Serpent

Holli Rachall

December 13, 2006

6.111 Fall 2006

Abstract

Serpent is a classic Snake type arcade game, wherein the user controls a serpentine avatar which maneuvers the screen trying to collect apples, while avoiding walls and its own tail. Eating these apples causes the serpent's tail to grow longer, thus increasing the difficulty of the game as the tail becomes harder to avoid. The system for this project consists mainly of a snake architecture, game logic, a graphic system, and a system for user input. These modules, along with a few others, were designed in Verilog and programmed to the 6.111 labkit in order to create a graphically smooth snake game which the user can play using an attached Nintendo Entertainment System controller.

1. Overview

Serpent is an arcade-type game wherein the user controls a snake which he directs around the screen using the directional pad on a Nintendo Entertainment System (NES) controller connected to the labkit. The snake must avoid collisions with walls, the edges of the screen, and its own tail as it moves about the screen, all while trying to collect the apples which appear. Each time the snake eats an apple, it grows by a segment, making it more difficult to avoid its own tail. Collisions with obstacles cause the snake to lose a life and restart its current level. The game has five levels, and the goal of each level is to reach a maximum length, at which point the user is brought to the next level. The final level has no maximum length, and thus continues until the user runs out of lives and reaches the ending screen. Pause can be toggled on and off by pressing either the start or select button, and one of these must be pressed to start each level, which begins in a paused state in order to prevent the user from being surprised by suddenly having to deal with the movement of the snake to its starting position. To restart the game, either after losing or at any time, start and select must be pushed at the same time. Also, the serpent's speed can be controlled with switches 2-0 on the labkit: the slowest speed corresponds to all switches down and the fastest corresponds to all switches up (although this speed is probably much too fast to be played by a human).

The implementation of Serpent uses 6 modules: a game logic module, a graphics module, a snake module, a randomizer, a timer, and a NES controller handler. The game logic module detects collisions between the snake and walls and apples, communicates between the snake module and the graphics module, handles the signals to generate new apples, deals with the number of lives remaining, and handles level and level transition information. The graphics module generates a signal which indicates when the current frame has been drawn and uses input from the game logic module to create a smooth

graphical representation of the snake and the current state of the game. The snake module holds a representation of the current state of the snake, deals with signals to move or grow received from the game logic module, and determines collisions between the serpent and the edges of the screen and between the serpent and itself. The randomizer module generates random locations which the game logic can use to place new apples. The timer module uses the new frame signals from the graphics module to alert the game logic as to when the snake needs to move. The NES controller handler interfaces with an NES controller to output useful signals corresponding to current button state. The game logic and snake modules operate under the assumption that the screen is divided into 32 by 32 pixel tiles, such that the screen is 32 tiles wide and 23 tiles tall. The graphics module then uses this tile data to generate smooth movement. These modules are connected in a main module in such a way that they combine to form a functioning game.

2. Modules

2.1 Timer

The timer module is a counter which sends to the game logic a signal whenever the serpent should move forward by a tile. It takes as its inputs a signal from the graphics module representing that a frame has been drawn, a signal from the game logic representing that a level should be started, a requested speed for the snake (from the state of switches 2-0), and the global clock and reset signals. It uses a simple counter which counts once per clock cycle until it reaches the desired period length then sets the output signal, tick, high for one clock period. The period is determined either at reset, level start, or the tick signal by the state of the speed switches. This ensures that the period changes just as the serpent moves from one tile into the next, thus keeping the serpent's movements and the graphical

system's interpretations of those movements synchronized. The tick signal from the timer goes to the game logic module which uses it to determine when to tell the serpent to move or grow.

2.2 NES Controller Handler

The NES controller handler module takes input from an NES controller and interprets it to output the current state of the buttons on the controller. The NES controller has 5 wires: power, ground, pulse, latch, and signal. Power and ground connect to power and ground signals. The chip in the NES controller operates much more slowly than the serpent logic, and so the clock needed to be slowed down in order to send appropriately timed signals to the controller. The controller handler module uses a counter to generate what is essentially a 1 mHz clock, which it sends to the controller through the pulse wire. The handler also generates a high signal along the latch line once every 9 pulse cycles. This latch signal tells the NES controller to save the state of the buttons, which it then sends to the handler on the signal line one button at a time over the next 8 pulse cycles in the order A, B, Select, Start, up, down, left, and then right, with each button state being active low. Each of these button states is put into a register based on a count which keeps track of how many pulse cycles have passed since the last latch signal. These registers are output to the main module, where the state of the directional buttons are sent to the snake module, the states of start and select (which are the reset buttons) are sent to the game logic module, and the state where both start and select are pressed is translated into the global user reset signal.

2.3 Randomizer

The randomizer module uses a pseudorandom number generator to generate a number between 0 (inclusive) and the parameter MAX (exclusive). For the purpose of the serpent game, MAX=736, and thus the randomizer chooses one of the 736 tiles on the board as the new location for the snake. The randomizer module uses a 16 bit linear feedback register to generate its random locations. This works by starting with a 16 bit seed (which can be anything but zero) and then appending the XOR of bits 5, 3, 2, and 0 to the beginning of the first 15 bits of the seed. If this was the entirety of the random number generator, it would be guaranteed to cycle through every number from 1 to 2^{16}-1 once each in a seemingly random order and then repeat the cycle. The randomizer also, however, incorporates a counter which counts the number of clock pulses since the last reset (mod 2^{16}), and then adds this number to the current random number before performing the XOR and linear shift. To prevent problems caused by the rare case when this total might be zero (in which case the linear feedback register on its own would just continue to compute the number zero), the total of the previous number and the counter is checked, and if it sums to zero, then the linear feedback shift is performed only on the previous number, without the addition of the counter. When the randomizer receives a signal from the game logic telling it to compute a new number, it computes a new number then checks to see if the lower 10 bits of that number is less than MAX. If so, those lower 10 bits are output to the game logic, which will interpret them as the new apple location, and the signal done goes high. Otherwise, the signal done goes low, and new totals continue to be calculated until one is found in the correct range, at which point done is finally signaled. The use of a 16 bit linear feedback shift register to calculate the needed 10 digit output means that, even without the addition of the time based system, the pattern produced by the randomizer would require 2^{16} outputs before repeating, and that the current 10 bit output would not be enough to determine conclusively the next 10 bit output. The combination of the use of a larger shift register and time based additions makes the output random enough that no two games should have the exact same pattern of apples.

2.4 Snake

The snake module controls the representation and movement of the snake. The snake is stored as a sort of 2-dimensional linked list in a 32 by 23 array which represents the tiles of the gameplay area. This array is stored as a 1024 by 14 bit BRAM, where the first and last 5 bits of the BRAM address correspond to the x-location and y-location respectively of a tile. Each location in the BRAM holds a value which represents the information about the portion of the snake which is in that tile. The first 4 bits of this value represent the orientation of the snake segment, with 1111 representing a tile with no snake segment occupying it. The orientations represented by other values are summarized in Table 1. The next 10 bits of the value represent the location in the BRAM of the next segment of the snake (going in a head-to-tail direction). The tail, which has no next segment, has its last 10 bits set to all 1's. The snake module also contains the snake's length and a pointer to the head and a pointer to the tail location of the snake. These are useful for updating the snake's position. On reset, the head and tail location pointers are set to their default values, and a counter runs through each location in the BRAM, setting it to its appropriate initial value for the serpent's starting location at the bottom left of the screen.

Table 1: Numerical Representations of Orientations

Number	**Orientation**
0000	up
0001	right
0010	down
0011	left
0100	up-right
0101	up-left
0110	right-up

0111	right-down
1000	down-right
1001	down-left
1010	left-up
1011	left-down

If the serpent is not in the process of moving or growing, then the snake module returns the orientation and type of the tile at the location of the input location. The orientation is determined from the contents of the BRAM at that location. The type is returned as 00 if the location contains the snake's head, which is determined by checking the equality of location and the head location register. In a similar manner, 11 is returned as the type if the location contains the tail. Otherwise, either even or odd is returned, based on the current parity of the game and the location of the tile. The parity is just a 1 bit value which switches on and off each time the serpent moves. By XORing the final bit of the x-location, the final bit of the y-location, and the parity bit, a checkerboard pattern emerges whereby every other tile is odd, and the rest are even. When the parity switches after the serpent moves, this causes the tile type to switch from even to odd or vice versa, thus giving the appearance that each individual segment, whether even or odd, is moving forward along with the snake.

The snake module also deals with directional button state inputs from the NES controller handler. If the game is paused, these buttons have no effect. Otherwise, the snake module keeps track of the most recent direction pressed, unless that direction is opposite the snake's current direction of movement, at which point the module ignores it. This is so the user will not accidentally try to turn in the direction opposite the serpent's movement, thus causing it to immediately run into itself and lose a life. The default value if no buttons are pushed is just the serpent's current direction, and so on its own the serpent will move in a straight line in whatever direction it is pointing.

The serpent responds to a high signal on its move input by starting a process which begins by checking to see if moving in the direction last pressed will cause it to move beyond the edge of the screen. If so, it makes no more effort to move and outputs a high value on its collision output. If not, then the serpent begins to look for its tail so that it can move. Starting at the location specified by the head location register, the snake gets the value of the location of the next segment, checks if this location is the tail location, and if not, moves to that next segment and continues the process of following the snake. Once it reaches the segment whose next segment pointer points to the tail, it makes this segment the new tail by changing its next segment pointer to all 1s and changing the tail location pointer to be the location of this segment. It also continues to the previous tail segment and writes over its orientation with all 1s so that it is replaced by a tile without a snake. Once the tail has been moved forward and the old tail removed, the old head location is given a new orientation based on the previous and new directions. Next, the location where the new head location is going to be is checked to determine if it is occupied, if it is, the collision signal goes high. Whether or not this collision occurs, the head is moved to its new location by moving in the appropriate new direction and writing its location in the BRAM to have the new direction as its orientation and the old head location as its next segment location. At this point, the parity bit is switched. Finally, the new head location is saved in the head location register, and the move is complete.

The serpent responds to a high signal on its grow input in nearly the same way, except that when growing, the serpent moves its head forward and leaves its tail in place, thus increasing in length by 1. In this case, the length register has 1 added to it and the serpent skips over the tail moving portion of its behavior and goes to the portion where it moves its head.

The head location, length, collision signal, and a busy signal which is high when the serpent is moving or growing are all output to be used by the game logic.

2.5 Game Logic

The game logic module detects collisions between the snake and walls and apples, communicates between the snake module and the graphics module, handles the signals to generate new apples, deals with the number of lives remaining, and handles level and level transition information.

During normal operation, when the serpent is not moving, the game logic receives a location from the graphics module and then outputs the type and orientation of the tile at that location. It first asks the snake module if the location is occupied by a snake, and returns the appropriate tile type if it is, replacing the snake's head with an explosion if a collision has occurred. If there is no snake segment at the tile, the game logic checks if there is a wall occupying the tile on the current level, and if so returns the wall tile type. Otherwise, the game logic module checks if there is currently an apple showing and if its location is the same as the location of the tile. If so, it outputs the apple tile type, and if not it outputs the empty tile type. The orientation is set to be the orientation of the snake segment at location if there is one and up, which is the default orientation, otherwise.

When the game logic receives a tick from the timer, it delays for 1024 clock cycles, long enough for the current state to be saved within the graphics, then communicates that tick to the snake as a grow signal if an apple was eaten on the previous turn and a move signal otherwise. After the serpent moves or grows, the game logic checks for collisions. If the serpent reports a collision or the new head location is occupied by a wall in the current level, the game logic checks to see if there are lives remaining. If there are not, then the game goes to the ending screen and remains stopped until it is reset. If there are lives remaining, the game pauses while displaying the collision, and the game logic

goes to a state where it waits for pause to be pressed and then sends out a new level signal to the other modules such that the current level is restarted with one less life. If there were no collisions reported by the serpent and no collisions with walls are detected, the game logic checks for collisions with apples. If an apple is collided with, the game logic remembers that the serpent should grow on the next tick, and it also removes the apple from the screen. It then checks if the serpent is long enough to progress to the next level (unless is is on the NUM_LEVELS-1 level which continues until the user runs out of lives), sending out the new level signal, pausing, and increasing the level by one if it is, and returning to normal operation otherwise. If the game logic detects no collisions with either obstacles or apples, it checks to see if there is currently an apple present, and if there is not, it generates one. Otherwise, it goes back to normal operation.

To generate a new apple, which is done one tick after the apple signal goes low, the game logic sends a new apple signal to the randomizer and then waits for the done signal. Once the randomizer is done, the game logic checks the new location for the presence of a wall or a snake segment. If the new apple would overlap with one of these, then the game logic signals to the randomizer to generate another location. This continues until the location generated is empty, at which point the apple signal goes high to indicate the apple's presence.

Each of the 5 levels is stored in a 1024 by 1 bit BRAM, where each address corresponds to a tile with the first and last 5 bits of the address corresponding to the x-location and y-location respectively of the tile and a 1 represents the presence of a wall and a 0 the absence of a wall. For a given address which is being checked, that address is looked at for each of the 5 BRAMs, then the results are stored in a 5 bit register which is indexed according to level, so the level 0 wall state is saved in the register wall[0], and so on. This makes adding additional levels incredibly simple, as it requires only creating a new BRAM representing the new level, increasing the size of the wall register, and increasing the maximum

level, the second and third of which can be done by merely changing the NUM_LEVELS parameter.

2.6 Graphics

The graphics module conveys information about the game state through a smoothly moving graphical interface and also signals to the timer when a frame has finished being drawn. The graphics module uses the XVGA module from Project 5 to generate the appropriate signals for hcount, vcount, etc. so that the graphics module can output the correct signals to the VGA monitor. It also uses an image processor which transforms the images of various tile types into their correct orientations. The graphics module is based on an 8 color system, with each of the 8 colors mapped to a 3 bit parameter representing it. Colors are generally present in their 3 bit representation until just before they are output to the VGA monitor, at which point they are translated into their 24 bit VGA color equivalents.

Each multicolored tile on the screen is represented by a 1024 by 3 bit BRAM, with the first and last 5 bits of the address corresponding to the x-location and y-location respectively of the pixel being represented. The 3 bit value at each of these addresses corresponds to one of eight colors that each pixel on the screen could possibly be: black, red, brick, green, lime, brown, orange, or yellow. By inputing a pixel location as an address to these BRAMs, it is easy to retrieve the color which should be associated with that pixel.

The bottom portion of the screen is dedicated to a graphical representation of the number of lives remaining as seen in Figure 1. When the vcount is in the appropriate region to draw this lives remaining bar, then hcount is looked at to determine which portion of the bar should be drawn. If hcount is at the far left (less than 212), then the pixel color to be output is determined by the contents of

a 212 by 32 bit BRAM which represents the drawing of the words “Lives Remaining:,” with a 1 representing a black letter and a 0 representing the brick background. Then there are places to display up to 3 additional lives. When hcount moves into each of these spaces, it checks the number of lives, and if it is greater than the current spot, it draws in that spot an egg from the BRAM which represents the picture of an egg, and otherwise it just outputs the brick color. For instance, in Figure 1, the number of lives must be 2 because the egg in spot 1 is showing and the egg in spot 2 is not. This means that the number of eggs displayed at the bottom of the screen shows how many more lives can be lost before the game is over.

Figure 1: Left Side of Lives Remaining Bar

The rest of the screen displays the progression of the game. To give the appearance of smooth movement even though the logic is actually tile based, the graphical system saves the previous state of the screen and then transitions from the previous state to the current state while the logic is in the current state. Each time the graphics module finishes drawing the last frame in the current game state, it saves the current tile type of all of the tiles into a 1024 by 3 bit BRAM. The graphics module determines when it has reached the final frame of the current game state with a counter which decrements a certain amount based on speed with each new frame. Just as the timer only reacts to speed changes upon ticks, the graphics module reacts to speed changes once its counter is equal to 0. This ensures that the graphics thinks the serpent is moving between tiles at the moment when it actually is, and thus the graphics and logic remain synchronized.

Each tile is drawn using its tile type and orientation as returned by the game logic, its saved previous tile type, and the current value of the counter. Because tiles are 32 pixels by 32 pixels, the x-location

and y-location within a tile are specified by the lower 5 bits of the hcount and vcount respectively. Also, the x-location and y-location of the tile in the array are specified by the upper 5 bits of hcount and vcount respectively. Thus the graphics asks the game logic about the tile with location specified by the concatenation of the upper 5 bits of hcount followed by the upper 5 bits of vcount, and then with the resulting tile type and orientation, it can use the lower 5 bits of hcount and vcount concatenated to determine which color should be present at the current pixel. To do this for the snake tiles, the graphics module first gives the current pixel location and orientation to the image processing module, which uses simple BRAMs which map output coordinates to input coordinates in order to transform the space of a straight, upward pointing image into a straight image pointing in any of 4 directions or one of 8 curved images (these 12 transformations correspond to the 12 possible orientations). The image processor also outputs a valid bit which represents whether the output pixel corresponds to an input pixel or should become background. This was necessary for the curves, which have a corner which does not correspond to any input pixel. Once the pixel location is input to the image processor along with the orientation, it returns the corresponding pixel location which should be used from the untransformed image. This corresponding pixel location is then input into each of the various snake tile type BRAMs, and the correct color value is then chosen from these outputs based on tile type. If the tile is not a snake type tile, the pixel location is input directly into the other non-snake tile type BRAMs, and the correct color value is then chosen from these outputs (or a simple solid color in the case of walls and empty tiles) based on tile type. Using the current tile type and validity, this outputs the correct tile image at each actual stage of snake movement.

In order to make the movement appear smooth, this method is changed slightly. For snake tiles, once the transformed pixel location is returned from the image processor, the value of the counter is subtracted from the y-coordinate, and then the x-coordinate and this shifted y-coordinate are input to the images of each if the . This is because the transformed pixel location assumes everything is

pointing, and thus moving, upwards. Thus by subtracting counter (which decreases from 32 to 0 as the serpent traverses a tile) from the y-coordinate, the image moves upwards as time progresses. To finish this illusion of movement, each tile, snake or not, has its previous tile type drawn if the y-coordinate (which defaults to the lower 5 bits of vcount for upright tiles) is less than the value of the counter, and its current value drawn otherwise. For instance, for tiles with an upward orientation, the previous type of the tile is drawn in the top half, and the current type is drawn in the bottom half. Because of the space transformations created by the image processor, this results in smooth movement even around corners. The many BRAMs used to implement this behavior cause this module to be heavily pipelined.

The intricacies of graphics made some changes necessary in order to get the best looking game. The method of smooth movement meant that both apples and collision explosions would appear as though being uncovered instead of all at once. This looked awkward, and so was fixed by excluding those 2 tile types from having their previous state drawn. This graphical method also encountered a strange problem due to the fact that non-snake tiles default to an upright orientation. Because the space occupied by the tail is generally occupied by empty space the next move, the place which was previously the tail nearly always has an upward orientation. This meant that the portion of the tail still in the previous tile was drawn pointing upwards instead of in its correct location. To fix this, the graphics module records the previous tail orientation as it records the previous types of all of the tiles. Then, once it reaches the tile whose previous tile type was tail, it substitutes the previous tail orientation for the actual orientation as input into the image processor. This makes the tail move correctly, but caused a problem in the unlikely case that the tile previously occupied by the tail becomes occupied by the head. The fix for the tail orientation would then cause the head to appear in the wrong orientation. This is fixed by substituting the previous tail orientation only if the previous tile is the tail and the current tile is not the head. In the case that the previous tile is the tail and the current tile is the head, the head is drawn in its correct orientation and the end tip of the tail is undrawn unless

it has the same orientation as the head, as the system architecture does not allow for multiple image transformations in the same tile, and this seemed the most elegant way of dealing with what was a small and rare problem without having to overhaul the graphical system.

These graphical methods produce smooth images which move across the screen as a real object would, with the even and odd stripes of the serpent and even its tail gliding smoothly around corners.

3. Testing and Debugging

While it was simple to test the randomizer module, the rest of the modules were not so easy. The randomizer I created and tested with ModelSim by asking it for new random locations and getting a steady stream of random seeming numbers under my maximum value of 736. After the randomizer, however, most of the modules required BRAMs, and thus could not be modeled. This meant that in order to test my snake and game logic modules, I had to create a rudimentary graphics module so that I could see what was happening. Once the game worked with the rudimentary graphics, I created the image processor and changed the graphics module to use it and output correct still images instead of just colored blocks representing the tile types. Once this was debugged, I changed the graphics module to get the smooth movement functionality. Finally, I added changeable speeds, opening and closing screens, and the NES controller, which brought many debugging problems with it.

The first big module I wrote and tested was the snake module. To test this I made a rudimentary graphics module which displayed red for tiles that were apples, brick for tiles that were walls, green for the tile containing the head, lime for tiles containing odd snake segments, orange for tiles containing even snake segments, brown for the tile containing the tail, yellow for tiles containing explosions, and

black for empty tiles. With this, I tested the snake module, which had many more problems than I had anticipated. After much debugging, throughout most of which part or all of my serpent would disappear when it moved, I found that my main problem was with accessing memory to seek out the tail. I had been waiting one cycle between segments, not realizing that it took one cycle to get the address of the next segment out, then another cycle to put that address into the memory, and only then would the address of the following segment be revealed.

Once my snake was finally cured, putting me almost a week behind schedule, I began to work on my game logic. This module came together surprisingly well. All was fine until I reached the second level and realized that walls were being drawn, but collisions with them were not recognized and the apples were sometimes being drawn under them. This turned out to be a pipelining problem, I think, and once fixed the game logic worked fairly well for the time being.

I started testing the still graphics without making the image processor, just to see if I could get the images to draw from BRAMs. I did, but at the time did not think of using a program to transcribe my graphics, so I transcribed my 32 pixel by 32 pixel graphics that I designed in Paint by hand to get the COE files for the BRAMs. In retrospect, I should have written a program to do this, as it might have saved me a lot of time. Still, once the BRAMs had been generated, the snake moved around the screen with a head that looked like a head, apples that looked like apples, and looking very silly going sideways with all of its segments pointing up. It was an exciting moment. It took a few more days to figure out how to implement the image processor. My original idea had been to give it all 1024 pixels of an image and a background color and have it return 1024 pixels of the image in its correct orientation. After some though, I decided that this plan did not seem very practical, especially when each pixel is represented by 3 bits of color. So I thought and came up with the implementation I used, sort of the reverse of my previous idea, with the valid bit output replacing the background color input,

which was the most difficult part for me to figure out how to replace. Once again, this part was debugged visually, making sure the graphical responses of the snake on the screen were what they should have been. I spent quite a while trying to debug graphical glitches on this part, and learned that sometimes, when you've messed with something too much, it's best to take out all the fixes and start over at a point that's less complicated. Eventually, after taking out much of my previous pipelining and starting over, I got it pipelined correctly.

Perhaps the longest part of the project was getting the snake to move smoothly. Because the snake was colored so as to make the segments blend into one another, I could tell that errors were occurring, but I had a very hard time identifying them. To fix this, I replaced the even and odd segments of my snake with orange and lime boxes, as this made it much easier to determine what was happening with each segment. It then became very clear that the glitch was being caused by only the current and not the previous segments being drawn. This made me curious to see if the previous segments were saving correctly. They were not. I had managed to save the state after the move, and thus I was getting 2 sliding copies of the same image. I fixed the timing between modules to prevent this error, along with several other errors which were not problems when using discrete graphics, but which the smooth graphics had revealed. My final two problems were the tail always moving upwards, which I eventually realized was because it had no orientation, and a segment in the upper left corner where the saving of previous tiles was glitching. The tail moving upwards was the most difficult to solve, primarily because at first I implemented a half-correct solution which only made things worse and caused them to look more confusing. Instead of fixing the orientation of the tile that had been the tail, I only fixed part of its orientation, and in a weird way that caused parts of it to still go upwards, but this time in uneven stripes that made no sense. I eventually realized that I had put in a constraint I shouldn't have, and thus the orientation was being changed based on the output of the image processor rather than just on the previous tile type.

Efforts to create an opening screen failed at first because the file I created was too large to be implemented as a BRAM, even with only 3 bits per color. Instead of giving me an error, however, Xilinx would just run for a few hours and stop with no output when I tried to generate it. When I finally calculated the size of the BRAM it was trying to generate, I realized the problem – it would have taken up all of the BRAM in the labkit, and some of it was already being used by my other images. So, I created a new image at half the resolution, and a closing screen two, and I managed to generate them without many problems.

Adding changeable speeds caused problems at first when they were not synchronized with the movement of the serpent from one tile to the next. I first fixed this by making the speed changeable only on reset, but that seemed a little annoying so I found a way to implement changing speeds once the end of the current tile is reached.

Finally, I had a lot of problems attaching and debugging the NES controller. I hooked the results of my controller handler to the LEDs on the labkit to see if the correct buttons were registering. I kept getting outputs from its signal line that made no sense, I now know because I was clocking it at far too high a speed. Once I finally downshifted my clock enough, many of the problems with the controller went away. Then, however, when I connected it to my Serpent game, things got really bad. Nothing would register or show up on the screen at all. It turns out it was a problem with not initializing some states for the controller such that the reset signal never settled and thus the logic got stuck. Once I fixed that, even though the LEDs seemed to be lighting up correctly, the controller was not guiding the snake correctly at all. This turned out to be because I had assumed the buttons were active high, instead of active low as they actually are. Fixing this and debouncing the buttons fixed all controller problems except one – after a reset the paused or unpaused state of the game varied based on how synchronously

the start and select buttons were released. This I fixed, at Gim's suggestion, by implementing an extra state in the game logic such that after a reset, there is a small break before the pause button will register.

After fixing this, I integrated the opening and closing screens, and then had a lot of fun testing my finished game.

4. Conclusion

I am very happy with my project. I feel it came out looking smooth and free of bugs and almost professional, which is what I wanted. I was really hoping to come out of this with a final product that felt finished and polished, and that was what I ended up with. I learned a lot about pipelining, data representation, implementing graphical systems in hardware, the usefulness of writing programs to do tedious tasks for you, how to perform image manipulations, random number generation, and even new debugging skills from this project. I am very proud of my end result, and thus I feel I have succeeded at the objectives of this final project.

Appendix: Verilog Code

Utilities.v

```
//This file contains simple modules which might be useful for many video applications

//////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
// (from Project 5 Code)
//
//////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock_65mhz, noisy, clean);
  input reset, clock_65mhz, noisy;
  output clean;

  reg [20:0] count;
  reg new, clean;

  always @(posedge clock_65mhz)
    if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
    else if (noisy != new) begin new <= noisy; count <= 0; end
    else if (count == 650000) clean <= new;
    else count <= count+1;

endmodule


//////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
// (from Project 5 Code)
//
//////////////////////////////////////////////////////////////////////////////

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
  input vclock;
  output [10:0] hcount;
  output [9:0] vcount;
  output vsync, hsync, blank;

  reg hsync, vsync, hblank, vblank, blank;
  reg [10:0] hcount;    // pixel number on current line
  reg [9:0] vcount;      // line number

  // horizontal: 1344 pixels total
  // display 1024 pixels per line
```

```
  wire      hsyncon,hsyncoff,hreset,hblankon;
  assign    hblankon = (hcount == 1023);
  assign    hsyncon = (hcount == 1047);
  assign    hsyncoff = (hcount == 1183);
  assign    hreset = (hcount == 1343);

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

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

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

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


//////////////////////////////////////////////////////////////////////////////////
//
// timer: count a specific number of input signals, then output a tick
//
//////////////////////////////////////////////////////////////////////////////////

module timer(period,enable,clock,reset,tick);
      input [5:0] period;
      input enable, clock, reset;
      output tick;

      reg tick;
      reg [9:0] count;

      always @ (posedge clock)
            begin
                  //start count from beginning on reset
```

```
        if (reset)
            begin
                count <= 0;
                tick <= 0;
            end
        //otherwise increase count once per enable signal
        else
            begin
                if (enable)
                    begin
                        if (count == period - 1)
                            begin
                                count <= 0;
                                tick <= 1;
                            end
                        else
                            begin
                                count <= count + 1;
                            end
                    end
            end

            if (tick)
                tick <= 0;

    end

endmodule
```

labkit.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//////////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//              and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
```

```
//             actually populated on the boards. (The boards support up to
//             256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//             value. (Previous versions of this file declared this port to
//             be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//             actually populated on the boards. (The boards support up to
//             72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
////////////////////////////////////////////////////////////////////////////////

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
               ac97_bit_clock,

               vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
               vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
               vga_out_vsync,

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
               tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

               tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
               tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
               tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
               tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

               ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
               ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

               ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
               ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

               clock_feedback_out, clock_feedback_in,

               flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
               flash_reset_b, flash_sts, flash_byte_b,

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,
```

```
        disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
        disp_reset_b, disp_data_in,

        button0, button1, button2, button3, button_enter, button_right,
        button_left, button_down, button_up,

        switch,

        led,

        user1, user2, user3, user4,

        daughtercard,

        systemace_data, systemace_address, systemace_ce_b,
        systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

        analyzer1_data, analyzer1_clock,
        analyzer2_data, analyzer2_clock,
        analyzer3_data, analyzer3_clock,
        analyzer4_data, analyzer4_clock);

output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
    vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
    tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
    tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
    tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
    tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output [18:0] ram1_address;
```

```
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;

input  clock_feedback_in;
output clock_feedback_out;

inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input  disp_data_in;
output  disp_data_out;

input  button0, button1, button2, button3, button_enter, button_right,
     button_left, button_down, button_up;
input  [7:0] switch;
output [7:0] led;

inout [31:0] user1, user2, user3, user4;

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
input  systemace_irq, systemace_mpbrdy;

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
          analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
```

```
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input
```

```
    // Flash ROM
    assign flash_data = 16'hZ;
    assign flash_address = 24'h0;
    assign flash_ce_b = 1'b1;
    assign flash_oe_b = 1'b1;
    assign flash_we_b = 1'b1;
    assign flash_reset_b = 1'b0;
    assign flash_byte_b = 1'b1;
    // flash_sts is an input

    // RS-232 Interface
    assign rs232_txd = 1'b1;
    assign rs232_rts = 1'b1;
    // rs232_rxd and rs232_cts are inputs

    // PS/2 Ports
    // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

    // LED Displays
    assign disp_blank = 1'b1;
    assign disp_clock = 1'b0;
    assign disp_rs = 1'b0;
    assign disp_ce_b = 1'b1;
    assign disp_reset_b = 1'b0;
    assign disp_data_out = 1'b0;
    // disp_data_in is an input

    // Buttons, Switches, and Individual LEDs
    assign led = 8'hFF;
    // button0, button1, button2, button3, button_enter, button_right,
    // button_left, button_down, button_up, and switches are inputs

    // User I/Os
//    assign user1 = 32'hZ;
    assign user2 = 32'hZ;
    assign user3 = 32'hZ;
    assign user4 = 32'hZ;

    // Daughtercard Connectors
    assign daughtercard = 44'hZ;

    // SystemACE Microprocessor Port
    assign systemace_data = 16'hZ;
    assign systemace_address = 7'h0;
    assign systemace_ce_b = 1'b1;
    assign systemace_we_b = 1'b1;
    assign systemace_oe_b = 1'b1;
    // systemace_irq and systemace_mpbrdy are inputs
```

```
    // Logic Analyzer
assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

    //Snake Implemented Here

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

    // power-on reset generation
    wire power_on_reset;    // remain high for first 16 clocks
    SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
            .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
    defparam reset_sr.INIT = 16'hFFFF;

    wire [2:0] tile_type, num_lives;
    wire reset, user_reset, speed2, speed1, speed0;
    wire [9:0] location, snake_location;
    wire [23:0] vga_color;
    wire new_apple, apple_done, new_frame,
                vga_blank_b, vga_clock, vga_hsync, vga_vsync, tick, occupied;
    wire collision, snake_busy, pause, pause_button, new_life, move, grow;
    wire [1:0] type, screen;
    wire [3:0] orientation, tile_orientation;
    wire [9:0] head_location, apple_location_rev;
    wire [9:0] length;
    wire up, down, left, right, start, select, a, b, snake_up, snake_down, snake_left, snake_right;

    parameter RESET_TIME = 1024;

    //Reset if start and select pushed simultaneously
    assign reset = power_on_reset | user_reset;

    debounce speed2switch(reset, clock_65mhz, switch[2], speed2);
    debounce speed1switch(reset, clock_65mhz, switch[1], speed1);
    debounce speed0switch(reset, clock_65mhz, switch[0], speed0);
```

```
debounce updebounce(reset, clock_65mhz, up, snake_up);
debounce downdebounce(reset, clock_65mhz, down, snake_down);
debounce leftdebounce(reset, clock_65mhz, left, snake_left);
debounce rightdebounce(reset, clock_65mhz, right, snake_right);
debounce resetdebounce(power_on_reset, clock_65mhz, (start & select), user_reset);
debounce pausedebounce(reset, clock_65mhz, (start | select), pause_button);

randomizer apple_generator(.new(new_apple), .clock(clock_65mhz), .reset(reset),
                           .number(apple_location_rev), .done(apple_done));

game_logic logic(.apple_location_rev(apple_location_rev), .apple_done(apple_done), .tick(tick),
                           .location_from_graphics(location),
                           .pause_button(pause_button), .snake_busy(snake_busy),
                           .occupied(occupied), .snake_type(type),
                           .head_location(head_location),
                           .snake_orientation(orientation), .length(length),
                           .collision(collision), .clock(clock_65mhz), .reset(reset),
                           .pause(pause), .new_life(new_life),
                           .new_apple(new_apple), .tile_orientation(tile_orientation),
                           .tile_type(tile_type), .num_lives(num_lives),
                           .snake_location(snake_location), .move(move),
                           .grow(grow), .screen(screen));

graphics graph(.screen(screen), .speed({speed2,speed1,speed0}), .tile_type(tile_type),
                           .orientation(tile_orientation), .num_lives(num_lives),
                           .new_life(new_life), .pause(pause), .clock(clock_65mhz),
                           .reset(reset), .location(location),
                           .new_frame(new_frame), .vga_color(vga_color),
                           .vga_blank_b(vga_blank_b), .vga_clock(vga_clock),
                           .vga_hsync(vga_hsync), .vga_vsync(vga_vsync));

defparam graph.RESET_TIME = RESET_TIME;

snake_timer ticker(.speed({speed2,speed1,speed0}), .new_frame(new_frame), .new_life(new_life),
                           .clock(clock_65mhz), .reset(reset), .tick(tick));

snake avatar(.up(snake_up), .down(snake_down), .left(snake_left), .right(snake_right),
                           .location(snake_location), .move(move), .grow(grow),
                           .new_life(new_life), .pause(pause), .clock(clock_65mhz),
                           .reset(reset), .occupied(occupied), .type(type),
                           .orientation(orientation), .head_location(head_location),
                           .collision(collision), .length(length), .busy(snake_busy));

NEShandler controller(.power(user1[0]), .data(user1[1]), .clock(clock_65mhz), .reset(reset),
                           .pulse(user1[3]), .latch(user1[2]), .a(a), .b(b), .start(start),
                           .select(select), .up(up), .down(down), .left(left),
                           .right(right));

assign user1[31:4] = 28'hZ;
```

```
    // VGA Output.  In order to meet the setup and hold times of the
    // AD7125, we send it ~clock_65mhz.
    assign vga_out_red = vga_color[23:16];
    assign vga_out_green = vga_color[15:8];
    assign vga_out_blue = vga_color[7:0];
    assign vga_out_sync_b = 1'b1;    // not used
    assign vga_out_blank_b = vga_blank_b;
    assign vga_out_pixel_clock = vga_clock;
    assign vga_out_hsync = vga_hsync;
    assign vga_out_vsync = vga_vsync;

endmodule
```

snake_timer.v

```
//snake_timer module outputs a tick whenever the snake should move one tile
//input new_frame: snake has completed one frame of movement
//input new_life: snake is beginning a level
//input clock: global 65mHz clock
//input reset: global reset signal
//output tick: goes high for one clock cycle when snake has moved enough frames to cross a tile
module snake_timer(speed, new_frame, new_life, clock, reset, tick);
	input [2:0] speed;
	input new_frame, new_life, clock, reset;
	output tick;
	reg [5:0] frames_per_tile;

	//reset on reset or new_life
	wire clicker_reset;
	assign clicker_reset = reset | new_life;

	always @(posedge clock)
		begin
			if (clicker_reset | tick)
				begin
					case (speed)
						3'b000: frames_per_tile <= 32;
						3'b001: frames_per_tile <= 16;
						3'b010: frames_per_tile <= 8;
						3'b011: frames_per_tile <= 4;
						3'b100: frames_per_tile <= 2;
						default: frames_per_tile <= 1;
					endcase
				end
		end

	//Use timer utility to generate ticks
	timer clicker(.period(frames_per_tile), .enable(new_frame), .clock(clock), .reset(clicker_reset),
				.tick(tick));

endmodule
```

NEShandler.v

```
//NES handler module takes input from an NES controller and returns the relevant signals
//input data: data line from NES controller
//input clock: global 65mHz clock
//input reset: global reset signal
//output pulse: pulse to NES controller
//output latch: latch to NES controller
//output a: high when the A button is pressed
//output b: high when the B button is pressed
//output start: high when the Start button is pressed
//output select: high when the Select button is pressed
//output up: high when the up button is pressed
//output down: high when the down button is pressed
//output left: high when the left button is pressed
//output right: high when the right button is pressed
module NEShandler(data, clock, reset, power, pulse, latch, a, b, start, select, up, down, left, right);

	input data, clock, reset;
	output power, pulse, latch, a, b, start, select, up, down, left, right;

	reg pulse;
	reg [7:0] buttons;
	reg [3:0] button_counter;
	reg [4:0] pulse_counter;

	assign power = 1;

	assign a = ~buttons[0];
	assign b = ~buttons[1];
	assign start = ~buttons[3];
	assign select = ~buttons[2];
	assign up = ~buttons[4];
	assign down = ~buttons[5];
	assign left = ~buttons[6];
	assign right = ~buttons[7];

	assign latch = (button_counter == 8) ? 1 : 0;

	always @ (posedge clock)
		begin
			if (reset)
				begin
					button_counter <= 0;
					pulse_counter <= 0;
					pulse <= 0;
					buttons <= 8'hFF;
				end
			else
```

```
                begin
                    if (pulse_counter == 31)
                        begin
                            pulse_counter <= 0;
                            pulse <= ~pulse;
                            //Increase button_counter on posedge of pulses
                            if (~pulse)
                                begin
                                    button_counter <= button_counter + 1;
                                    if (button_counter < 8)
                                        begin
                                            buttons[button_counter] <= data;
                                        end
                                    else
                                        begin
                                            button_counter <= 0;
                                        end
                                end
                        end
                    else
                        begin
                            pulse_counter <= pulse_counter + 1;
                        end
                end
        end

endmodule
```

randomizer.v

```
//NES handler module takes input from an NES controller and returns the relevant signals
//input data: data line from NES controller
//input clock: global 65mHz clock
//input reset: global reset signal
//output pulse: pulse to NES controller
//output latch: latch to NES controller
//output a: high when the A button is pressed
//output b: high when the B button is pressed
//output start: high when the Start button is pressed
//output select: high when the Select button is pressed
//output up: high when the up button is pressed
//output down: high when the down button is pressed
//output left: high when the left button is pressed
//output right: high when the right button is pressed
module NEShandler(data, clock, reset, power, pulse, latch, a, b, start, select, up, down, left, right);

	input data, clock, reset;
	output power, pulse, latch, a, b, start, select, up, down, left, right;

	reg pulse;
	reg [7:0] buttons;
	reg [3:0] button_counter;
	reg [4:0] pulse_counter;

	assign power = 1;

	assign a = ~buttons[0];
	assign b = ~buttons[1];
	assign start = ~buttons[3];
	assign select = ~buttons[2];
	assign up = ~buttons[4];
	assign down = ~buttons[5];
	assign left = ~buttons[6];
	assign right = ~buttons[7];

	assign latch = (button_counter == 8) ? 1 : 0;

	always @ (posedge clock)
		begin
			if (reset)
				begin
					button_counter <= 0;
					pulse_counter <= 0;
					pulse <= 0;
					buttons <= 8'hFF;
				end
			else
```

```
                begin
                    if (pulse_counter == 31)
                        begin
                            pulse_counter <= 0;
                            pulse <= ~pulse;
                            //Increase button_counter on posedge of pulses
                            if (~pulse)
                                begin
                                    button_counter <= button_counter + 1;
                                    if (button_counter < 8)
                                        begin
                                            buttons[button_counter] <= data;
                                        end
                                    else
                                        begin
                                            button_counter <= 0;
                                        end
                                end
                        end
                    else
                        begin
                            pulse_counter <= pulse_counter + 1;
                        end
                end
        end

endmodule
```

snake.v

```
//snake_module deals with the snake location and movement - attributes at location may not appear
// correctly for up to 1000 clock cycles after a move or grow signal
//input up: up button status
//input down: down button status
//input left: left button status
//input right: right button status
//input location: location at which to check for snake segment
//input move: enable snake to move in appropriate direction
//input grow: enable snake to grow while moving in appropriate direction
//input new_life: restart snake at original length and position
//input pause: pause snake (no registering of buttons)
//input clock: global system clock
//input reset: global system reset
//output occupied: high if snake is present in location
//output type: snake segment type (00=head, 01=odd, 10=even, 11=tail)
//output orientation: orientation of snake segment
// (0000=up, 0001= right, 0010=down, 0011=left, 0100=up-right, 0101=up-left,
//  0110=right-up, 0111=right-down, 1000=down-right, 1001=down-left,
//  1010=left-up, 1011=left-down)
//output head_location: 10 bit location of head
//output collision: high if recent move resulted in collision
//output length: 10 bit length of snake
//output busy: high when snake is busy writing to memory (moving, growing, etc.)

module snake(up, down, left, right, location, move, grow, new_life, pause, clock, reset,
                                occupied, type, orientation, head_location, collision, length, busy);

    input up, down, left, right, move, grow, new_life, pause, clock, reset;
    input [9:0] location;
    output occupied, collision, busy;
    output [1:0] type;
    output [3:0] orientation;
    output [9:0] head_location;
    output [9:0] length;

    reg [3:0] state;
    reg collision, parity, we, read;
    reg [1:0] type, head_direction, new_direction;
    reg [9:0] rcount, length, head_location, tail_location, address;
    reg [13:0] segment_in;
    wire [13:0] segment_out;

    //States
    parameter STATE_RESET = 0;
    parameter STATE_000 = 1;
    parameter STATE_001 = 2;
    parameter STATE_010 = 3;
```

```
parameter STATE_011 = 4;
parameter STATE_100 = 5;
parameter STATE_101 = 6;
parameter STATE_110 = 7;
parameter STATE_111 = 8;
parameter STATE_MEM_INIT = 9;

parameter WIDTH = 32;
parameter HEIGHT = 23;

ram1024x14 snakemem(address, clock, segment_in, segment_out, we);

//Occupied if orientation not 1111
assign occupied = (state == STATE_000) & (segment_out[13:10]!=4'b1111);

//Orientation is first 4 bits of segment
assign orientation = segment_out[13:10];

//Busy when not in standard read state
assign busy = (state != STATE_000);

always @ (posedge clock)
    begin
        //type based only on address and current parity value
        //delayed to match occupied
        type <= (address == head_location) ?
                        2'b00 :
                        (address == tail_location) ?
                                2'b11 :
                                (address[5]^address[0]^parity) ? 2'b01 : 2'b10;

        if (reset | new_life)
            begin
                state <= STATE_RESET;
            end
        else if (state == STATE_MEM_INIT)
            begin
                we <= 1;
                rcount <= rcount - 1;
                address <= rcount;
                //10 segments, the rest empty
                case (rcount)
                    10'b0000110101: segment_in <= 14'b00011111111111;
                    10'b0001010101: segment_in <= 14'b00010000110101;
                    10'b0001110101: segment_in <= 14'b00010001010101;
                    10'b0010010101: segment_in <= 14'b00010001110101;
                    10'b0010110101: segment_in <= 14'b00010010010101;
                    10'b0011010101: segment_in <= 14'b00010010110101;
                    10'b0011110101: segment_in <= 14'b00010011010101;
```

```
            10'b0100010101: segment_in <= 14'b00010011110101;
            10'b0100110101: segment_in <= 14'b00010100010101;
            10'b0101010101: segment_in <= 14'b00010100110101;
            default: segment_in <= 14'b11111111111111;
        endcase

        //Go to state 0 when done
        if (rcount==0)
            begin
                state <= STATE_000;
            end
    end
//If memory is done initializing
else
    begin
        //Deal with buttons
        if (~pause)
            begin
                case (head_direction)
                    2'b00:    //up
                              begin
                                  if (up)
                                      new_direction <= 2'b00;
                                  else if (right)
                                      new_direction <= 2'b01;
                                  else if (left)
                                      new_direction <= 2'b11;
                              end
                    2'b01:    //right
                              begin
                                  if (right)
                                      new_direction <= 2'b01;
                                  else if (up)
                                      new_direction <= 2'b00;
                                  else if (down)
                                      new_direction <= 2'b10;
                              end
                    2'b10:    //down
                              begin
                                  if (down)
                                      new_direction <= 2'b10;
                                  else if (right)
                                      new_direction <= 2'b01;
                                  else if (left)
                                      new_direction <= 2'b11;
                              end
                    2'b11:    //left
                              begin
                                  if (left)
```

```
                    new_direction <= 2'b11;
                else if (up)
                    new_direction <= 2'b00;
                else if (down)
                    new_direction <= 2'b10;
            end
        default: new_direction <= head_direction;
    endcase
end

case (state)
    STATE_RESET: begin
                    //prepare to reinitialize memory
                    rcount <= 1023;
                    head_location <= 10'b0101010101;
                    tail_location <= 10'b0000110101;
                    head_direction <= 2'b01;
                    new_direction <= 2'b01;
                    length <= 10;
                    collision <= 0;
                    parity <= 0;
                    we <= 0;
                    read <= 0;
                    address <= location;
                    segment_in <= 14'b11111111111111;
                    state <= STATE_MEM_INIT;
                end
    STATE_000:  begin
                    //Read from location in memory
                    we <= 0;
                    address <= location;

                    //Increase Length
                    if (grow & ~collision)
                        begin
                            length <= length + 1;
                        end
                    //Check for collisions with sides
                    if ((move | grow) & ~collision)
                        begin
                            we <= 0;
                            case (new_direction)
                                2'b00:begin
                                        //Check for collision with top
                                        if (head_location[4:0] == 5'b00000)
                                            begin
                                                collision <= 1;
                                            end
                                        else
```

```
                begin
                    //If moving, start with tail,
                    //if growing, just move head
                    state <= move ?     STATE_001 :
                                        STATE_100;
                end
            end
        2'b01:begin
                //Check for collision with right
                if (head_location[9:5] == WIDTH-1)
                    begin
                        collision <= 1;
                    end
                else
                    begin
                        //If moving, start with tail,
                        //if growing, just move head
                        state <= move ?     STATE_001 :
                                            STATE_100;
                    end
            end
        2'b10:begin
                //Check for collision with bottom
                if (head_location [4:0] == HEIGHT-1)
                    begin
                        collision <= 1;
                    end
                else
                    begin
                        //If moving, start with tail,
                        //if growing, just move head
                        state <= move ?     STATE_001 :
                                            STATE_100;
                    end
            end
        2'b11:begin
                //Check for collision with left
                if (head_location[9:5] == 5'b00000)
                    begin
                        collision <= 1;
                    end
                else
                    begin
                        //If moving, start with tail,
                        //if growing, just move head
                        state <= move ?     STATE_001 :
                                            STATE_100;
                    end
            end
```

```
                    default: begin
                                state <= STATE_RESET;
                             end
                endcase
            end
        end
STATE_001:  //Start looking for tail at head
        begin
            we <= 0;
            address <= head_location;
            read <= 0;
            state <= STATE_010;
        end
STATE_010:  //Follow snake until tail is reached
        begin
            read <= ~read;
            //Wait for memory to be read
            if (read)
                begin
                    //Found Tail
                    if (segment_out[9:0] == tail_location)
                        begin
                            we <= 1;
                            segment_in <= {segment_out[13:10],
                                           10'b1111111111};
                            state <= STATE_011;
                        end
                    else
                        begin
                            we <= 0;
                            //Follow snake to next segment
                            address <= segment_out[9:0];
                        end
                end
            else
                begin
                    we <= 0;
                end
        end
STATE_011:  //Remove old tail
        begin
            //Set tail location to address of previous segment
            tail_location <= address;
            //Write over previous tail
            we <= 1;
            address <= tail_location;
            segment_in <= 14'b11111111111111;

            state <= STATE_100;
```

```
                end
STATE_100:      //Find the old head location
                begin
                    we <= 0;
                    address <= head_location;
                    read <= 0;
                    state <= STATE_101;
                end
STATE_101:      //Change old head orientation
                begin
                    read <= ~read;
                    //Wait for memory to be read
                    if (read)
                        begin
                            //Change orientation of previous head segment
                            // (0000=up, 0001= right, 0010=down, 0011=left,
                            //  0100=up-right, 0101=up-left,
                            //  0110=right-up, 0111=right-down,
                            //  1000=down-right, 1001=down-left,
                            //  1010=left-up, 1011=left-down)
                            we <= 1;
                            case ({head_direction, new_direction})
                                4'b0001: segment_in <= {4'b0100, segment_out[9:0]};
                                4'b0011: segment_in <= {4'b0101, segment_out[9:0]};
                                4'b0100: segment_in <= {4'b0110, segment_out[9:0]};
                                4'b0110: segment_in <= {4'b0111, segment_out[9:0]};
                                4'b1001: segment_in <= {4'b1000, segment_out[9:0]};
                                4'b1011: segment_in <= {4'b1001, segment_out[9:0]};
                                4'b1100: segment_in <= {4'b1010, segment_out[9:0]};
                                4'b1110: segment_in <= {4'b1011, segment_out[9:0]};
                                default: segment_in <= segment_out;
                            endcase

                            state <= STATE_110;
                        end
                    else
                        begin
                            we <= 0;
                        end
                end
STATE_110:      //Check for collision with other snake segment
                begin
                    we <= 0;
                    case (new_direction)
                        2'b00:    address <= head_location - 10'b0000000001;
                        2'b01:    address <= head_location + 10'b0000100000;
                        2'b10:    address <= head_location + 10'b0000000001;
                        2'b11:    address <= head_location - 10'b0000100000;
                        default:  address <= head_location;
```

```
                endcase
                read <= 0;

                state <= STATE_111;
            end
        STATE_111:  //Place new head
            begin
                read <= ~read;
                //Wait for memory to be read
                if (read)
                    begin
                        we <= 1;
                        //Head direction is just direction moved in,
                        // point to previous head
                        segment_in <= {2'b00, new_direction, head_location};
                        head_direction <= new_direction;
                        //Move to correct new address
                        head_location <= address;

                        //Nonempty tile
                        if (segment_out[13:10] != 4'b1111)
                            begin
                                collision <= 1;
                            end
                        //Switch Parities
                        parity <= ~parity;

                        state <= STATE_000;
                    end
                else
                    begin
                        we <= 0;
                    end
            end
        default:    state <= STATE_RESET;
    endcase
  end
 end
endmodule
```

game_logic.v

```
//Game logic module controlls level transitions, collisions with walls and apples,
// and communication between snake and graphics modules
//input apple_location_rev: recieves a new location for the apple {5 bits y, 5 bits x}
//input apple_done: high when valid location is on the apple_location line
//input tick: high when the snake should move
//input location_from_graphics: tile to return contents of
//input pause_button: user button which toggles game pause
//input snake_busy: high when snake cannot be read from
//input occupied: high if snake is present at snake_location (2 cycle delay)
//input snake_type: type of snake segment at snake_location (2 cycle delay)
// (00=head, 01=odd, 10=even, 11=tail)
//input head_location: current location of snake head
//input snake_orientation: orientation of snake segment at snake_location (2 cycle delay)
// (0000=up, 0001= right, 0010=down, 0011=left, 0100=up-right, 0101=up-left,
//  0110=right-up, 0111=right-down, 1000=down-right, 1001=down-left,
//  1010=left-up, 1011=left-down)
//input length: length of snake
//input collision: high if snake has collided with self or edge of screen
//input clock: global 65mHz clock
//input reset: global reset signal
//output pause: high when game should be paused
//output new_life: high when a level should begin
//output busy: high when snake is busy
//output new_apple: high when a new apple location should be computed
//output tile_orientation: orientation of tile at location_from_graphics (3 cycle delay)
// (0000=up, 0001= right, 0010=down, 0011=left, 0100=up-right, 0101=up-left,
//  0110=right-up, 0111=right-down, 1000=down-right, 1001=down-left,
//  1010=left-up, 1011=left-down)
//output tile_type: type of tile at location_from_graphics (3 cycle delay)
// (000=blank, 001=head, 010=odd, 011=even, 100=tail, 101=wall, 110=apple, 111=collision)
//output num_lives: number of remaining lives
//output snake_location: location to request information about from snake
//output move: high when snake should move a tile
//output grow: high when snake has eaten an apple
//output screen: indicates screen to be displayed (00=title, 01=game, 11=end)
module game_logic(apple_location_rev, apple_done, tick, location_from_graphics, pause_button,
                  snake_busy, occupied, snake_type, head_location, snake_orientation,
                  length, collision, clock, reset, pause, new_life, new_apple, tile_orientation,
                  tile_type, num_lives, snake_location, move, grow, screen);
    input [9:0] apple_location_rev, location_from_graphics, head_location, length;
    input apple_done, tick, pause_button, snake_busy, occupied, collision, clock, reset;
    input [1:0] snake_type;
    input [3:0] snake_orientation;
    output pause, new_life, new_apple, move, grow;
    output [3:0] tile_orientation;
    output [2:0] tile_type;
    output [2:0] num_lives;
```

```
output [9:0] snake_location;
output [1:0] screen;

parameter NUM_LEVELS = 5;
parameter MAX_LENGTH = 25;
parameter WAIT_TO_MOVE = 1024;
reg [9:0] movecounter;
reg pause, pause_button_prev, pause_next, new_life, apple, new_apple,
        collided, stopped, grow_next, delayed_tick;
reg [9:0] snake_location;
reg [2:0] tile_type, num_lives, level;
reg [1:0] screen;
reg [3:0] state;
reg [9:0] address;
reg [3:0] snake_orientation_prev;
reg [9:0] location_prev2, location_prev1, location_prev;
wire [NUM_LEVELS-1:0] wall;
wire is_wall;
reg is_wall_prev;
wire [9:0] apple_location;
reg [19:0] resetting;

//Levels
level0 level_0(address, clock, wall[0]);
level1 level_1(address, clock, wall[1]);
level2 level_2(address, clock, wall[2]);
level3 level_3(address, clock, wall[3]);
level4 level_4(address, clock, wall[4]);

//States
parameter STATE_RESET = 0;
parameter STATE_0000 = 1;
parameter STATE_0001 = 2;
parameter STATE_0010 = 3;
parameter STATE_0011 = 4;
parameter STATE_0100 = 5;
parameter STATE_0101 = 6;
parameter STATE_0110 = 7;
parameter STATE_0111 = 8;
parameter STATE_1000 = 9;
parameter STATE_1001 = 10;
parameter STATE_1010 = 11;
parameter STATE_1011 = 12;
parameter STATE_1100 = 13;

//is_wall indicates presence of wall at address in current level
assign is_wall = wall[level];
//Correct x and y for apple location
assign apple_location = {apple_location_rev[4:0], apple_location_rev[9:5]};
```

```
//Grow on tick when collision with apple has occurred
assign grow = ~pause_next & delayed_tick & grow_next;
//Move on tick when not growing
assign move = ~pause_next & delayed_tick & ~grow;
//Orientation based on tile type and snake orientation
assign tile_orientation = (tile_type == 3'b001 | tile_type == 3'b010 | tile_type == 3'b011
                                    | tile_type == 3'b100) ? snake_orientation_prev : 4'b0000;

always @ (posedge clock)
    begin
        if (reset)
            begin
                state <= STATE_RESET;
            end
        else
            begin
                //Remember snake orientation
                snake_orientation_prev <= snake_orientation;
                //Remember wall status
                is_wall_prev <= is_wall;
                //Remember location from graphics
                location_prev <= location_prev1;
                location_prev1 <= location_prev2;
                location_prev2 <= location_from_graphics;
                //Toggle pause on button press
                pause_button_prev <= stopped ? 1 : pause_button;
                if (~pause_button_prev & pause_button)
                    begin
                        pause <= ~ pause;
                    end

                if (delayed_tick)
                    begin
                        delayed_tick <= 0;
                    end

                if (tick | movecounter!=0)
                    begin
                        if (movecounter == WAIT_TO_MOVE-1)
                            begin
                                movecounter <= 0;
                                delayed_tick <= 1;
                            end
                        else
                            begin
                                movecounter <= movecounter + 1;
                            end
                    end
```

```
if (state == STATE_RESET | ~snake_busy)
    begin
        case (state)
            STATE_RESET: begin
                             pause <= 1;
                             new_life <= 0;
                             new_apple <= 0;
                             apple <= 0;
                             tile_type <= 3'b000;
                             num_lives <= 4;
                             address <= location_from_graphics;
                             snake_location <= 10'b0000000000;
                             level <= 0;
                             state <= STATE_1100;
                             collided <= 0;
                             stopped <= 1;
                             grow_next <= 0;
                             movecounter <= 0;
                             pause_button_prev <= 1;
                             resetting <= 1048575;
                             screen <= 0;
                         end
            STATE_1100:    //Wait for input to be valid
                         begin
                             if (resetting > 0)
                                 begin
                                     resetting <= resetting - 1;
                                 end
                             else
                                 begin
                                     stopped <= 0;
                                     if (~pause)
                                         begin
                                             state <= STATE_0100;
                                             screen <= 1;
                                         end
                                 end
                         end
            STATE_0000:    //Give done time to propagate
                         begin
                             new_apple <= 0;
                             state <= STATE_1010;
                         end
            STATE_1010:    //Wait for apple to generate
                         begin
                             if (apple_done)
                                 begin
                                     address <= apple_location;
                                     snake_location <= apple_location;
```

```
                    state <= STATE_0001;
                end
            end
STATE_0001:     //Wait for is_wall to be valid
            begin
                state <= STATE_0010;
            end
STATE_0010:     //Check apple location for wall
            begin
                if (is_wall)
                    begin
                        new_apple <= 1;
                        state <= STATE_0000;
                    end
                else
                    begin
                        state <= STATE_0011;
                    end
            end
STATE_0011:     //Check apple location for snake
            begin
                if (occupied)
                    begin
                        new_apple <= 1;
                        state <= STATE_0000;
                    end
                else
                    begin
                        apple <= 1;
                        state <= STATE_0100;
                    end
            end
STATE_0100:     //Normal operation
            begin
                new_life <= 0;
                //Move on ticks
                if (delayed_tick)
                    begin
                        //Pause if needed
                        if (pause_next)
                            begin
                                pause <= 1;
                                pause_next <= 0;
                            end
                        else
                            begin
                                grow_next <= 0;
                                //check for wall collision
                                state <= STATE_0101;
```

```
                    end
                end
            else
                begin
                    //Figure out type of tile
                    address <= location_from_graphics;
                    snake_location <= location_from_graphics;

                    if (occupied)
                        begin
                            case (snake_type)
                                2'b00:    begin
                                              //Replace head with explosion
                                              // on collision
                                              if (collided)
                                                  tile_type <= 3'b111;
                                              else
                                                  tile_type <= 3'b001;
                                          end
                                2'b01:    tile_type <= 3'b010;
                                2'b10:    tile_type <= 3'b011;
                                2'b11:    tile_type <= 3'b100;
                                default:  tile_type <= 3'b000;
                            endcase
                        end
                    else if (is_wall_prev)
                        begin
                            tile_type <= 3'b101;
                        end
                    else if (apple & location_prev == apple_location)
                        begin
                            tile_type <= 3'b110;
                        end
                    else
                        begin
                            tile_type <= 3'b000;
                        end
                end
            end
        STATE_0101:    //Wait for snake to move, then check new head location
            begin
                if (~snake_busy)
                    begin
                        address <= head_location;
                        state <= STATE_1001;
                    end
            end
        STATE_1001:    //Wait for is_wall to be valid
            begin
```

```
                    state <= STATE_0110;
                end
STATE_0110:     //Check for collisions
                begin
                    if (is_wall | collision)
                        begin
                            pause_next <= 1;
                            collided <= 1;
                            //Out of lives
                            if (num_lives - 1 == 0)
                                begin
                                    stopped <= 1;
                                    state <= STATE_0100;
                                    screen <= 3;
                                end
                            else
                                begin
                                    state <= STATE_1000;
                                end
                        end
                    //Collision with apple causes growth
                    else if (apple & head_location == apple_location)
                        begin
                            grow_next <= 1;
                            apple <= 0;
                            state <= STATE_0111;
                        end
                    else
                        begin
                            //Generate apple if there is none
                            if (~apple)
                                begin
                                    new_apple <= 1;
                                    state <= STATE_0000;
                                end
                            else
                                begin
                                    state <= STATE_0100;
                                end
                        end
                end
STATE_0111:     //Check for new level
                begin
                    if (length >= MAX_LENGTH
                                && level < NUM_LEVELS-1)
                        begin
                            state <= STATE_1011;
                            level <= level + 1;
                            new_life <= 1;
```

```
                        pause <= 1;
                        grow_next <= 0;
                    end
                else
                    begin
                        state <= STATE_0100;
                    end
            end
STATE_1000:     //Wait for level restart
            begin
                //Figure out type of tile
                address <= location_from_graphics;
                snake_location <= location_from_graphics;

                if (occupied)
                    begin
                        case (snake_type)
                            2'b00:    begin
                                          //Replace head with explosion
                                          // on collision
                                          if (collided)
                                              tile_type <= 3'b111;
                                          else
                                              tile_type <= 3'b001;
                                      end
                            2'b01:    tile_type <= 3'b010;
                            2'b10:    tile_type <= 3'b011;
                            2'b11:    tile_type <= 3'b100;
                            default:  tile_type <= 3'b000;
                        endcase
                    end
                else if (is_wall_prev)
                    begin
                        tile_type <= 3'b101;
                    end
                else if (apple & location_prev == apple_location)
                    begin
                        tile_type <= 3'b110;
                    end
                else
                    begin
                        tile_type <= 3'b000;
                    end

                //Pause if needed
                if (pause_next & delayed_tick)
                    begin
                        pause <= 1;
                        pause_next <= 0;
```

```
                    end
                //Start next level once pause is pushed
                if (~pause_next & ~pause)
                    begin
                        new_life <= 1;
                        num_lives <= num_lives - 1;
                        collided <= 0;
                        apple <= 0;
                        state <= STATE_0100;
                    end
            end
STATE_1011:     //Wait for next level
            begin
                new_life <= 0;

                //Figure out type of tile
                address <= location_from_graphics;
                snake_location <= location_from_graphics;

                if (occupied)
                    begin
                        case (snake_type)
                            2'b00:  begin
                                        //Replace head with explosion
                                        // on collision
                                        if (collided)
                                            tile_type <= 3'b111;
                                        else
                                            tile_type <= 3'b001;
                                    end
                            2'b01:  tile_type <= 3'b010;
                            2'b10:  tile_type <= 3'b011;
                            2'b11:  tile_type <= 3'b100;
                            default: tile_type <= 3'b000;
                        endcase
                    end
                else if (is_wall_prev)
                    begin
                        tile_type <= 3'b101;
                    end
                else if (apple & location_prev == apple_location)
                    begin
                        tile_type <= 3'b110;
                    end
                else
                    begin
                        tile_type <= 3'b000;
                    end
```

```
                                        //Start next level once pause is pushed
                                        if (~pause)
                                            begin
                                                collided <= 0;
                                                apple <= 0;
                                                state <= STATE_0100;
                                            end
                                    end
                            default:        state <= STATE_RESET;
                        endcase
                    end
                end
            end

endmodule
```

image_processor.v

```
//Image processor module transforms images into a correct orientation
//by giving the corresponding pixel location  based on a desired location and an orientation
module image_processor(x_in, y_in, orientation, clock, x_out, y_out, valid);

    input [4:0] x_in, y_in;
    input [3:0] orientation;
    input clock;
    output [4:0] x_out, y_out;
    output valid;
    reg [4:0] orientation_prev[1:0];
    reg [4:0] x, y, x_out, y_out;
    reg [4:0] up_x, up_y, right_x, right_y, down_x, down_y, left_x, left_y;
    reg valid;
    wire [10:0] upright_out, upleft_out, rightup_out, rightdown_out,
                downright_out, downleft_out, leftup_out, leftdown_out;

    upright uprightbend({x, y}, clock, upright_out);
    upleft upleftbend({x, y}, clock, upleft_out);
    rightup rightupbend({x, y}, clock, rightup_out);
    rightdown rightdownbend({x, y}, clock, rightdown_out);
    downright downrightbend({x, y}, clock, downright_out);
    downleft downleftbend({x, y}, clock, downleft_out);
    leftup leftupbend({x, y}, clock, leftup_out);
    leftdown leftdownbend({x, y}, clock, leftdown_out);

    always @(orientation_prev[0] or up_x or up_y or right_x or right_y
                    or down_x or down_y or left_x or left_y
                    or upright_out or upleft_out or rightup_out or rightdown_out
                    or downright_out or downleft_out or leftup_out or leftdown_out)
        begin
            //0000=up, 0001= right, 0010=down, 0011=left, 0100=up-right, 0101=up-left,
            //0110=right-up, 0111=right-down, 1000=down-right, 1001=down-left,
            //1010=left-up, 1011=left-down
            case (orientation_prev[0])
                4'b0000: begin
                            x_out = up_x;
                            y_out = up_y;
                            valid = 1;
                        end
                4'b0001: begin
                            x_out = right_x;
                            y_out = right_y;
                            valid = 1;
                        end
                4'b0010: begin
                            x_out = down_x;
                            y_out = down_y;
```

```
                valid = 1;
            end
4'b0011:    begin
                x_out = left_x;
                y_out = left_y;
                valid = 1;
            end
4'b0100:    begin
                x_out = upright_out[9:5];
                y_out = upright_out[4:0];
                valid = upright_out[10];
            end
4'b0101:    begin
                x_out = upleft_out[9:5];
                y_out = upleft_out[4:0];
                valid = upleft_out[10];
            end
4'b0110:    begin
                x_out = rightup_out[9:5];
                y_out = rightup_out[4:0];
                valid = rightup_out[10];
            end
4'b0111:    begin
                x_out = rightdown_out[9:5];
                y_out = rightdown_out[4:0];
                valid = rightdown_out[10];
            end
4'b1000:    begin
                x_out = downright_out[9:5];
                y_out = downright_out[4:0];
                valid = downright_out[10];
            end
4'b1001:    begin
                x_out = downleft_out[9:5];
                y_out = downleft_out[4:0];
                valid = downleft_out[10];
            end
4'b1010:    begin
                x_out = leftup_out[9:5];
                y_out = leftup_out[4:0];
                valid = leftup_out[10];
            end
4'b1011:    begin
                x_out = leftdown_out[9:5];
                y_out = leftdown_out[4:0];
                valid = leftdown_out[10];
            end
default:    begin
                x_out = 0;
```

```
                        y_out = 0;
                        valid = 0;
                    end
            endcase
        end

    always @ (posedge clock)
        begin
            x <= x_in;
            y <= y_in;
            orientation_prev[0] <= orientation_prev[1];
            orientation_prev[1] <= orientation;
            //Set correct outputs for all possible transformations
            up_x <= x;
            up_y <= y;
            right_x <= y;
            right_y <= 31 - x;
            down_x <= 31 - x;
            down_y <= 31 - y;
            left_x <= 31 - y;
            left_y <= x;
        end

endmodule
```

graphics.v

```
//
module graphics(screen, speed, tile_type, orientation, num_lives, new_life, pause, clock, reset,
                        location, new_frame, vga_color, vga_blank_b, vga_clock, vga_hsync, vga_vsync);

    input [1:0] screen;
    input [2:0] speed;
    input [2:0] tile_type;
    input [3:0] orientation;
    input [2:0] num_lives;
    input new_life, pause, clock, reset;
    output [9:0] location;
    output [23:0] vga_color;
    output new_frame, vga_blank_b, vga_clock, vga_hsync, vga_vsync;

    reg [23:0] vga_color;
    reg [2:0] color;
    reg [9:0] tileindex[6:0];
    reg [7:0] livesindex[6:0];
    reg [9:0] snakeindex[3:0];
    reg [17:0] screenindex[7:0];
    reg [2:0] tile_type_prev[4:0];
    reg valid_prev[2:0];
    wire [10:0] hcount;
    wire [9:0] vcount;
    wire hsync, vsync, blank;
    wire [2:0] title_out, endscreen_out, apple_, egg_, even_, explosion_, head_, odd_, tail_;
    reg [2:0] apple_out, egg_out, even_out, explosion_out, head_out, odd_out, tail_out;
    wire [31:0] lr_;
    reg [31:0] lr_out;
    wire [4:0] x, y;
    wire valid;

    //cycles for reset to finish
    parameter RESET_TIME = 0;
    reg [10:0] waiting;
    parameter REGLENGTH = 11;
    reg [REGLENGTH-1:0] vga_blank_b_wait, vga_hsync_wait, vga_vsync_wait;

    //smooth movement
    reg [5:0] pixels_per_frame;
    reg [4:0] counter;
    reg [4:0] shifted_x, shifted_y, delayed_x, delayed_y;
    wire [4:0] snake_x, snake_y;
    reg saving;
    reg saving_prev[3:0];
    reg [9:0] save_counter;
    reg [9:0] save_counter_prev[3:0];
```

```
wire [2:0] last_tile_type;
reg [2:0] last_tile_type_prev[7:0];
reg [4:0] y_prev[2:0];
wire [9:0] prevstate_location;
reg [3:0] last_tail_orientation;
wire [3:0] snake_orientation;
reg [3:0] orientation_prev[4:0];

//Give colors actual values according to their indices
parameter COLOR_000 = 24'b000000000000000000000000;
parameter COLOR_001 = 24'b000000001000000000000000;
parameter COLOR_010 = 24'b000000001111111100000000;
parameter COLOR_011 = 24'b100000000000000000000000;
parameter COLOR_100 = 24'b111111110000000000000000;
parameter COLOR_101 = 24'b100000000100000000000000;
parameter COLOR_110 = 24'b111111111000000000000000;
parameter COLOR_111 = 24'b111111111111111100000000;

parameter BLACK =  3'b000;
parameter GREEN =  3'b001;
parameter LIME =   3'b010;
parameter BRICK =  3'b011;
parameter RED =    3'b100;
parameter BROWN =  3'b101;
parameter ORANGE = 3'b110;
parameter YELLOW = 3'b111;

title title_screen(screenindex[0], clock, title_out);
endscreen end_screen(screenindex[0], clock, endscreen_out);

apple apple_segment(tileindex[0], clock, apple_);
head head_segment({snake_x,snake_y}, clock, head_);
even even_segment({snake_x,snake_y}, clock, even_);
odd odd_segment({snake_x,snake_y}, clock, odd_);
tail tail_segment({snake_x,snake_y}, clock, tail_);
explosion explosion_segment(tileindex[0], clock, explosion_);
//Shift egg segment by 16
egg egg_segment({~tileindex[0][9], tileindex[0][8:0]}, clock, egg_);
lives_remaining_image lives_rem_image(livesindex[0], clock, lr_);

//Transform images
image_processor transformer(snakeindex[0][9:5],snakeindex[0][4:0], snake_orientation,
                                    clock, x, y, valid);
assign snake_orientation = (last_tile_type_prev[5]==3'b100 & tile_type!=3'b001) ?
                                                                    last_tail_orientation
                                                                    : orientation;
assign snake_x = (tile_type_prev[2] == last_tile_type_prev[2]) ? delayed_x : shifted_x;
assign snake_y = (tile_type_prev[2] == last_tile_type_prev[2]) ? delayed_y : shifted_y;
```

```
//Store tiles from previous iteration
previous_tiles prevstate(prevstate_location, clock, tile_type, last_tile_type, saving_prev[0]);
assign prevstate_location = saving_prev[0] ? save_counter_prev[0] : {hcount[9:5],vcount[9:5]};

//location is first five bits of hcount and vcount
assign location = saving ? save_counter : {hcount[9:5],vcount[9:5]};
//new_frame is true once vcount = 767 and hcount = 1024 + the number of register delays
assign new_frame = (hcount == 1024 + REGLENGTH) & (vcount == 767) & ~pause;

//Generate XVGA Signals
xvga vga(.vclock(clock), .hcount(hcount), .vcount(vcount), .hsync(hsync), .vsync(vsync),
                .blank(blank));

assign vga_clock = ~clock;
assign vga_blank_b = ~vga_blank_b_wait[0];
assign vga_hsync = vga_hsync_wait[0];
assign vga_vsync = vga_vsync_wait[0];

always @ (posedge clock)
    begin
        //delay signals
        vga_blank_b_wait[REGLENGTH-2:0] <= vga_blank_b_wait[REGLENGTH-1:1];
        vga_hsync_wait[REGLENGTH-2:0] <= vga_hsync_wait[REGLENGTH-1:1];
        vga_vsync_wait[REGLENGTH-2:0] <= vga_vsync_wait[REGLENGTH-1:1];

        vga_blank_b_wait[REGLENGTH-1] <= blank;
        vga_hsync_wait[REGLENGTH-1] <= hsync;
        vga_vsync_wait[REGLENGTH-1] <= vsync;

        tileindex[0] <= tileindex[1];
        tileindex[1] <= tileindex[2];
        tileindex[2] <= tileindex[3];
        tileindex[3] <= tileindex[4];
        tileindex[4] <= tileindex[5];
        tileindex[5] <= tileindex[6];
        tileindex[6] <= {hcount[4:0],vcount[4:0]};

        livesindex[0] <= livesindex[1];
        livesindex[1] <= livesindex[2];
        livesindex[2] <= livesindex[3];
        livesindex[3] <= livesindex[4];
        livesindex[4] <= livesindex[5];
        livesindex[5] <= livesindex[6];
        livesindex[6] <= hcount[7:0];

        snakeindex[0] <= snakeindex[1];
        snakeindex[1] <= snakeindex[2];
        snakeindex[2] <= snakeindex[3];
        snakeindex[3] <= {hcount[4:0],vcount[4:0]};
```

```
screenindex[0] <= screenindex[1];
screenindex[1] <= screenindex[2];
screenindex[2] <= screenindex[3];
screenindex[3] <= screenindex[4];
screenindex[4] <= screenindex[5];
screenindex[5] <= screenindex[6];
screenindex[6] <= screenindex[7];
screenindex[7] <= {vcount[9:1],hcount[9:1]};

tile_type_prev[0] <= tile_type_prev[1];
tile_type_prev[1] <= tile_type_prev[2];
tile_type_prev[2] <= tile_type_prev[3];
tile_type_prev[3] <= tile_type_prev[4];
tile_type_prev[4] <= tile_type;

valid_prev[0] <= valid_prev[1];
valid_prev[1] <= valid_prev[2];
valid_prev[2] <= valid;

saving_prev[0] <= saving_prev[1];
saving_prev[1] <= saving_prev[2];
saving_prev[2] <= saving_prev[3];
saving_prev[3] <= saving;

save_counter_prev[0] <= save_counter_prev[1];
save_counter_prev[1] <= save_counter_prev[2];
save_counter_prev[2] <= save_counter_prev[3];
save_counter_prev[3] <= save_counter;

y_prev[0] <= y_prev[1];
y_prev[1] <= y_prev[2];
y_prev[2] <= y;

last_tile_type_prev[0] <= last_tile_type_prev[1];
last_tile_type_prev[1] <= last_tile_type_prev[2];
last_tile_type_prev[2] <= last_tile_type_prev[3];
last_tile_type_prev[3] <= last_tile_type_prev[4];
last_tile_type_prev[4] <= last_tile_type_prev[5];
last_tile_type_prev[5] <= last_tile_type_prev[6];
last_tile_type_prev[6] <= last_tile_type_prev[7];
last_tile_type_prev[7] <= last_tile_type;

orientation_prev[0] <= orientation_prev[1];
orientation_prev[1] <= orientation_prev[2];
orientation_prev[2] <= orientation_prev[3];
orientation_prev[3] <= orientation_prev[4];
orientation_prev[4] <= orientation;
```

```
//Save value of last tail orientation
if (saving_prev[0] & tile_type==3'b100)
    begin
        last_tail_orientation <= orientation;
    end

//y for snake segments should be shifted by counter
shifted_x <= x;
shifted_y <= y - counter;
delayed_x <= x;
delayed_y <= y;

//Pipeline
apple_out <= apple_;
egg_out <= egg_;
even_out <= even_;
explosion_out <= explosion_;
head_out <= head_;
odd_out <= odd_;
tail_out <= tail_;
lr_out <= lr_;

//Assign colors
//vga_color is one of 8 possible
case (color)
    3'b001: vga_color <= COLOR_001;
    3'b010: vga_color <= COLOR_010;
    3'b011: vga_color <= COLOR_011;
    3'b100: vga_color <= COLOR_100;
    3'b101: vga_color <= COLOR_101;
    3'b110: vga_color <= COLOR_110;
    3'b111: vga_color <= COLOR_111;
    default: vga_color <= COLOR_000;
endcase

//Wait for reset to finish
if (reset | new_life)
    begin
        //pixels per frame is based on speed
        case (speed)
            3'b000:  begin
                        pixels_per_frame <= 1;
                        counter <= 31;
                     end
            3'b001:  begin
                        pixels_per_frame <= 2;
                        counter <= 30;
                     end
            3'b010:  begin
```

```
                    pixels_per_frame <= 4;
                    counter <= 28;
                end
            3'b011:  begin
                    pixels_per_frame <= 8;
                    counter <= 24;
                end
            3'b100:  begin
                    pixels_per_frame <= 16;
                    counter <= 16;
                end
            default: begin
                    pixels_per_frame <= 32;
                    counter <= 0;
                end
        endcase
        color <= BLACK;
        waiting <= RESET_TIME;
        saving <= 1;
        save_counter <= 0;
    end
else if (waiting != 0)
    begin
        waiting <= waiting - 1;
    end
else if (saving)
    begin
        if (save_counter == 1023)
            begin
                save_counter <= 0;
                saving <= 0;
            end
        else
            begin
                save_counter <= save_counter + 1;
            end
    end
else if (new_frame)
    begin
        if (counter == 0)
            begin
                //pixels per frame is based on speed
                case (speed)
                    3'b000:  begin
                                pixels_per_frame <= 1;
                                counter <= 31;
                            end
                    3'b001:  begin
                                pixels_per_frame <= 2;
```

```
                        counter <= 30;
                    end
                3'b010:  begin
                        pixels_per_frame <= 4;
                        counter <= 28;
                    end
                3'b011:  begin
                        pixels_per_frame <= 8;
                        counter <= 24;
                    end
                3'b100:  begin
                        pixels_per_frame <= 16;
                        counter <= 16;
                    end
                default: begin
                        pixels_per_frame <= 32;
                        counter <= 0;
                    end
            endcase

            saving <= 1;
        end
    else
        begin
            counter <= counter - pixels_per_frame;
        end
end
//show game
else if (screen == 1)
    begin
        if (vcount > 735 & vcount < 768)
            begin
                //Display Lives Remaining Bar
                if (hcount > REGLENGTH-2 & hcount - (REGLENGTH-1) < 212)
                    begin
                        if (lr_out[31-tileindex[0][4:0]])
                            begin
                                color <= BLACK;
                            end
                        else
                            begin
                                color <= BRICK;
                            end
                    end
                else if (hcount - (REGLENGTH-1) > 239 & hcount - (REGLENGTH-1) < 272)
                    begin
                        if (num_lives > 1)
                            begin
                                color <= egg_out;
```

```
                end
            else
                begin
                    color <= BRICK;
                end
        end
    else if (hcount - (REGLENGTH-1) > 303 & hcount - (REGLENGTH-1) < 336)
        begin
            if (num_lives > 2)
                begin
                    color <= egg_out;
                end
            else
                begin
                    color <= BRICK;
                end
        end
    else if (hcount - (REGLENGTH-1) > 367 &  hcount - (REGLENGTH-1) < 400)
        begin
            if (num_lives > 3)
                begin
                    color <= egg_out;
                end
            else
                begin
                    color <= BRICK;
                end
        end
    else
        begin
            color <= BRICK;
        end
 end
else if (valid_prev[0] &
            ~(last_tile_type_prev[0]==3'b100 & tile_type_prev[0]==3'b001
              & orientation_prev[0]!=last_tail_orientation & y_prev[0] < counter))
  begin
    //display previous if looking at top part of image and image is not explosion
    // or apple or near miss with tail
    if (tile_type_prev[0]!=3'b111 & tile_type_prev[0]!=3'b110 & y_prev[0] < counter)
        begin
            //Display color based on type
            case (last_tile_type_prev[0])
                //head
                3'b001:   color <= head_out;
                //odd
                3'b010:   color <= odd_out;
                //even
                3'b011:   color <= even_out;
```

```
                        //tail
                        3'b100:  color <= tail_out;
                        //wall
                        3'b101:  color <= BRICK;
                        //apple
                        3'b110:  color <= apple_out;
                        //collision
                        3'b111:  color <= explosion_out;
                        //Background
                        default: color <= BLACK;
                    endcase
                end
            //otherwise display current
            else
                begin
                    //Display color based on type
                    case (tile_type_prev[0])
                        //head
                        3'b001:  color <= head_out;
                        //odd
                        3'b010:  color <= odd_out;
                        //even
                        3'b011:  color <= even_out;
                        //tail
                        3'b100:  color <= tail_out;
                        //wall
                        3'b101:  color <= BRICK;
                        //apple
                        3'b110:  color <= apple_out;
                        //collision
                        3'b111:  color <= explosion_out;
                        //Background
                        default: color <= BLACK;
                    endcase
                end
        end
    else
        begin
            color <= BLACK;
        end
end
//show title screen
else if (screen == 0)
    begin
        color <= title_out;
    end
//show end screen
else
    begin
```

```
                color <= endscreen_out;
            end
        end
endmodule
```