
waitForInput: begin
    zbuffer_we <=0;
    if(newFrame)begin
       xaddress_counter <=0;
       yaddress_counter <=0;
       state<=cleaningBuffer;
    end
    else begin
       if(floorValid)begin
    state <= writing;
    floorAccepted <= 1;
    x <= floorx;
    y <= floory;
    color <= floorColor;
    depth <= floorDepth;
       end
       else if(ballValid)begin
    state <= writing;
    ballAccepted <=1;
    x <= ballx;
    y <= bally;
    color <= ballColor;
    depth <= ballDepth;
       end
    end
end
//Ball States
writing: begin
    floorAccepted <= 0;
    ballAccepted <= 0;
    if(depth<zbuffer_data)begin
       writex <= x;
       writey <= y;
       pixelColor <= color;
       zbuffer_we <=1;
    end
    if(newFrame)
      state <= cleaningBuffer;
    else
      state <= waitForInput;
end
//        ballWrite: begin
//           state<=waitForInput;
//           if(posBall ==0)begin
//              if(read_data[15:8]> dataBall[15:8])begin
//                 we<=1;
//                 addr<= addrBall;
//                 pixelData <= {read_data[35:16], dataBall};
//              end
//           end
//           else begin
```

```
  //            if(read_data[31:24]> dataBall[15:8]) begin
  //              we<=1;
  //              addr<= addrBall;
  //              pixelData <= {4'b0000, dataBall, read_data[15:0]};
  //            end
  //          end
  //        end
  //        //Floor States
  //        floorInput: begin
  //          floorAccepted <= 0;
  //          writex <=
  //          if(newFrame)
  //            state <= cleaningBuffer;
  //          else
  //            state <= waitForInput;
  //        end
  //        floorWait: state <= floorWrite;
  //        floorWrite: begin
  //          state<=waitForInput;
  //          if(posFloor ==0)begin
  //            if(read_data[15:8]> dataFloor[15:8]) begin
  //              we<=1;
  //              addr<= addrFloor;
  //              pixelData <= {read_data[35:16], dataFloor};
  //            end
  //          end
  //          else begin
  //            if(read_data[31:24]> dataFloor[15:8]) begin
  //              we<=1;
  //              addr<= addrFloor;
  //              pixelData <= {4'b0000, dataFloor, read_data[15:0]};
  //            end
  //          end
  //        end
  //        default: state <= waitForInput;
      endcase
    end
endmodule

`timescale 1ns / 1ps
//
  ////////////////////////////////////////////////////////////////////////////////

// Company:
// Engineer:
//
// Create Date:    15:45:47 12/05/2009
// Design Name:
// Module Name:    projection
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
  ////////////////////////////////////////////////////////////////////////////////
```

```
module projection(clk_27mhz, addr, dataIn, xpos, ypos,color,ready,newFrame,done,
    distance);
  input clk_27mhz;
  output reg [8:0] addr;
  input [127:0] dataIn;
  output [9:0] xpos;
  output [9:0] ypos;
  output [2:0] color;
  input ready;
  input newFrame;
  output reg done;
  output [8:0]distance;

   parameter       maxMemAddr = 9'b1_1111_1111;
   parameter       signed camerax=0;
   parameter       signed cameray= 18'b11_1111_1111_1111_0000;
   parameter       signed cameraz = 0;
   parameter       signed ey = 18'b11_1111_1111_1111_1111;


   wire      signed [17:0] xloc, yloc, zloc,xd,yd,zd;
   wire [18:0]      xshifted,yshifted,zshifted;

   divider divider1(clock,ey,dy,ans);


   assign      color = dataIn[18:16];
   assign      xloc = dataIn[127:110];
   assign      yloc = dataIn[109:92];
   assign      zloc = dataIn[91:74];

   assign      xd = xloc-camerax;
   assign      yd = yloc - cameray;
   assign      zd = zloc-cameraz;


   assign       xshifted = xloc + 131072;
   assign       zshifted = zloc + 131072;
   assign       yshifted = yloc + 131072;

   assign       xpos =dx * ans + 620;
   assign       ypos = dz * ans + 240;
   assign       distance = yshifted[18:11];

  always @(posedge clk_27mhz) begin
    if(newFrame)begin
      addr <=0;
      done <=0;
    end
    if(ready && (addr<maxMemAddr)&&~done)begin
      addr <= addr+1;
    end
    if((addr==maxMemAddr) && ready)begin
      done <=1;
      addr <=0;
    end
  end


endmodule

`timescale 1ns / 1ps
//
```

```
    ////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:17:59 12/05/2009
// Design Name:
// Module Name:    shading
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
    ////////////////////////////////////////////////////////////////////////////////
module shading(clock,reset,ready,done,pixelValid,distance,colorOut,colorIn,xpos,
    ypos,pixelx,pixely,pixelDepth,accepted);
  input clock, done, accepted, reset;
  input [2:0]colorIn;
  input [9:0] xpos;
  input [8:0] ypos;
  input [7:0] distance;
  output reg [9:0]pixelx;
  output reg [8:0]pixely;
  output reg [7:0]pixelDepth;
  output [23:0] colorOut;
  output reg pixelValid;
  output ready;

  parameter radius = 16;
  reg [9:0] xcounter;
  reg [8:0] ycounter;
  reg [2:0] state;
  reg [2:0] color;
  reg [9:0] xcenter, xLow, xHigh;
  reg [8:0] ycenter, yLow, yHigh;
  //wire [8:0] radius;
  wire [20:0] xsquared,ysquared,rsquared;
  wire signed [9:0] x;
  wire signed [8:0] y;
  //assign radius = ~distance;
  assign x = xcenter-xcounter;
  assign y = ycenter-ycounter;


  assign colorOut = (color==0)? {24'h002EB8}:
             (color==1)? {24'h3366FF}:
             (color==2)? {24'h33CCFF}:
             (color==3)? {24'h33FF66}:
             (color==4)? {24'hFFCC33}:
             (color==5)? {24'hFF6633}:
             (color==6)? {24'hFF3366}:
             {24'hFF33CC};


  parameter initialize = 0;
  parameter calculateValid = 1;
  parameter increment = 3;
  parameter waitForRam=4;
  parameter multiply = 5;

  assign ready = (state==initialize);
  assign xsquared = x*x;
  assign ysquared = y*y;
  assign rsquared = radius*radius;

  always @(posedge clock) begin
    if(reset)
      state<=initialize;
    case(state)
      initialize: begin
        color <= colorIn;
        pixelDepth <= distance;
        xcenter <= xpos;
        ycenter <= ypos;
        xLow <= xpos-radius;
        xHigh <= xpos+radius;
        yLow <= ypos-radius;
        yHigh <= ypos + radius;
        xcounter<=xpos-radius;
        ycounter<=ypos-radius;
        state <= done ? initialize:calculateValid;
      end
      calculateValid: begin
        if(xsquared+ysquared<rsquared)begin
          pixelx <= xcounter;
          pixely <= ycounter;
          pixelValid<=1;
          state <= waitForRam;
        end
        else
          state<=increment;
      end
      waitForRam: begin
        state<=accepted?increment:waitForRam;
        pixelValid <= accepted?0:1;
        end
      increment:begin
        pixelValid <=0;
        if(xcounter<xHigh)
          xcounter<=xcounter+1;
        else begin
          xcounter<=xLow;
          ycounter<=ycounter+1;
        end
        if(ycounter==yHigh)
          state<=initialize;
        else
          state<=calculateValid;
      end
    endcase

  end

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// 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, fb_clk, lock1, lock2, dcm_reset,ram_clock;

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

  assign ref_clk = ref_clock;  // used to fix BUFG constraint

   //IBUFG ref_buf (.O(ref_clk), .I(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
```