FPGA Fruit Ninja

By Nadia Salahuddin & Lydia Sun

Table of Contents

Overview

Fruit Ninja is a quite popular video game developed by Halfbrick and released in 2010. Countless Android and iOS users have played the game, and if not, have certainly heard of it. The conjunction of a straightforward user interface and Skinner box mechanics gave rise to an extremely popular game which was commercially successful by numerous standards.

The objective of the game is simple: drag your finger across a touch screen on a smart device of your choice and slice as many fruit as possible. Difficulty can vary based on which mode is selected in the start menu, and there exist various unique features to each mode. The simplicity of the design makes for an accessible game that can be catered to players of all ages.

The translation from software to hardware is always an interesting one; the methods for performing tasks are not always the same for both, and there are surely different problems to think about based on which medium is being used. We wanted to see how the different challenges in developing Fruit Ninja would change when recreating the game in hardware. The rest of this document describes the various revelations and trials we ran into in the creation of FPGA Fruit Ninja.

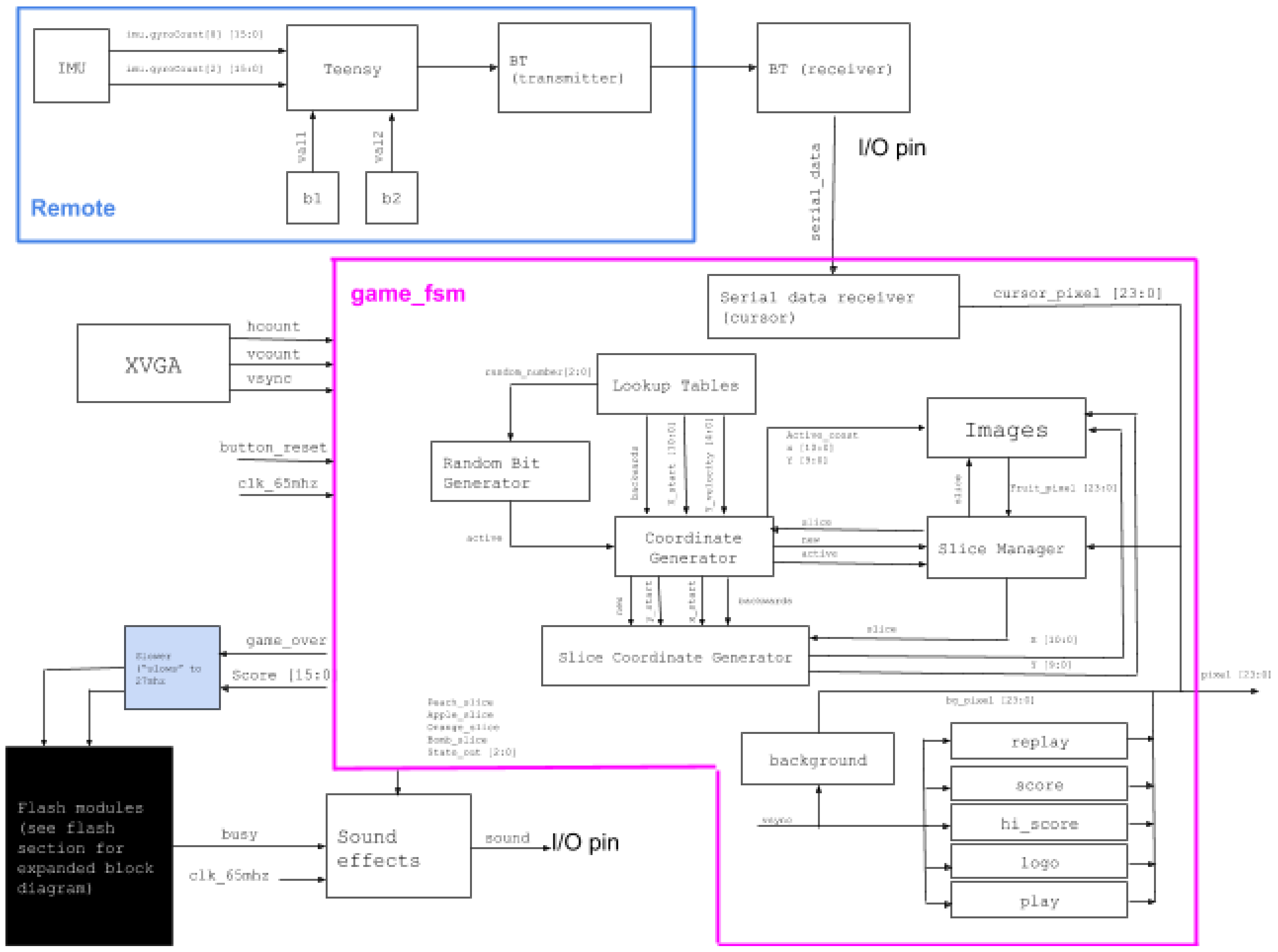
Motivation

One of our motivations for making a game was its visibility. We personally find more satisfaction in a project with visible results. It also seemed like a fun idea to recreate something that we were already familiar with, but with our own spin. Additionally, we were especially interested in creating a remote because it combines previous knowledge from 6.08 (Arduino, IMUs, C++) with new concepts from 6.111 (serial). We were also eager to interface with different components like flash memory and sound creation.

Summary

Our team decided to create an FPGA version of Fruit Ninja. Instead of using a touch screen, we created a remote with which a user could control a cursor. We loaded all the game graphics into the FPGA's BRAM, kept track of all time high score with flash memory, and added sound effects with a piezo buzzer. The basic rules and mechanics of the game are similar to the original, with fruit launching at random, game over on touching a bomb, and lives decreasing if fruits drop without being sliced.

Block Diagram



Modules

Game FSM (Joint)

module game_fsm (input [7:0] highscore, input vclock, input vsync, input serial_data, input reset, input [10:0] hcount, input [9:0] vcount, output [23:0] pixel, output [7:0] ultimate_score, output game_state, output apple_slice, output orange_slice, output peach_slice, output bomb_slice, output [1:0] fell_out, output [1:0] state_out, input resetbutton);

This module does much of the heavy lifting in terms of where other modules ultimately are wired together. There are a couple of key parts of the game FSM:

1. Dealing with game state transitions: As the name of the module implies, this module determines when the game enters the start, play, and end states.
    a. START: This is the startup screen. It displays some images; pressing the start button on the remote whilst placing the cursor on the play button allows the user to start playing the game.
    b. PLAY: The user may play the game until he loses three lives by letting three fruit fall to the bottom of the screen unsliced, or by swiping through a bomb.
    c. GAME_OVER: The user has lost the game. This screen displays the score the user accumulated in this iteration of the game and the high score. Pressing the start button on the remote whilst placing the cursor on the replay button allows the user to play the game once again.
2. Score and lives display: Using the image-drawing module from lab 3 (for rectangular shapes), we displayed the lives the user currently has using the `fell` and `score` outputs from the coordinate generator. The high score is also displayed at the end screen based on the input from the flash to the game FSM module.
    a. The score is displayed using numerous instantiations of the rectangular image-drawing module and displaying them or not based on what the score should be (so, a seven segment display but displayed using VGA).
    b. The location of the score shifts based on which game state the user is in, so the position registers for the score digits are changed in game state transitions at the clock edge.
3. Graphics: The spatial precedence in the game is determined in this module as below (elements sorted by z values):

    background < fruits < bomb < cursor

    The fruits have an arbitrary order. In general, order is set by checking for overlaps between elements and assigning pixel values to the desired top image. The state machine also determines which elements are on screen in which states.

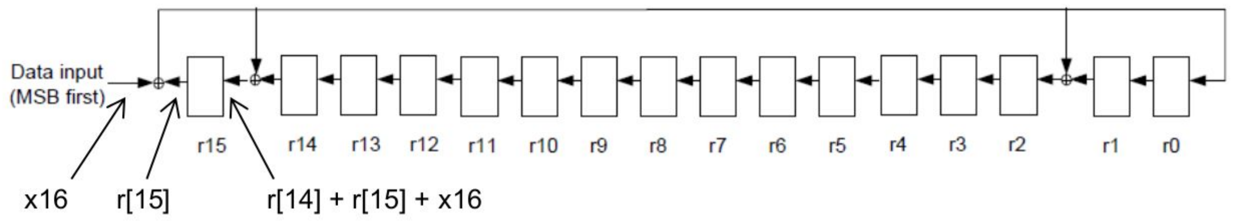
Random number generator (Nadia)

module randombitsgenerator (input clk, input data, output [15:0] randomnumber);

The cyclic redundancy check is quite similar to a linear-feedback shift register in their mathematical nature. LFSRs can seem to produce “random” bit strings if certain parameters are adjusted. This allows us to use a CRC-16 calculation to be a good proxy for “randomness”, though not truly random (Rosenberg, 1997). Of course, trying out different parameters may be useful, as some data strings work better than others, as do different lengths of CRC generators and indexing into different bits for our lookup tables.

This module shifts in data bits to a CRC computation from an external source and outputs a 16 bit number that we can subsequently use in our lookup table to choose parameters that seem somewhat random for fruit motion generation.

The following is how CRC-16 is calculated:



(Lpset 6, 2018)

Lookup Table (Nadia)

module lookuptable (input clk, input [2:0] randomnumber, output reg [4:0] yvel1, output reg backwards1, output reg [9:0] xcostart1);

This module uses the random number generator to create usable parameters for the coordinate generator module. Based on the 3 bit input *randomnumber*, a case statement determines an initial y-velocity, backwards (whether the fruit moves left or right), and the initial x-coordinate for the fruit. These values are pre-selected by the envisioner based on what looks nice on the screen. Because the the gravity is chosen to be “2” for the duration of the game, initial velocities are chosen to be multiples of 2 to avoid overflow when determining the peak of the parabola in the coordinate generator module.

Coordinate Generator (Nadia)

module coord_generator (input rdy, input [2:0] slice, input active, input vsync, input [4:0] yvel, input backwards, input [9:0] xcostart, output [9:0] x_coord, output reg [2:0] fell, output [9:0] y_coord, output reg activeconst, output reg [7:0] score, output [4:0] yvelocity, output reg new);

This module takes in the inputs `slice`, `active`, `vsync`, `backwards`, `yvel`, `xcostart`, and outputs an `x_coord`, `y_coord`, `activeconst`, `score`, `yvelocity`, and `new`. It performs the coordinate calculation that allows the fruit on the screen to rise and fall along a parabolic trajectory. Instead of using the traditional parabolic motion equations and pipelining, the module takes advantage of sequential circuitry: numbers can be multiplied by repeatedly adding (which is what multiplication is, in essence). The module describes a finite state machine with two states: START and CALC. The finite state machine will only do state transitions while a ready signal (`rdy`) is asserted. Additionally, `fell` and `y_coord` are reset at the positive edge of the ready signal (so as to restore lives at a new game and reduce a one-cycle flashing of the fruits and bomb) and the score is reset at the negative edge of the ready signal (so as to set the score to zero after a game is finished). Resetting the score at the positive edge proved to be an issue, if one is wondering.

START
the FSM latches on to an active bit (provided by the random number generator) and resets the y-coordinate, while taking in various parameters that determine the path in total from a lookup table (initial y velocity, initial x coordinate, and whether the fruit will travel backwards (`left`)). This latched active bit is then fed into `activeconst` to produce a level signal. This signal is routed to the image-drawing module to indicate that images that are inactive should not be drawn. `New` is asserted. This latching mechanism is important because the random number generator is changing at 65 MHz.

CALC
Y-coordinates and x-coordinates are generated. `New` is deasserted (this allows for a pulse signal when a new fruit appears on the screen; this is essential for deciding slice mechanics). The x-coordinate decreases or increases, based on the input `backwards`, by a fixed value:

```
x_coord1 <= (left)? x_coord1 - x_vel: x_coord1 + x_vel;
```

*left is the latched value of backwards provided as an input

The y-velocity changes every 9 (this is up to the user) cycles, as this means the fruit will move more slowly and hence more aesthetically. The y-velocity increases or decreases based on which part of the parabola the fruit is currently on, as does the y-coordinate:

```
y_coord1 <= (reachedzero && y_coord1 >768)? 768: (y_up)? (y_coord1 -
             y_vel) : (y_coord1 + y_vel);
```

*reachedzero goes high when the y-velocity has, you guessed it, reached zero; y_up goes low when reachedzero goes high

The `fell`, `score`, and `new` outputs are used to calculate game mechanics external to this module. `Fell` is calculated by seeing how many fruit go past the bottom of the screen while being active and unsliced. Score increments at the positive edge of the slice input. Finally, a state transition occurs when the fruit has traveled an entire parabola and is now past the bottom of the screen.

Coordinate Generator for Slices (Nadia)

module coord_generator_slice (input begincalc, input vsync, input [4:0] yvel, input new, input backwards, *input [9:0] xcostart, input [9:0] ycostart, output [9:0] x_coord, output [9:0] y_coord);*

Logically, this module is very similar to the regular coordinate generator except for the fact that it is a bit more specialized for bottom half slices of fruit. Essentially, there are two modules that determine the coordinates of the top (coord_generator) and bottom (coord_generator_slice) halves at all times, but control signals such as `active`, `new`, and `slice/begincalc` (the two are used synonymously across coord_generator and coord_generator_slice) are used to determine how exactly these coordinates are used.

The module calculates the coordinates for a downward parabolic motion from the peak of a trajectory. Such behavior necessitates the initial y velocity to be 0. The initial x-coordinate and y-coordinate are determined from the current x and y coordinates of the fruit's motion when it is unsliced. There are two states to this module's finite state machine: WAIT and CALC.

WAIT
In the wait state, the coordinates are such that the bottom half of the fruit is moving with the top half of the fruit, as it is unsliced (so the x-coordinate is the same and the y-coordinate is the same as the y-coordinate added to half the height of the picture so as to position the bottom half in the correct position). The state machine stays in this state until `begincalc` is asserted, which is the equivalent of the corresponding fruit being sliced.

CALC
In the calc state, the coordinates are generated and depend on where on the xy plane the fruit was split. The y-velocity still changes every nine cycles, as in the regular coordinate generator, but the y-velocity is either staying the same or incrementing. The x-coordinate depends on the `backwards` input; the bottom half could move either way. In this case, the input is always opposite the top half, as we wanted our fruit halves to separate in different directions for aesthetic purposes. When a new, active fruit is launched from the bottom, the finite state machine would go back to the wait state and again, the coordinates of the bottom half of the fruit would follow the top half of the fruit.

Slice Finite State Machine (Nadia)

module slice_dealer (input peachactive, input appleactive, input orangeactive, input clk, input apple_new, input orange_new, input peach_new, input [23:0] applepix, input [23:0] cursorpix, input [23:0] orangepix, input [23:0] peachpix, output applesliced, output orangesliced, output peachsliced);

This module takes in the pixel values for the different fruit on the screen and the cursor and determines whether a slice has occurred. Separate finite machines for the different fruit allow for the `slice` output to stay high for the corresponding fruit while it is still in motion. The module additionally deals with the problem of inactive fruit and their slice status (in that inactive fruit cannot be sliced). The two states for the finite state machine are as follows:

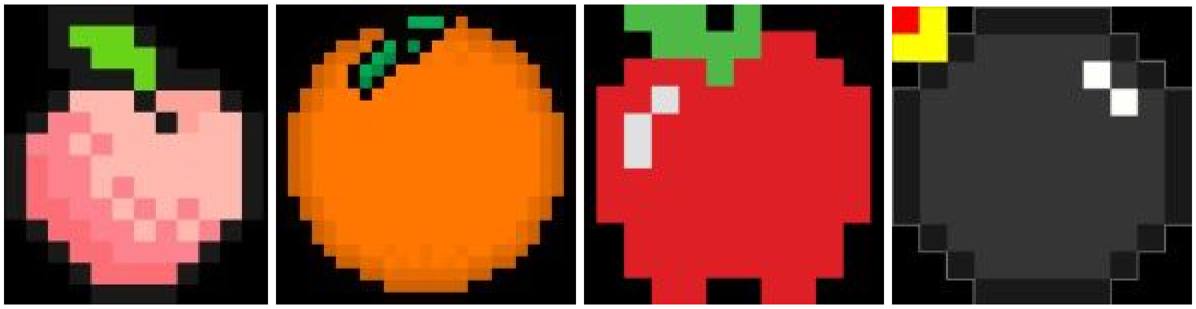
WAIT
This state is the idle equivalent. The state machine resides in this state until it detects an overlap between any of the fruit pixels and the cursor pixels. `Slice` is deasserted the entire time the FSM is in this state.

SLICE
This state asserts `slice` as long as a new, active fruit has not appeared. The consequence of this is `slice` being a level signal as opposed to a pulse signal which allows for calculations in other modules that require such a construct (or, really, it's a design choice).

Image Drawing Modules (Joint)

Fruit and Bombs

Fruit and bombs are 150 x 150 8-bit bitmaps (except for the peach, which is 132x150). We used this site to convert pngs to 8 bit bmp files. The coe files are generated with the MATLAB script provided under tools on the website and stored in the FPGA's BRAM. The background color on all the images is black, because the Verilog for the game replaces black pixels with the background color - all seemingly black colors in the fruit images are dark shades of grey.



Drawing

Fruit are only drawn if the fruit is considered active (`activeconst` from coordinate generator); otherwise, the fruit will not appear on screen or have any effect on game score/lives.

The fruits are drawn similarly to the picture_blob module provided in lab 3. However, because the fruits split apart, the top and bottom halves are actually drawn separately. Both read from the same ROM/coe file, - the top half of the fruit reads normally, but the bottom half of the fruit reads only from the bottom half of the coe file. Therefore, there are two image addresses (`image_addr_1` for the top and `image_addr_2` for the bottom):

```
assign image_addr_1 = (hcount-x) + (vcount-y) * WIDTH;
assign image_addr_2 = (hcount-xslice) + (vcount-yslice) * WIDTH +
(WIDTH * HEIGHT/2);
```

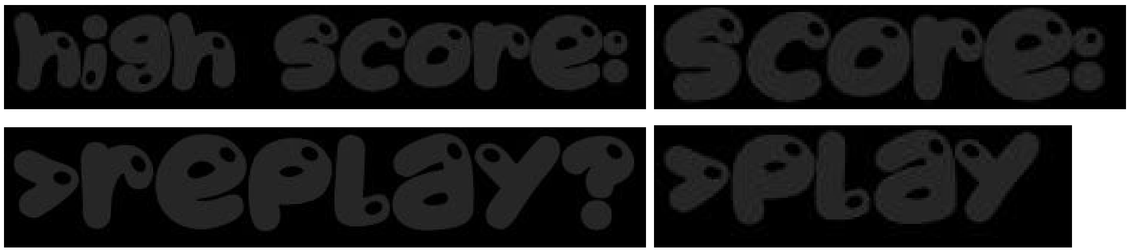
*xslice and yslice correspond to the x and y coordinates of the bottom half of the fruit.

When a fruit is not sliced, `image_addr_1` is used to index into the coe file. When the fruit is sliced, the top and bottom halves are drawn separately, and a register image_addr is set to either 1 or 2 depending on which half is being drawn. `Image_addr_2` indexes only into the bottom half of the coe file. `Image_addr` is passed to the rom, which generates a signal `image_bits` for mapping colors.

The bomb is drawn exactly like the `picture_blob` module from lab 3, because it cannot be "sliced" (except for requiring `active` to be high in order for it to be drawn).

Text & logo

The text and logos are also 8-bit bitmaps. All the text images are grayscale and only required one color coe file. These were made by downloading the font Juicy Fruit and exporting PowerPoint slides as images. These images also have a black background that is replaced by the game background.



fruit
ninja

Background (Lydia)

module bg (input vsync, output [23:0] pixel);

The background for the game shifts between shades of pink and blue. This is achieved by fixing green and blue values at 164 and 255, respectively, while varying red. At every pulse of `vsync`, red increases by 1 until it reaches 255, at which point it decreases by 1 until it reaches 0 and the cycle begins again. I chose the fixed blue and green values by looking up an online RGB slider and picking which colors I thought would look best for a background.

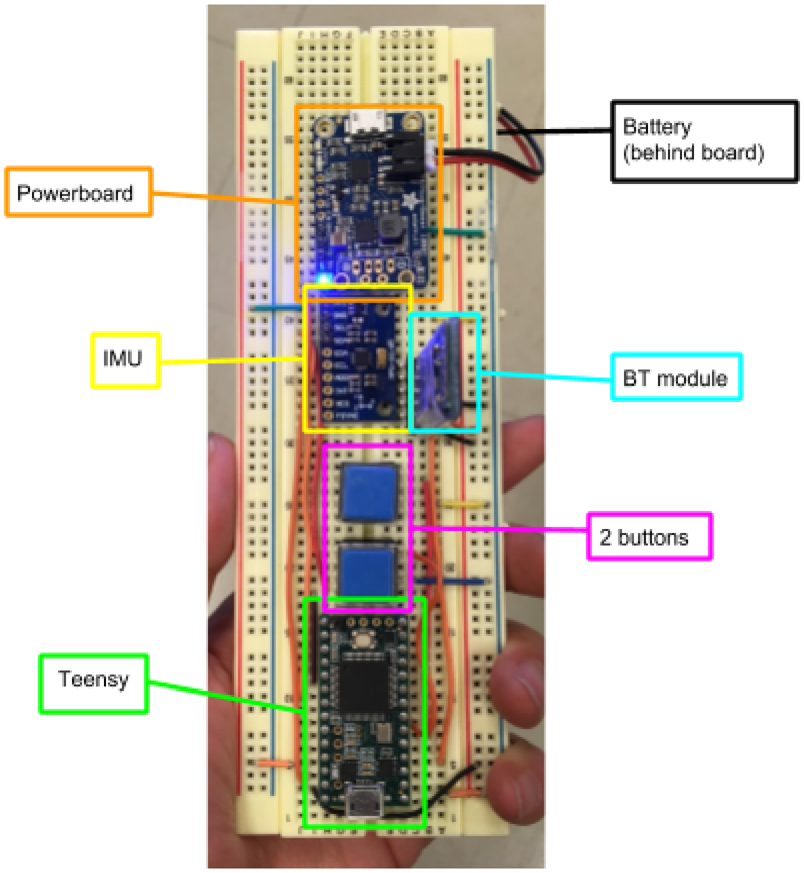
Remote (Lydia)

Description

The remote has two main purposes: to collect motion data from an IMU and process it, and to transmit data to the FPGA. The first task is accomplished by retrieving angular velocity from the gyroscope, integrating it to get position, and then translating the resulting values into a scale fit for the monitor (1024 x 768). The second task is accomplished by writing data to a pin (code provided by Joe) according to a serial protocol. The remote acts as the serial transmitter and the FPGA acts as the serial receiver.

Components

The remote includes several components:



Teensy
The Teensy is connected to everything else on the board. It computes x and y coordinates based on data from the IMU, and writes data to its TX pin to be sent by the transmitting BT module. Loop speed is set at 10 milliseconds.

2 buttons
The top button is meant for the user to select items, and the bottom button is meant to recalibrate the cursor position. Both buttons are active low. The top button's state is packaged into the data to be transmitted (see below for serial data transmission). A press on the bottom button triggers x and y coordinates to be reset to the center of the screen (coordinates (512, 389) on a 1024x768 screen).

MPU9250

This IMU is needed to retrieve angular velocity, so we used the gyroscope. The gyroscope measures 3 axes of motion, but we only needed two - the -z and x axes on the gyroscope, which correspond to the x and y axes, respectively, in the game and in relation to the screen. Integration is performed as so:

`velocity = velocity`$^{-1}$ `✕ δt (loop speed) ✕ leaky factor ✕ sensitivity`

The leaky factor is used to combat drift. Sensitivity helps the cursor move more with a smaller motion. The tricky part was balancing these two variables, because a lower leak is better for reducing drift, but a higher sensitivity is better for smoother and less constrained motion. Decreasing the loop speed also helps to create a smoother motion.

Adafruit BT module

The Bluetooth module was part of our stretch goals. Instead of wiring a connection from the Teensy's TX pin directly to an I/O pin on the FPGA, we connected BT modules to each of these. The BT modules are pre-programmed to connect to each other, and will reconnect and send data each time both are powered on. The tricky part is wiring them up correctly - the transmitting module receives data at its RX pin from the Teeny's TX pin, and the receiving module is connected to the FPGA with its TX pin.

Powerboard

This board allows the remote to be truly wireless, connecting power to a battery instead of an external source through micro-USB. As long as it is wired up correctly to supply power to the Teensy and the battery is charged, nothing more is needed for it to work. You can charge the battery by plugging in a micro-USB to the port on the board.

Serial data transmission

The data is packaged into 5 bytes to be sent from the Teensy - 2 bytes each for the 16 bit x and y coordinates, and 1 byte for the button. The data is big endian at the bit level and little endian at the byte level. The button data is only 1 bit long; the upper 7 bits are packed with 0s. The data is sent at 9600 baud.

Serial data receiver (Lydia)

This module is similar to the serial receiver from lab 5c. Every time a sample is collected from the incoming serial data, it is shifted into a register. Instead of terminating after a certain number of bits have been received like in lab 5c, this serial receiver terminates after a certain number of bytes have been received and re-centers after every byte (see more in challenges and difficulties). The state machine is as follows:

HIGH (default/start state)

In this state, we wait for a certain amount of time before we transition into FALLING to wait for a falling edge. This is to establish that the FSM will not start in the middle of a data stream. Originally, the time was set to 2 ms. It was decreased to 200 ns after adding in Bluetooth functionality because the BT modules send packets at unequal spacing; sometimes bytes 4 and 5 will be bunched closely to byte 1 of the next round of data, so we shouldn't wait for data to be high too long before looking for a falling edge.

FALLING

This state transitions into FIRST when serial data goes low.

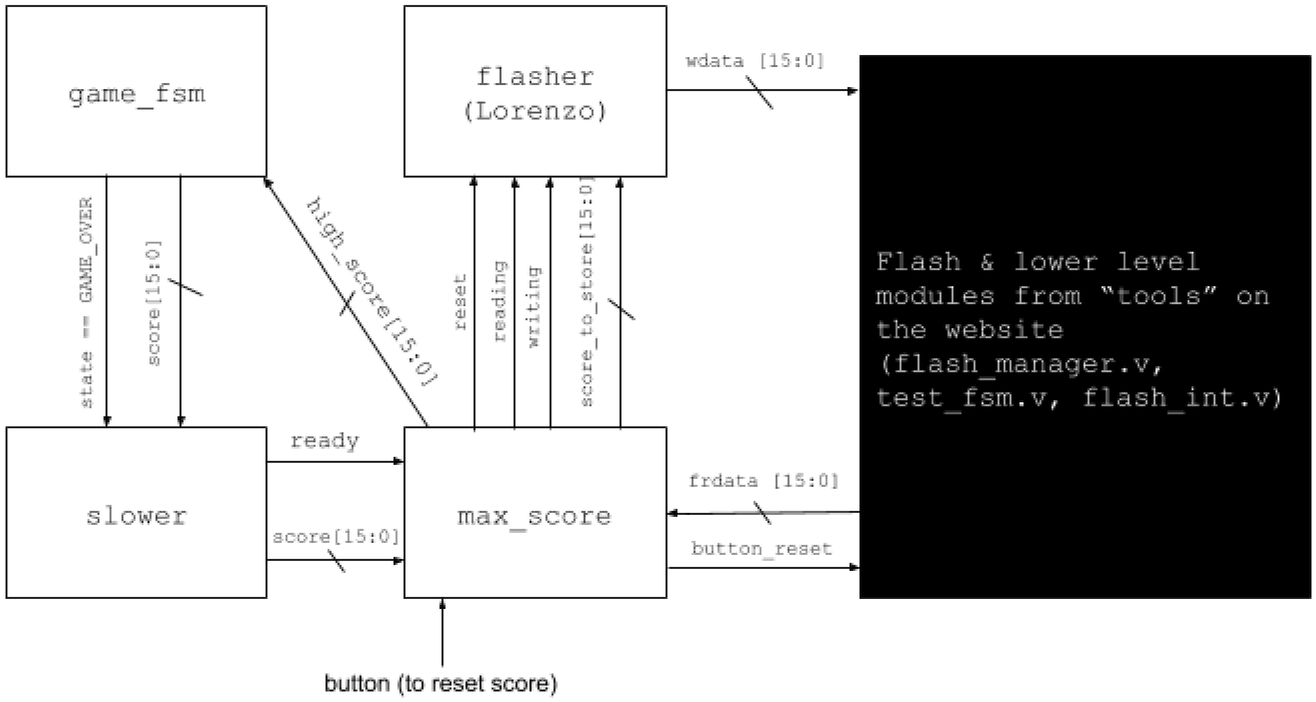
FIRST
The name of this state refers to the first bit of a byte, in which the retrieval of the start bit occurs 8 sampling clocks from the falling edge. After 8 sampling clocks, the state transitions to DATA.

DATA
In this state, a new bit is read in every 16 sampling clocks. After 8 bits have been collected, this state transitions back to FALLING in preparation for the next byte of data. If after 8 bits have been collected, 5 bytes have been collected in total (the total amount of data sent from the Teeny for this project is 5 bytes), the state transitions to DONE.

DONE
In the DONE state, the x coordinate, y coordinate, and button data are reconstructed from the bitstream according to its endianness.

Flash (Lydia)

Flash memory is used in this project to store a high score that persists through shutdowns. On startup, the system is designed to read the high score from flash and store it in a register. Every time the user enters game over in the current power-on, if the user has gained a new high score, the register value is replaced and the flash is rewritten.



Reading and Writing

In order to read and write to the flash, I used Lorenzo's modules from the 6.111 website and passed in signals from a max_score module, which determines when to read and write from flash. `Reading` and `writing` are the two signals which trigger their corresponding events when asserted in Lorenzo's module. The state machine for the max score module is as follows:

STARTUP
`Reading` is asserted and state immediately transitions to CHECK.

CHECK
Stay in this state if flash is busy; if flash is not busy, store the data coming from flash in a register and transition to IDLE.

IDLE
If a new high score has been achieved at game over, assert reset signals for one clock cycle, store the new high score in our register, put the new high score as data to be written to flash, and transition to RESET. Our project also supports the ability to reset the high score to 0, where all the signals are the same, except write data and the register are set to 0 and the events are triggered by a button press.

RESET
Reset signals are de-asserted. As soon as the flash is not busy, we transition to the STORE state and assert writing.

STORE
As soon as the flash is not busy, transition back to IDLE. The new score has now been written into flash.

Interface to game

All the flash modules are timed with a 27 mhz clock, while all the game modules are timed with a 65 mhz clock. Because we need to pass signals from the game to the flash modules, all the game signals pass through 3 registers at 27 mhz before being used by the max score module.

Sound effects (Joint)

module sfx (input clk, input apple_slice, input orange_slice, input peach_slice, input bomb_slice, input [1:0] lost_life, input busy, input [1:0] state_in, output reg sound);

This module generates the sound effects associated with certain events in the game. This is done by asserting high at certain frequencies of choice to an I/O pin which is wired to a piezo buzzer. The module is an FSM where states correspond to different sounds, Each state corresponds to a different sound, and each sound or state is only played for a certain amount of time before expiring back to an IDLE state and waiting for a new trigger. State transitions also determine play order in a logical fashion. For example, swiping a bomb is not mutually exclusive to receiving a high score, so both sounds should be played if both events occur. This can be done by adding conditions into the different states based on what can happen (i.e. if a bomb is hit and a new high score is achieved, the state transitions from BOMB to HIGH_SCORE to IDLE, instead of directly returning to IDLE after BOMB). The sounds are the following:

1. Fruit slice: This sound plays upon slicing any fruit.

2. Lost life: This sound plays upon losing a life. It uses the `fell` output from the coordinate generator to see whether the previous number of lives is unequal to the current number of lives (naturally, if they are different, the sound will play). The sound does not play if the lives change from 0 to 3 (when the game is restarted).
3. Bomb swipe: This sound plays upon swiping a bomb. It uses the `bomb_slice` input to see whether the bomb was slashed.
4. High score: This sound plays upon achieving a high score on the “game over” screen. The module uses the busy input to transition into playing a high score sound, because busy is asserted when a new high score is being written into flash.

Challenges/Difficulties

We ended up changing our timeline substantially based on what would take the most amount of time. The graphics pipeline ended up taking as much time as suspected, but there were important design decisions that had to be made which would determine how modules ended up getting wired together and how signals had to be defined.

Parabolic motion

The initial problem with graphics that had to be decided was how to generate the coordinates for the motion of the fruit. Our initial conception was that we would use the equations for parabolic motion:

$$\text{Horizontal distance, } x = V_x t$$
$$\text{Horizontal Velocity, } V_x = V_{xo}$$
$$\text{Vertical distance, } y = V_{yo} t - \frac{1}{2} g t^2$$
$$\text{Vertical Velocity, } V_y = V_{yo} - g t$$

(TutorVista, 2018)

This would require us to multiply numbers which is expensive time-wise. This would also require pipelining given the small period of the clock the module uses. Furthermore, we planned to have a “timer” module which would allow us to have small, discrete time intervals to supply to these equations. This is tedious given that iteratively adding does the same task, along with some ternary operators that allow us to determine the direction of motion for the fruit in the game.

Slice drawing

When I (Nadia) was initially prototyping how to slice fruit for our stretch goal, I did it in a separate project (our compilation time was getting a bit high). The downwards motion after slicing the fruit was generated with the coord_generator_slice module that I had written, and the picture used was just an apple. Drawing a separate rectangular image of uniform color and having that separate from the top half of the apple is easy: draw the top half and when they separate, draw the bottom half as a uniformly-colored rectangle.

However, to use the actual bottom half of the picture, we had to use different image addresses for the top and bottom halves. We thought a quick solution would be to instantiate two ROM modules: one for drawing the top half and one for drawing the bottom half. Of course, that is quite space-inefficient, so we ended up using the ROM module once and then varying the image address based on which part of the fruit the module should be drawing based on the `slice` input.

Serial receiver

The serial receiver was modified from lab 5c because of the Bluetooth modules we used. We originally wrote the receiver similarly to lab 5c, where data collection starts at the falling edge for the first start bit and continues until a certain number of bits have been collected. However, adding Bluetooth modules meant that we needed to rewrite the receiver to terminate data collection after a certain number of bytes instead of bits, looping back to wait for a falling edge multiple times instead of just once.

The change wasn't difficult to make; it involved changing a few rules on state transitions, as well as keeping track of the number of bits and bytes stored. Another problem that we encountered, though, had to do with the sporadic nature of the BT module's packets. The data we sent was 5 bytes long; sometimes, byte 5 would be grouped more closely with byte 1 of the next pulse than byte 4 of its own. This meant that we had to decrease the amount of time waiting for data to be high before transitioning to watch for a falling edge (start bit). Otherwise, we might not have caught the start bit of byte 1 for the next pulse.

Flash memory

Flash memory was frustrating to work with. The original approach involved using our max_score module to send signals like `doread`, `dowrite`, and `reset` to the flash_manager module provided on the 6.111 website. However, we ended up using Lorenzo's flash module as an intermediary module in between max_score and flash_manager, because his Verilog worked for sure. We didn't know until after using Lorenzo's code that two reset signals needed to be asserted to wipe flash. If we were to improve the game, it would make sense to do away with Lorenzo's module and pass signals directly from max_score to flash_manager, this time asserting both reset signals instead of just one.

Potential Improvements

Double Dabble

The method we used for generating the score on the screen was "quick and dirty". A nice, methodical way of converting binary into binary-coded decimal is the double dabble algorithm. From the output from a double dabble module, one could index into certain bits and find the ones, tens, and hundreds digit (and so on).

FIFO for sound effects

One of the challenges of implementing sound effects included the situations where multiple sounds could be triggered. Because of the way sound was implemented, with states corresponding to each sound, it was easy to miss out on a sound if another was triggered. Our current fix introduces extra variables and conditions to the state transitions. To simplify the module, we could create a buffer for state transitions or sound outputs, where each event pushes another set of values onto the queue.

Rotating fruit slices

Using the CORDIC algorithm, one can calculate the matrix transformations to rotate the halves of the fruit by rapidly changing the coordinates of the different pixels.

Animated state transitions

When clicking play and replay, there could be some animations or movement on the buttons to make them look clicked. To do this, we could add an extra state in between START and PLAY that triggers on a button press and exits after completing its animation. There could also be some kind of game over animation with a flashing screen or exploding bomb, which would involve a similar fix.

New project idea

We received a lot of compliments on the text used in our game, and that inspired the idea for a potential next final project - a program where users can sketch out a new font, then save the font and use it to type. This project could use a cursor like the Fruit Ninja project to write fonts, and store the data for each character in flash. It would also involve interfacing with a keyboard, which is not something that we explored in Fruit Ninja.

Conclusions

We had a lot of fun integrating different concepts and labs from 6.111 into this final project. We were happy to adopt and modify certain parts of labs into our project (i.e. drawing with different image addresses from lab 3, creating different sounds on a buzzer from lab 4). Here is a video Lydia made to document the process. Below are some thoughts for future students:

The absolute number one thing to keep in mind about not only this project but projects in general is that communication and timing are key. We found that consistently dedicating time to the project allowed for less stress later on, and more slack when needed. Slack can be useful when a certain aspect of a project is taking longer than expected, or for implementing more interesting features, both of which are possible situations.

We, personally, had quite a good history of communication. Though for most of the project, we worked on entirely different aspects, there were a handful of times when we pair-programmed. This helped us spot each other's logical inconsistencies and actually helped us get through bugs we would not have otherwise noticed as quickly. Asking your partner for advice is a good way to ensure that he or she knows what you are doing while giving you new ways of thinking about problems. Although not all project partners will work as closely as we did, it was nice to understand each other's modules for debugging/overall familiarity with all parts of the project.

We would also recommend making the project as modular as possible. This entails making sure there are no internal bugs in a certain aspect of the project, so that it can be integrated with ease. We found that making our modules fool-proof (or as much as we could) helped a lot during integration. Additionally, we would recommend learning how to use the logic analyzer, which we used to debug the serial receiver. We also used the hex display and LEDs a lot - the hex display was especially helpful, but had a lot of timing violations (note for future students).

Finally, ask for help, and do it often. The lovely staff in 6.111 are dedicated to helping you understand what you are doing, and there are many resources to help you debug. However, the prerequisite to this is not procrastinating. Starting early and giving substantial effort at all times will mean that you will run into problems early on, which is exactly what you want. You do not want to run into major design issues or tough bugs when there is much else to be done, but this is a fact of life, and not necessarily something you don't know.

References

1. "CRC-16 Calculation." *6.111 Introduction to Digital Systems Laboratory.* October 2018. http://web.mit.edu/6.111/volume2/www/f2018/index.html
2. Rosenberg, B. (1997, October 28). Generation of pseudorandom numbers by a shift register. Retrieved from http://www.cs.miami.edu/home/burt/learning/Csc609.022/random_numbers.html
3. TutorVista. "Projectile Motion Formula." *TutorVista.com*. 2018. Retrieved from https://formulas.tutorvista.com/physics/projectile-motion-formula.html

Appendix

Game FSM

```
module fruit_ninja (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
//lab3 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;

////////////////////////////////////////////////////////////////////////////
//
// fruit ninja
//
////////////////////////////////////////////////////////////////////////////

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

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

// ENTER button is user reset
wire reset,user_reset;
                                                                                                  debounce
db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(user_reset));
   assign reset = user_reset | power_on_reset;

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

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

reg data1,data2, data3;
wire game_state;

// feed XVGA signals to fruit ninja game
wire [23:0] pixel;

// scorekeeping
wire [7:0] ultimate_score;
wire [15:0] score;

// Sounds
wire [1:0] fell;
wire apple_slice, orange_slice, peach_slice, bomb_slice;
wire [1:0] state;
game_fsm fruit_ninja(.resetbutton(~button_enter), .state_out(state), .vsync(vsync),
.serial_data(data3), .vclock(clock_65mhz),.reset(reset), .ultimate_score(ultimate_score),
.hcount(hcount), .vcount(vcount) ,.pixel(pixel), .game_state(game_state), .highscore(score[7:0]),
.fell_out(fell), .apple_slice(apple_slice), .orange_slice(orange_slice),
.peach_slice(peach_slice), .bomb_slice(bomb_slice));

reg [23:0] rgb;
reg b,hs,vs;
always @(posedge clock_65mhz) begin
       data1 <= user2[0];
       data2 <= data1;
       data3 <= data2;
       hs <= hsync;
       vs <= vsync;
       b <= blank;
       rgb <= pixel;
end

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

wire [639:0] dots;
wire writemode;
wire [15:0] wdata;
wire dowrite;
wire [22:0] raddr;
wire [15:0] frdata;
wire doread;
wire busy;
wire [11:0] fsmstate;

wire [4:0] state_out;
wire [15:0] score_to_store;
wire writing;
wire reading;
wire flash_reset;
wire up_reset;

wire ready;
wire [15:0] current_score;

flash_manager flash(.clock(clock_27mhz), .reset(flash_reset), .dots(dots), .writemode(writemode),
.wdata(wdata), .dowrite(dowrite), .raddr(raddr), .frdata(frdata), .doread(doread), .busy(busy),
.flash_data(flash_data), .flash_address(flash_address),
.flash_ce_b(flash_ce_b),.flash_oe_b(flash_oe_b), .flash_we_b(flash_we_b),
.flash_reset_b(flash_reset_b), .flash_sts(flash_sts), .flash_byte_b(flash_byte_b),
.fsmstate(fsmstate));

flasher my_flash (.clk(clock_27mhz), .busy(busy), .fsmstate(fsmstate), .dots(dots),
.writing(writing),.reading(reading), .wdata(wdata), .writemode(writemode), .dowrite(dowrite),
.raddr(raddr), .doread(doread), .score_to_store(score_to_store), .up(up_reset));

slower slow(.clk(clock_27mhz), .r_in(game_state), .c_in(ultimate_score), .r(ready),
.c(current_score));
```

```
max_score my_max (.current_score(current_score), .score_from_flash(frdata), .clk(clock_27mhz),
.ready(ready), .busy(busy), .reset(flash_reset), .writing(writing), .reading(reading),
.up_reset(up_reset),.score(score), .score_to_store(score_to_store), .state_out(state_out),
.reset_score(!button_up));

wire r;
sfx sounds(.state_in(state), .clk(clock_65mhz), .apple_slice(apple_slice),
.orange_slice(orange_slice), .peach_slice(peach_slice), .bomb_slice(bomb_slice),
.busy(busy), .sound(user4[0]), .lost_life(fell), .r(r));

assign led[7] = ~r;
assign led[6] = ~busy;
assign led[5] = ~game_state;
assign led[4:0] = 5'b11111;

endmodule

module slower(input clk,
       input r_in,
       input [6:0] c_in,
       output reg r,
       output reg [6:0] c);

       reg a, b, d;
       reg [15:0] one;
       reg [15:0] two;
       reg [15:0] three;
       always @ (posedge clk) begin
              a <= r_in;
              b <= a;
              d <= b;
              one <= c_in;
              two <= one;
              three <= two;
              r <= d;
              c <= three;
       end
endmodule

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

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

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

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

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;

   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

////////////////////////////////////////////////////////////////////////////////
//
// game_fsm: the main game FSM
//
////////////////////////////////////////////////////////////////////////////////

module game_fsm (
       input [7:0] highscore,
       input vclock,         // 65MHz clock
       input vsync,
       input serial_data,
       input reset,          // 1 to initialize module
       input [10:0] hcount,  // horizontal index of current pixel (0..1023)
       input [9:0] vcount,   // vertical index of current pixel (0..767)
       output [23:0] pixel,  // game's pixel  // r=23:16, g=15:8, b=7:0
       output [7:0] ultimate_score,
       output game_state,
       output apple_slice,
       output orange_slice,
       output peach_slice,
       output bomb_slice,
       output [1:0] fell_out,
       output [1:0] state_out,
       input resetbutton);

       wire [15:0] yeet; wire data;
       // apple wires
       wire [4:0] yvel; wire [9:0] xcostart; wire backwards;
       wire [23:0] apple_pixel; wire [9:0] apple_x; wire [9:0] apple_y;

       // orange wires
       wire [4:0] yvel1; wire [9:0] xcostart1; wire backwards1;
       wire [23:0] orange_pixel; wire [9:0] orange_x; wire [9:0] orange_y;

       // bomb wires
```

```
wire [4:0] yvel2; wire [9:0] xcostart2; wire backwards2;
wire [23:0] bomb_pixel; wire [9:0] bomb_x; wire [9:0] bomb_y;

// peach wires
wire [4:0] yvel3; wire [9:0] xcostart3; wire backwards3;
wire [23:0] peach_pixel; wire [9:0] peach_x; wire [9:0] peach_y;

// other stuff lol
wire [23:0] game_pixel;
wire overlap;

reg [1:0] state = 0;
parameter START = 0;
parameter PLAY = 1;
parameter GAME_OVER = 2;

assign state_out = state;

// Random number generation

randombitsgenerator randos (.clk(vsync), .data(data), .randomnumber(yeet));

lookuptable appletable (.clk(vsync), .randomnumber(yeet[14:12]), .yvel1(yvel),
.xcostart1(xcostart), .backwards1(backwards));
lookuptable orangetable (.clk(vsync), .randomnumber(yeet[2:0]), .yvel1(yvel1),
.xcostart1(xcostart1), .backwards1(backwards1));
lookuptablebomb bombtable (.clk(vsync), .randomnumber(yeet[8:6]), .yvel1(yvel2),
.xcostart1(xcostart2), .backwards1(backwards2));
lookuptable peachtable (.clk(vsync), .randomnumber(yeet[3:1]), .yvel1(yvel3),
.xcostart1(xcostart3), .backwards1(backwards3));

// Coordinates
reg ready;

wire [2:0] apple_fell; wire active_apple; wire [7:0] apple_score; wire applenew; wire
applesliced; wire [4:0] yvelocity_apple;
coord_generator apple_coords(.yvelocity(yvelocity_apple), .new(applenew),
.score(apple_score), .active(yeet[0]), .activeconst(active_apple),  .rdy(ready),
.slice(applesliced), .vsync(vsync), .yvel(yvel), .xcostart(xcostart),
.backwards(backwards), .x_coord(apple_x), .y_coord(apple_y), .fell(apple_fell));

wire [2:0] orange_fell; wire active_orange; wire [7:0] orange_score; wire orangenew; wire
orangesliced; wire [4:0] yvelocity_orange;
coord_generator orange_coords(.yvelocity(yvelocity_orange), .new(orangenew),
.score(orange_score), .active(yeet[1]), .activeconst(active_orange),
.rdy(ready), .slice(orangesliced), .vsync(vsync), .yvel(yvel1), .xcostart(xcostart1),
.backwards(backwards1), .x_coord(orange_x), .y_coord(orange_y), .fell(orange_fell));

wire [2:0] bomb_fell; wire active_bomb; wire [3:0] bomb_score;
coord_generator bomb_coords(.score(bomb_score), .active(yeet[2]),
.activeconst(active_bomb), .rdy(ready), .slice(0), .vsync(vsync), .yvel(yvel2),
.xcostart(xcostart2), .backwards(backwards2), .x_coord(bomb_x), .y_coord(bomb_y),
.fell(bomb_fell));

wire [2:0] peach_fell; wire active_peach; wire [7:0] peach_score; wire peachnew; wire
peachsliced; wire [4:0] yvelocity_peach;
coord_generator peach_coords(.yvelocity(yvelocity_peach), .new(peachnew),
.score(peach_score), .active(yeet[5]), .activeconst(active_peach), .rdy(ready),
.slice(peachsliced), .vsync(vsync), .yvel(yvel3), .xcostart(xcostart3),
.backwards(backwards3), .x_coord(peach_x), .y_coord(peach_y), .fell(peach_fell));
```

```
assign ultimate_score = apple_score + orange_score + peach_score;

// Fruit & bombs
wire [9:0] apple_x2, peach_x2, orange_x2;
wire [9:0] orange_y2, apple_y2, peach_y2;

apple appley(.active(active_apple), .slice(applesliced), .pixel_clk(vclock),
.hcount(hcount), .xslice(apple_x2), .yslice(apple_y2), .vcount(vcount), .x(apple_x),
.y(apple_y), .pixel(apple_pixel));

orange orangey(.active(active_orange), .slice(orangesliced), .pixel_clk(vclock),
.hcount(hcount), .vcount(vcount), .x(orange_x), .y(orange_y), .xslice(orange_x2),
.yslice(orange_y2), .pixel(orange_pixel));

bomb bomby(.active(active_bomb), .slice(0), .pixel_clk(vclock), .hcount(hcount),
.vcount(vcount), .x(bomb_x), .y(bomb_y), .pixel(bomb_pixel));

peach peachy(.active(active_peach), .slice(peachsliced), .pixel_clk(vclock),
.hcount(hcount), .vcount(vcount), .x(peach_x), .y(peach_y), .xslice(peach_x2),
.yslice(peach_y2), .pixel(peach_pixel));

// Flying slices
coord_generator_slice apple_slice_coords (.yvel(0), .begincalc(applesliced),
.vsync(vsync), .new(applenew), .backwards(~backwards),
.xcostart(apple_x), .ycostart(apple_y + 75), .x_coord(apple_x2), .y_coord(apple_y2));

coord_generator_slice orange_slice_coords (.yvel(0), .begincalc(orangesliced),
.vsync(vsync), .new(orangenew), .backwards(~backwards1),
.xcostart(orange_x), .ycostart(orange_y + 75), .x_coord(orange_x2), .y_coord(orange_y2));

coord_generator_slice peach_slice_coords (.yvel(0), .begincalc(peachsliced),
.vsync(vsync), .new(peachnew), .backwards(~backwards3),
.xcostart(peach_x), .ycostart(peach_y + 75), .x_coord(peach_x2), .y_coord(peach_y2));

assign apple_slice = applesliced && (state == PLAY);
assign peach_slice = peachsliced && (state == PLAY);
assign orange_slice = orangesliced && (state == PLAY);
assign fell_out = apple_fell + orange_fell + peach_fell;

// Lives

wire [23:0] pixel_life1; reg display_life1;
blob life1 (.color(24'hFF_FF_FF), .x(100), .y(100), .clk(vclock), .hcount(hcount),
.vcount(vcount), .display(display_life1), .pixel(pixel_life1));
wire [23:0] pixel_life2; reg display_life2;
blob life2 (.color(24'hFF_FF_FF), .x(170), .y(100), .clk(vclock),.hcount(hcount),
.vcount(vcount), .display(display_life2), .pixel(pixel_life2));
wire [23:0] pixel_life3; reg display_life3;
blob life3 (.color(24'hFF_FF_FF), .x(240), .y(100), .clk(vclock),.hcount(hcount),
.vcount(vcount), .display(display_life3), .pixel(pixel_life3));

// Cursor
wire [23:0] cursor_pixel;
wire [15:0] cursor_x;
wire [15:0] cursor_y;
wire [7:0] button;

cursor_coords generator (.serial_data(serial_data), .clk(vclock), .x_coord(cursor_x),
.y_coord(cursor_y), .button(button));
```

```
blob #(.WIDTH(10), .HEIGHT(10)) cursor (.color(24'hFF_FF_00), .clk(vclock),
.x(cursor_x[10:0]), .y(cursor_y[9:0]), .hcount(hcount), .vcount(vcount), .display(1),
.pixel(cursor_pixel));

// Slice management

slice_dealer slicey (.peachactive(active_peach), .appleactive(active_apple),
.orangeactive(active_orange), .clk(vclock), .apple_new(applenew), .orange_new(orangenew),
.peach_new(peachnew), .applepix(apple_pixel), .cursorpix(cursor_pixel),
.orangepix(orange_pixel), .peachpix(peach_pixel), .applesliced(applesliced),
.orangesliced(orangesliced), .peachsliced(peachsliced));

// Background

wire [23:0] bg_pixel;
bg background(.vsync(vsync), .clk(vclock), .pixel(bg_pixel));

// Start and end screens

wire [23:0] start_pixel;
wire [23:0] logo_pixel;
wire [23:0] play_pixel;

logo_blob logo_picture (.pixel_clk(vclock), .x(362), .y(200), .hcount(hcount),
.vcount(vcount), .pixel(logo_pixel));

play_blob play_text(.pixel_clk(vclock), .x(424), .y(400), .hcount(hcount),
.vcount(vcount), .pixel(play_pixel));

assign start_pixel = play_pixel | logo_pixel;

wire [23:0] end_pixel;
wire [23:0] replay_pixel;
wire [23:0] score_pixel;
wire [23:0] high_score_pixel;

replay_blob replay_button(.pixel_clk(vclock), .x(297), .y(244), .hcount(hcount),
.vcount(vcount), .pixel(replay_pixel));

score_blob score_text (.pixel_clk(vclock), .x(297), .y(350), .hcount(hcount),
.vcount(vcount), .pixel(score_pixel));

hi_blob high_score (.pixel_clk(vclock), .x(297), .y(425), .hcount(hcount),
.vcount(vcount), .pixel(high_score_pixel));

// number scores
wire [23:0] top_left_1; reg display_1; reg [9:0] x1; reg [8:0] y1;
blob #(.WIDTH(10), .HEIGHT(44)) s_10(.clk(vclock), .display(display_1),
.color(24'hFF_FF_FF), .x(x1), .y(y1), .hcount(hcount), .vcount(vcount),
.pixel(top_left_1));
wire [23:0] top_1; reg display_2; reg [9:0] x2; reg [8:0] y2;
blob #(.WIDTH(44), .HEIGHT(10)) s_11(.clk(vclock), .display(display_2),
.color(24'hFF_FF_FF), .x(x2), .y(y2), .hcount(hcount), .vcount(vcount), .pixel(top_1));
wire [23:0] top_right_1; reg display_3; reg [9:0] x3; reg [8:0] y3;
blob #(.WIDTH(10), .HEIGHT(44)) s_12(.clk(vclock), .display(display_3),
.color(24'hFF_FF_FF), .x(x3), .y(y3), .hcount(hcount), .vcount(vcount),
.pixel(top_right_1));
wire [23:0] bottom_left_1; reg display_4; reg [9:0] x4; reg [8:0] y4;
blob #(.WIDTH(10), .HEIGHT(44)) s_13(.clk(vclock), .display(display_4),
.color(24'hFF_FF_FF), .x(x4), .y(y4), .hcount(hcount), .vcount(vcount),
.pixel(bottom_left_1));
wire [23:0] bottom_right_1; reg display_5; reg [9:0] x5; reg [8:0] y5;
blob #(.WIDTH(10), .HEIGHT(54)) s_14(.clk(vclock), .display(display_5),
.color(24'hFF_FF_FF), .x(x5), .y(y5), .hcount(hcount), .vcount(vcount),
.pixel(bottom_right_1));
wire [23:0] middle_1; reg display_6; reg [9:0] x6; reg [8:0] y6;
blob #(.WIDTH(44), .HEIGHT(10)) s_15(.clk(vclock), .display(display_6),
.color(24'hFF_FF_FF), .x(x6), .y(y6), .hcount(hcount), .vcount(vcount), .pixel(middle_1));
wire [23:0] bottom_1; reg display_7; reg [9:0] x7; reg [8:0] y7;
blob #(.WIDTH(44), .HEIGHT(10)) s_16(.clk(vclock), .display(display_7),
.color(24'hFF_FF_FF), .x(x7), .y(y7), .hcount(hcount), .vcount(vcount), .pixel(bottom_1));
wire [23:0] top_left_2; reg display_8; reg [9:0] x8; reg [8:0] y8;
blob #(.WIDTH(10), .HEIGHT(44)) s_20(.clk(vclock),.display(display_8),
.color(24'hFF_FF_FF), .x(x8), .y(y8), .hcount(hcount), .vcount(vcount),
.pixel(top_left_2));
wire [23:0] top_2; reg display_9; reg [9:0] x9; reg [8:0] y9;
blob #(.WIDTH(44), .HEIGHT(10)) s_21(.clk(vclock),.display(display_9),
.color(24'hFF_FF_FF), .x(x9), .y(y9), .hcount(hcount), .vcount(vcount), .pixel(top_2));
wire [23:0] top_right_2; reg display_10; reg [9:0] x10; reg [8:0] y10;
blob #(.WIDTH(10), .HEIGHT(44)) s_22(.clk(vclock),.display(display_10),
.color(24'hFF_FF_FF), .x(x10), .y(y10), .hcount(hcount), .vcount(vcount),
.pixel(top_right_2));
wire [23:0] bottom_left_2; reg display_11; reg [9:0] x11; reg [8:0] y11;
blob #(.WIDTH(10), .HEIGHT(44)) s_23(.clk(vclock),.display(display_11),
.color(24'hFF_FF_FF), .x(x11), .y(y11), .hcount(hcount), .vcount(vcount),
.pixel(bottom_left_2));
wire [23:0] bottom_right_2; reg display_12; reg [9:0] x12; reg [8:0] y12;
blob #(.WIDTH(10), .HEIGHT(54)) s_24(.clk(vclock),.display(display_12),
.color(24'hFF_FF_FF), .x(x12), .y(y12), .hcount(hcount), .vcount(vcount),
.pixel(bottom_right_2));
wire [23:0] middle_2; reg display_13; reg [9:0] x13; reg [8:0] y13;
blob #(.WIDTH(44), .HEIGHT(10)) s_25(.clk(vclock), .display(display_13),
.color(24'hFF_FF_FF), .x(x13), .y(y13), .hcount(hcount), .vcount(vcount),
.pixel(middle_2));
wire [23:0] bottom_2; reg display_14; reg [9:0] x14; reg [8:0] y14;
blob #(.WIDTH(44), .HEIGHT(10)) s_26(.clk(vclock), .display(display_14),
.color(24'hFF_FF_FF), .x(x14), .y(y14), .hcount(hcount), .vcount(vcount),
.pixel(bottom_2));

// high score instantiations
wire [23:0] top_left_3; reg display_15;
blob #(.WIDTH(10), .HEIGHT(44)) s_30(.clk(vclock), .display(display_15),
.color(24'hFF_FF_FF), .x(x1+ 60 + 65), .y(y1 + 78), .hcount(hcount), .vcount(vcount),
.pixel(top_left_3));
wire [23:0] top_3; reg display_16;
blob #(.WIDTH(44), .HEIGHT(10)) s_31(.clk(vclock), .display(display_16),
.color(24'hFF_FF_FF), .x(x2 + 60 + 65), .y(y2 + 78), .hcount(hcount), .vcount(vcount),
.pixel(top_3));
wire [23:0] top_right_3; reg display_17;
blob #(.WIDTH(10), .HEIGHT(44)) s_32(.clk(vclock), .display(display_17),
.color(24'hFF_FF_FF), .x(x3 + 60 + 65), .y(y3 + 78), .hcount(hcount), .vcount(vcount),
.pixel(top_right_3));
wire [23:0] bottom_left_3; reg display_18;
blob #(.WIDTH(10), .HEIGHT(44)) s_33(.clk(vclock), .display(display_18),
.color(24'hFF_FF_FF), .x(x4 + 60 + 65), .y(y4 + 78), .hcount(hcount), .vcount(vcount),
.pixel(bottom_left_3));
wire [23:0] bottom_right_3; reg display_19;
blob #(.WIDTH(10), .HEIGHT(54)) s_34(.clk(vclock), .display(display_19),
.color(24'hFF_FF_FF), .x(x5+ 60 + 65), .y(y5 + 78), .hcount(hcount), .vcount(vcount),
.pixel(bottom_right_3));
wire [23:0] middle_3; reg display_20;
blob #(.WIDTH(44), .HEIGHT(10)) s_35(.clk(vclock), .display(display_20),
.color(24'hFF_FF_FF), .x(x6 + 60 + 65), .y(y6 + 78), .hcount(hcount), .vcount(vcount),
.pixel(middle_3));
wire [23:0] bottom_3; reg display_21;
blob #(.WIDTH(44), .HEIGHT(10)) s_36(.clk(vclock), .display(display_21),
.color(24'hFF_FF_FF), .x(x7 + 60 + 65), .y(y7+ 78), .hcount(hcount), .vcount(vcount),
.pixel(bottom_3));
wire [23:0] top_left_4; reg display_22;
blob #(.WIDTH(10), .HEIGHT(44)) s_40(.clk(vclock),.display(display_22),
.color(24'hFF_FF_FF), .x(x8+ 60 + 65), .y(y8 + 78), .hcount(hcount), .vcount(vcount),
.pixel(top_left_4));
wire [23:0] top_4; reg display_23;
blob #(.WIDTH(44), .HEIGHT(10)) s_41(.clk(vclock),.display(display_23),
.color(24'hFF_FF_FF), .x(x9 + 60 + 65), .y(y9 + 78), .hcount(hcount), .vcount(vcount),
.pixel(top_4));
wire [23:0] top_right_4; reg display_24;
blob #(.WIDTH(10), .HEIGHT(44)) s_42(.clk(vclock),.display(display_24),
.color(24'hFF_FF_FF), .x(x10 + 60 + 65), .y(y10 + 78), .hcount(hcount), .vcount(vcount),
.pixel(top_right_4));
wire [23:0] bottom_left_4; reg display_25;
blob #(.WIDTH(10), .HEIGHT(44)) s_43(.clk(vclock),.display(display_25),
.color(24'hFF_FF_FF), .x(x11 + 60 + 65), .y(y11 + 78), .hcount(hcount), .vcount(vcount),
.pixel(bottom_left_4));
wire [23:0] bottom_right_4; reg display_26;
blob #(.WIDTH(10), .HEIGHT(54)) s_44(.clk(vclock),.display(display_26),
.color(24'hFF_FF_FF), .x(x12 + 60 + 65), .y(y12 + 78), .hcount(hcount), .vcount(vcount),
.pixel(bottom_right_4));
wire [23:0] middle_4; reg display_27;
blob #(.WIDTH(44), .HEIGHT(10)) s_45(.clk(vclock), .display(display_27),
.color(24'hFF_FF_FF), .x(x13 + 60 + 65), .y(y13 + 78), .hcount(hcount), .vcount(vcount),
.pixel(middle_4));
wire [23:0] bottom_4; reg display_28;
blob #(.WIDTH(44), .HEIGHT(10)) s_46(.clk(vclock), .display(display_28),
.color(24'hFF_FF_FF), .x(x14 + 60 + 65), .y(y14 + 78), .hcount(hcount), .vcount(vcount),
.pixel(bottom_4));

reg [7:0] scoreboard; reg [7:0] hiscore;
reg [7:0] subtractor; reg [7:0] subtractorhi;

wire [23:0] score; wire [23:0] highestscore;
assign score = top_left_1 | top_1 | top_right_1 | bottom_left_1 | bottom_right_1 |
middle_1 | bottom_1 |top_left_2 | top_2 | top_right_2 | bottom_left_2 | bottom_right_2 |
middle_2 | bottom_2;

assign highestscore = top_left_3 | top_3 | top_right_3 | bottom_left_3 | bottom_right_3 |
middle_3 | bottom_3 |top_left_4 | top_4 | top_right_4 | bottom_left_4 | bottom_right_4 |
middle_4 | bottom_4;

assign end_pixel = score_pixel | score | replay_pixel | high_score_pixel | highestscore;

wire play_overlap = cursor_x > 424 && cursor_x < 589 && cursor_y > 400 && cursor_y < 442;

wire replay_overlap = cursor_x > 297 && cursor_x < 718 && cursor_y > 244 && cursor_y <
314;
// Game State Machine

reg old_up; reg oldresetbutton;
```

```
always @(posedge vclock) begin

       hiscore <= highscore;
       scoreboard <= ultimate_score;

       if (scoreboard < 10) subtractor <= 0; else if (scoreboard >= 10 && scoreboard <
20)
              subtractor <= 10;
       else if (scoreboard >= 20 && scoreboard < 30) subtractor <= 20;
       else if (scoreboard >= 30 && scoreboard < 40) subtractor <= 30;
       else if (scoreboard >= 40 && scoreboard < 50) subtractor <= 40;
       else if (scoreboard >= 50 && scoreboard < 60) subtractor <= 50;
       else if (scoreboard >= 60 && scoreboard < 70) subtractor <= 60;
       else if (scoreboard >= 70 && scoreboard < 80) subtractor <= 70;
       else if (scoreboard >= 80 && scoreboard < 90) subtractor <= 80;
       else if (scoreboard >= 90 && scoreboard < 150) subtractor <= 90;

       case (scoreboard - subtractor)
       0: begin display_6 <= 0; display_1 <= 1; display_2 <= 1; display_3 <= 1; display_4
<= 1; display_5 <= 1; display_7 <= 1; end
       1: begin display_6 <= 0; display_1 <= 0; display_2 <= 0; display_3 <= 1; display_4
<= 0; display_5 <= 1; display_7 <= 0; end
       2: begin display_6 <= 1; display_1 <= 0; display_2 <= 1; display_3 <= 1; display_4
<= 1; display_5 <= 0; display_7 <= 1; end
       3: begin display_6 <= 1; display_1 <= 0; display_2 <= 1; display_3 <= 1; display_4
<= 0; display_5 <= 1; display_7 <= 1; end
       4: begin display_6 <= 1; display_1 <= 1; display_2 <= 0; display_3 <= 1; display_4
<= 0; display_5 <= 1; display_7 <= 0; end
       5: begin display_6 <= 1; display_1 <= 1; display_2 <= 1; display_3 <= 0; display_4
<= 0; display_5 <= 1; display_7 <= 1; end
       6: begin display_6 <= 1; display_1 <= 1; display_2 <= 1; display_3 <= 0; display_4
<= 1; display_5 <= 1; display_7 <= 1; end
       7: begin display_6 <= 0; display_1 <= 1; display_2 <= 1; display_3 <= 1; display_4
<= 0; display_5 <= 1; display_7 <= 0; end
       8: begin display_6 <= 1; display_1 <= 1; display_2 <= 1; display_3 <= 1; display_4
<= 1; display_5 <= 1; display_7 <= 1; end
       9: begin display_6 <= 1; display_1 <= 1; display_2 <= 1; display_3 <= 1; display_4
<= 0; display_5 <= 1; display_7 <= 0; end
       default: begin display_6 <= 1; display_1 <= 1; display_2 <= 1; display_3 <= 1;
display_4 <= 0; display_5 <= 1; display_7 <= 0; end
       endcase

       if(scoreboard < 10) begin
       display_13 <= 0; display_8 <= 1; display_9 <= 1; display_10 <= 1; display_11 <= 1;
display_12 <= 1; display_14 <= 1; end
              else if (scoreboard >= 10 && scoreboard < 20) begin display_13 <= 0;
display_8 <= 0; display_9 <= 0; display_10 <= 1; display_11 <= 0; display_12 <= 1;
display_14 <= 0; end
       else if (scoreboard >= 20 && scoreboard < 30) begin display_13 <= 1; display_8 <=
0; display_9 <= 1; display_10 <= 1; display_11 <= 1; display_12 <= 0; display_14 <= 1; end
       else if (scoreboard >= 30 && scoreboard < 40) begin display_13 <= 1; display_8 <=
0; display_9 <= 1; display_10 <= 1; display_11 <= 0; display_12 <= 1; display_14 <= 1; end
       else if (scoreboard >= 40 && scoreboard < 50) begin display_13 <= 1; display_8 <=
1; display_9 <= 0; display_10 <= 1; display_11 <= 0; display_12 <= 1; display_14 <= 0; end
       else if (scoreboard >= 50 && scoreboard < 60) begin display_13 <= 1; display_8 <=
1; display_9 <= 1; display_10 <= 0; display_11 <= 0; display_12 <= 1; display_14 <= 1; end
       else if (scoreboard >= 60 && scoreboard < 70)begin display_13 <= 1; display_8 <=
1; display_9 <= 1; display_10 <= 0; display_11 <= 1; display_12 <= 1; display_14 <= 1; end
       else if (scoreboard >= 70 && scoreboard < 80) begin display_13 <= 0; display_8 <=
1; display_9 <= 1; display_10 <= 1; display_11 <= 0; display_12 <= 1; display_14 <= 0; end
```

```
        else if (scoreboard >= 80 && scoreboard < 90) begin display_13 <= 1; display_8 <=
1; display_9 <= 1; display_10 <= 1; display_11 <= 1; display_12 <= 1; display_14 <= 1; end
        else if (scoreboard >= 90 && scoreboard < 150) begin display_13 <= 1; display_8 <=
1; display_9 <= 1; display_10 <= 1; display_11 <= 0; display_12 <= 1; display_14 <= 0; end

        if (hiscore < 10) subtractorhi <= 0; else if (hiscore >= 10 && hiscore < 20)
subtractorhi <= 10; else if (hiscore>= 20 && hiscore < 30) subtractorhi <= 20;
        else if (hiscore >= 30 && hiscore < 40) subtractorhi <= 30; else if (hiscore >= 40
&& hiscore < 50) subtractorhi <= 40;
        else if (hiscore >= 50 && hiscore < 60) subtractorhi <= 50; else if (hiscore >= 60
&& hiscore < 70) subtractorhi <= 60;
        else if (hiscore >= 70 && hiscore < 80) subtractorhi <= 70; else if (hiscore >= 80
&& hiscore < 90) subtractorhi <= 80;
        else if (hiscore >= 90 && hiscore < 150) subtractorhi <= 90;

        case (hiscore - subtractorhi)
                0: begin display_20 <= 0; display_15 <= 1; display_16 <= 1; display_17 <=
1; display_18 <= 1; display_19 <= 1; display_21 <= 1; end
                1: begin display_20 <= 0; display_15 <= 0; display_16 <= 0; display_17 <=
1; display_18 <= 0; display_19 <= 1; display_21 <= 0; end
                2: begin display_20 <= 1; display_15 <= 0; display_16 <= 1; display_17 <=
1; display_18 <= 1; display_19<= 0; display_21 <= 1; end
                3: begin display_20 <= 1; display_15 <= 0; display_16 <= 1; display_17 <=
1; display_18 <= 0; display_19 <= 1; display_21 <= 1; end
                4: begin display_20 <= 1; display_15 <= 1; display_16 <= 0; display_17 <=
1; display_18 <= 0; display_19 <= 1; display_21 <= 0; end
                5: begin display_20 <= 1; display_15 <= 1; display_16 <= 1; display_17 <=
0; display_18 <= 0; display_19 <= 1; display_21 <= 1; end
                6: begin display_20 <= 1; display_15 <= 1; display_16 <= 1; display_17 <=
0; display_18 <= 1; display_19 <= 1; display_21 <= 1; end
                7: begin display_20 <= 0; display_15 <= 1; display_16 <= 1; display_17 <=
1; display_18 <= 0; display_19 <= 1; display_21 <= 0; end
                8: begin display_20 <= 1; display_15 <= 1; display_16 <= 1; display_17 <=
1; display_18 <= 1; display_19 <= 1; display_21 <= 1; end
                9: begin display_20 <= 1; display_15 <= 1; display_16 <= 1; display_17 <=
1; display_18 <= 0; display_19 <= 1; display_21<= 0; end
                default: begin display_20 <= 1; display_15 <= 1; display_16 <= 1;
display_17 <= 1; display_18 <= 0; display_19 <= 1; display_21 <= 0; end
        endcase

        if(hiscore < 10) begin display_27 <= 0; display_22 <= 1; display_23 <= 1;
display_24 <= 1; display_25 <= 1; display_26 <= 1; display_28 <= 1; end
        else if (hiscore >= 10 && hiscore < 20) begin display_27 <= 0; display_22 <= 0;
display_23 <= 0; display_24 <= 1; display_25 <= 0; display_26 <= 1; display_28 <= 0; end
        else if (hiscore >= 20 && hiscore < 30) begin display_27 <= 1; display_22 <= 0;
display_23 <= 1; display_24 <= 1; display_25 <= 1; display_26 <= 0; display_28 <= 1; end
        else if (hiscore >= 30 && hiscore < 40) begin display_27 <= 1; display_22 <= 0;
display_23 <= 1; display_24 <= 1; display_25 <= 0; display_26 <= 1; display_28 <= 1; end
        else if (hiscore >= 40 && hiscore < 50) begin display_27 <= 1; display_22 <= 1;
display_23 <= 0; display_24 <= 1; display_25 <= 0; display_26 <= 1; display_28 <= 0; end
        else if (hiscore >= 50 && hiscore < 60) begin display_27 <= 1; display_22 <= 1;
display_23 <= 1; display_24 <= 0; display_25 <= 0; display_26 <= 1; display_28 <= 1; end
        else if (hiscore>= 60 && hiscore < 70)begin display_27 <= 1; display_22 <= 1;
display_23 <= 1; display_24 <= 0; display_25 <= 1; display_26 <= 1; display_28 <= 1; end
        else if (hiscore >= 70 && hiscore < 80) begin display_27 <= 0; display_22 <= 1;
display_23 <= 1; display_24 <= 1; display_25 <= 0; display_26 <= 1; display_28 <= 0; end
        else if (hiscore >= 80 && hiscore < 90) begin display_27 <= 1; display_22 <= 1;
display_23 <= 1; display_24 <= 1; display_25 <= 1; display_26 <= 1; display_28 <= 1; end
        else if (hiscore >= 90 && hiscore < 150) begin display_27 <= 1; display_22 <= 1;
display_23 <= 1; display_24 <= 1; display_25 <= 0; display_26 <= 1; display_28 <= 0; end
```

```
old_up <= button[0];
oldresetbutton <= resetbutton;

case (state)
	START: if (button[0] && ~old_up && play_overlap) begin
		state <= PLAY;
		display_life3 <= 1;
		display_life2 <= 1;
		display_life1 <= 1;
		end
		else begin
			state <= state;
			ready <= 0;
		end
	PLAY: if (apple_fell + orange_fell + peach_fell>= 3 || ((|cursor_pixel &&
	|bomb_pixel) && active_bomb)) begin
		ready <= 0;
		state <= GAME_OVER; end
		else if (resetbutton && ~oldresetbutton)
			state <= START;
		else begin
			x1 <= 900; y1 <= 100; x2 <= 900; y2 <= 100; x3 <= 934; y3 <= 100; x4
	<= 900; y4 <= 134; x5 <= 934; y5 <= 134; x6 <= 900; y6 <= 134; x7 <= 900; y7 <=
	178;x8 <= 850; y8 <= 100; x9 <= 850; y9 <= 100; x10 <= 884; y10 <= 100; x11 <=
	850; y11 <= 134; x12 <= 884; y12 <= 134; x13 <= 850; y13 <= 134; x14 <= 850; y14
	<= 178;
	case (apple_fell + orange_fell + peach_fell)
		0: begin display_life3 <= 1; display_life2 <= 1; display_life1 <= 1; end
		1: begin display_life3 <= 0; display_life2 <= 1; display_life1 <= 1; end
		2: begin display_life3 <= 0; display_life2 <= 0; display_life1 <= 1; end
		3: begin display_life3 <= 0; display_life2 <= 0; display_life1 <= 0; end
		default : begin
		display_life3 <= 1; display_life2 <= 1; display_life1 <= 1;
		end
	endcase
	ready <= 1;
	state <= state;
	end
	GAME_OVER: if ((button[0] && ~old_up && replay_overlap)) begin
	state <= PLAY;
		end
		else if (resetbutton && ~oldresetbutton) state <= START;
			else begin
				x1 <= 572; y1 <= 325; x2 <= 572; y2 <= 325; x3 <= 606; y3 <=
325; x4 <= 572; y4 <= 359; x5 <= 606; y5 <= 359; x6 <= 572; y6 <= 359; x7 <= 572; y7 <=
403; x8 <= 512; y8 <= 325; x9 <= 512; y9 <= 325; x10 <= 546; y10 <= 325; x11 <= 512; y11
<= 359; x12 <= 546; y12 <= 359; x13 <= 512; y13 <= 359; x14 <= 512; y14 <= 403;
				state <= state;
				ready <= 0;
			end
	default: state <= START;
endcase
end

wire [23:0] fruit_pixel;
wire bomb_first;
wire apple_first;
wire peach_first;
```

```
        assign bomb_first = ((|bomb_pixel) && (|orange_pixel)) || ((|bomb_pixel) &&
        (|apple_pixel)) || ((|bomb_pixel) && (|peach_pixel));
        assign apple_first = (|apple_pixel) && (|orange_pixel) || (|apple_pixel) &&
        (|peach_pixel);
        assign peach_first = (|peach_pixel) && (|orange_pixel);

        assign fruit_pixel = bomb_first ? bomb_pixel : apple_first ? apple_pixel : peach_first?
        peach_pixel: apple_pixel | orange_pixel | bomb_pixel | peach_pixel;
        assign game_pixel = ((state == PLAY) ? fruit_pixel | pixel_life1 | pixel_life2 |
        pixel_life3 | cursor_pixel | score : (state == START) ? start_pixel | cursor_pixel :
        end_pixel | cursor_pixel);
        assign overlap = ((|game_pixel) && (|bg_pixel));
        assign pixel = overlap ? game_pixel : bg_pixel;

        assign game_state = (state == GAME_OVER);

        assign bomb_slice = (|cursor_pixel && |bomb_pixel) && active_bomb && (state == PLAY);
endmodule
```

Lookup Tables

```
module lookuptable (input clk, input [2:0] randomnumber,
                output reg [4:0] yvel1, output reg backwards1, output reg [9:0] xcostart1);

        always@(posedge clk) begin
                case (randomnumber)
                3'b00: begin yvel1 <= 16; xcostart1 <= 100; backwards1 <= 1; end
                3'b01: begin yvel1 <= 12; xcostart1 <= 200; backwards1 <= 0; end
                3'b10: begin yvel1 <= 16; xcostart1 <= 300; backwards1 <= 0; end
                3'b11: begin yvel1 <= 14; xcostart1 <= 400; backwards1 <= 0; end
                3'b100: begin yvel1 <= 10; xcostart1 <= 500; backwards1 <= 1; end
                3'b101: begin yvel1 <= 14; xcostart1 <= 600; backwards1 <= 0; end
                3'b110: begin yvel1 <= 14; xcostart1 <= 250; backwards1 <= 1; end
                3'b111: begin yvel1 <= 14; xcostart1 <= 300; backwards1 <= 1; end
                endcase
        end
endmodule

module lookuptablebomb (input clk, input [2:0] randomnumber,
                output reg [4:0] yvel1, output reg backwards1, output reg [9:0] xcostart1);

        always@(posedge clk) begin
                case (randomnumber)
                3'b00: begin yvel1 <= 16; xcostart1 <= 500; backwards1 <= 0; end
                3'b01: begin yvel1 <= 14; xcostart1 <= 400; backwards1 <= 0; end
                3'b10: begin yvel1 <= 16; xcostart1 <= 200; backwards1 <= 1; end
                3'b11: begin yvel1 <= 14; xcostart1 <= 300; backwards1 <= 0; end
                3'b100: begin yvel1 <= 10; xcostart1 <= 300; backwards1 <= 0; end
                3'b101: begin yvel1 <= 14; xcostart1 <= 510; backwards1 <= 1; end
                3'b110: begin yvel1 <= 14; xcostart1 <= 300; backwards1 <= 0; end
                3'b111: begin yvel1 <= 14; xcostart1 <= 300; backwards1 <= 0; end
                endcase
        end
endmodule
```

Random Bits Generator

```
module randombitsgenerator(
       input clk,
       input data,
       output [15:0] randomnumber);

    reg [15:0] shift_reg = 16'hFF_FF; // initial value

    always @(posedge clk) begin // reset all counters
        shift_reg[0] <= shift_reg[15] ^ data;
        shift_reg[1] <= shift_reg[0];
        shift_reg[2] <= shift_reg[1] ^ shift_reg[15] ^ data;
        shift_reg[14:3] <= shift_reg[13:2];
        shift_reg[15] <= shift_reg[14] ^ shift_reg[15] ^ data;
    end

    assign randomnumber = shift_reg;

endmodule
```

Coordinate Generator and Slice Coordinate Generator

```
module coord_generator (input rdy,
               input [2:0] slice,
               input active,
               input vsync,
               input [4:0] yvel,
               input backwards,
               input [9:0] xcostart,
               output [9:0] x_coord,
               output reg [2:0] fell,
               output [9:0] y_coord,
              output reg activeconst,
              output reg [7:0] score,
               output [4:0] yvelocity,
              output reg new);

    parameter START = 0;
    parameter CALC = 1;

    reg [9:0] x_coord1;
    reg [9:0] y_coord1;
    reg [4:0] y_vel;
    reg [3:0] x_vel;
    reg y_up;
    reg change;
    reg left;
    reg reachedzero;
    reg [3:0] counter;
    reg state;
    reg oldrdy;
    reg oldslice;

    always @(posedge vsync) begin
        oldrdy <= rdy;

        if (~rdy && oldrdy) begin
               fell <= 0;
               state <= START;
               y_coord1 <= 768;
        end
        if (rdy && ~oldrdy) score <= 0;

        if (rdy) begin
              case (state)

              START: begin
                     activeconst <= active;
                     y_coord1 <= 700;
                     counter <= 0;
                     y_vel <= yvel;
                     x_coord1 <= xcostart;
                     reachedzero <= 0;
                     state <=  CALC;
                     x_vel <= 5;
                     left <= backwards;
                     new <= 1;
                     end
              CALC: begin
                     new <= 0;
                     oldslice <= ~(slice == 0) && (activeconst);

                     // change every 8 cycles
                     if (counter == 7) begin counter <= 0; change <= 1; end
                     else begin counter <= counter + 1; change <= 0; end

                     // has the velocity reached zero
                     if (y_vel == 0) reachedzero <= 1;

                     // which way does velocity change
                     y_vel  <=  (change  &&  !reachedzero)?  y_vel  -  2:  (change  &&  reachedzero)?
y_vel +
                     2: y_vel;

                     // does it move up our down
                     y_up <= ~reachedzero;

                     y_coord1 <= (reachedzero && y_coord1 >768)? 768: (y_up)?
                            (y_coord1 - y_vel) : (y_coord1 + y_vel);

                     // wraps around so user has more time to be able to cut the
                     // fruit
                     x_coord1 <= (left)?    x_coord1 - x_vel: x_coord1 + x_vel;

                     state <= (reachedzero && y_coord1 > 768)? START: CALC;

                     fell <= (reachedzero && (y_coord1 > 768) && (slice == 0) &&
                            (activeconst)) ? fell + 1: fell;

                     score <= ((~(slice == 0) && (activeconst)) && ~oldslice)? score
                            + 1: score;
                     end

                      default: state <= START;

               endcase
        end
    end
    assign x_coord = x_coord1; assign y_coord = y_coord1; assign yvelocity = y_vel;
endmodule
```

```
module coord_generator_slice (input begincalc,
                             input vsync,
                             input [4:0] yvel,
                             input new,
                             input backwards,
                             input [9:0] xcostart,
                             input [9:0] ycostart,
                             output [9:0] x_coord,
                             output [9:0] y_coord);

    parameter START = 0;
    parameter CALC = 1;

    reg [9:0] x_coord1;
    reg [9:0] y_coord1;
    reg [4:0] y_vel;
    reg [3:0] x_vel;
    reg y_up;
    reg change;
    reg left;
    reg reachedzero;
    reg [3:0] counter;
    reg state = 0;

    always @(posedge vsync) begin
               case (state)
                      START: begin
                             counter <= 0;
                             y_vel <= 0;
                             x_coord1 <= xcostart;
                             y_coord1 <= ycostart;
                             reachedzero <= 1;
                             state <=  (begincalc) ? CALC: START;
                             x_vel <= 4;
                             left <= backwards;
                      end

                      CALC: begin
                      // change every 8 cycles
                      if (counter == 8) begin counter <= 0; change <= 1; end
                      else begin counter <= counter + 1; change <= 0; end

                      y_vel <= (change)? y_vel + 2: y_vel;

                      if (new) state <= START;
                      else y_coord1 <= (y_coord1 > 768) ? 768 : y_coord1 + y_vel;

                      x_coord1 <= (left)?    x_coord1 - x_vel: x_coord1 + x_vel;
                      end
                      default: state <= START;
               endcase
        end
    assign x_coord = x_coord1;
    assign y_coord = y_coord1;
endmodule
```

Slice Dealer (aka Slice Manager/Slice Finite State Machine)

```
module slice_dealer(input clk,
                    input appleactive,
                    input orangeactive,
                    input peachactive,
                    input apple_new,
                    input orange_new,
                    input peach_new,
                    input [23:0] cursorpix,
                    input [23:0] applepix,
                    input [23:0] orangepix,
                    input [23:0] peachpix,
                    output reg  applesliced,
                    output reg orangesliced,
                    output reg peachsliced);

    parameter WAIT = 0;
    parameter SLICE = 1;
    reg state_apple;
    reg state_orange;
    reg state_peach;

    always@(posedge clk) begin
        case (state_apple)
              WAIT: begin
              if (|cursorpix && |applepix) begin
                     applesliced <= 1;
                     state_apple <= SLICE;
              end
              else applesliced <= 0;
              end
              SLICE: begin
              if (apple_new && appleactive) begin
                     state_apple <= WAIT;
                     applesliced <= 0;
              end
              else applesliced <= 1;
              end
               default: state_apple <= SLICE;
        endcase
        case (state_orange)
               WAIT: begin
              state_orange <= (|cursorpix && |orangepix)? SLICE: WAIT;
              orangesliced <= 0;
              end
              SLICE: begin
              orangesliced <= 1;
              state_orange <= (orange_new && orangeactive)? WAIT: SLICE;
              end
              default: state_orange <= WAIT;
        endcase
        case (state_peach)
              WAIT: begin
              state_peach <= (|cursorpix && |peachpix)? SLICE: WAIT;
              peachsliced <= 0;
              end
              SLICE: begin
```

```
            peachsliced <= 1;
            state_peach <= (peach_new && peachactive)? WAIT: SLICE;
            end
            default: state_peach <= WAIT;
        endcase
    end
endmodule
```

Sound Effects

```
module sfx(
    input clk,
    input apple_slice,
    input orange_slice,
    input peach_slice,
    input bomb_slice,
    input [1:0] lost_life,
    input busy,
    input [1:0] state_in,
    output reg sound,
    output reg r);

    reg old_apple_slice, old_orange_slice, old_peach_slice, old_state;
    reg old_bomb_slice;
    reg [1:0] old_lost_life;
    reg [7:0] old_high_score;
    reg [24:0] counter= 0;
    reg [2:0] state = 0;
    reg [17:0] freq_counter = 0;

    parameter WAIT = 0;
    parameter PLAY_FRUIT = 1;
    parameter PLAY_BOMB = 2;
    parameter PLAY_LIFE = 3;
    parameter PLAY_HIGHSCORE0 = 6;
    parameter PLAY_HIGHSCORE1 = 4;
    parameter PLAY_HIGHSCORE2 = 5;

    always @(posedge clk) begin
        old_apple_slice <= apple_slice;
        old_orange_slice <= orange_slice;
        old_peach_slice <= peach_slice;
        old_bomb_slice <= bomb_slice;
        old_lost_life <= lost_life;
        old_state <= state_in;
        case (state)
               WAIT: begin
                     if (~old_bomb_slice && bomb_slice) state <= PLAY_BOMB;
                     else if (((~old_apple_slice && apple_slice) || (~old_orange_slice &&
                     orange_slice) || (~old_peach_slice && peach_slice)) && (old_state ==
                     state_in)) state <= PLAY_FRUIT;
                     else if (old_lost_life != lost_life && old_lost_life != 3)
                            state <= PLAY_LIFE;
                     else state <= WAIT;
                     sound <= 0;
                     counter <= 0;
                     freq_counter <= 0;
               end
               PLAY_FRUIT: begin
                      if (counter < 6500000) counter <= counter + 1; // play for 0.1 sec
                      else begin
                              freq_counter <= 0;
                              counter <= 0;
                              state <= WAIT;
                      end
                      if (freq_counter < 30000) begin
                              freq_counter <= freq_counter + 1;
                      end
                      else begin
                              freq_counter <= 0;
                              sound <= ~sound;
                      end
              end
               PLAY_BOMB: begin
                      if (busy) r <= 1;
                      if (counter < 30000000) counter <= counter + 1; // play for 0.5 sec
                      else begin
                              counter <= 0;
                              freq_counter <= 0;
                              state <= r ? PLAY_HIGHSCORE0 : WAIT;
                      end
                      if (freq_counter < 150000) begin
                              freq_counter <= freq_counter + 1;
                      end
                      else begin
                              freq_counter <= 0;
                              sound <= ~sound;
                      end
               end
               PLAY_LIFE: begin
                      if (busy) r <= 1;
                      if (counter < 6500000) counter <= counter + 1; // play for 0.5 sec
                      else begin
                              counter <= 0;
                              freq_counter <= 0;
                              state <= (r && lost_life == 3) ? PLAY_HIGHSCORE0 : WAIT;
                      end
                      if (freq_counter < 150000) begin
                              freq_counter <= freq_counter + 1;
                      end
                      else begin
                              freq_counter <= 0;
                              sound <= ~sound;
                      end
               end
               PLAY_HIGHSCORE0: begin
                      if (counter < 30000000) counter <= counter + 1;
                      else begin
                              counter <= 0;
                              freq_counter <= 0;
                              state <= PLAY_HIGHSCORE1; end
               end
               PLAY_HIGHSCORE1: begin
                      r <= 0;
                      if (counter < 15000000) counter <= counter + 1; // play for 0.1 sec
                      else begin
                              freq_counter <= 0;
                              counter <= 0;
                              state <= PLAY_HIGHSCORE2;
```

```
                    end
                    if (freq_counter < 110670) begin
                        freq_counter <= freq_counter + 1;
                    end
                    else begin
                        freq_counter <= 0;
                        sound <= ~sound;
                    end
                end
                PLAY_HIGHSCORE2: begin
                    if (counter < 30000000) counter <= counter + 1; // play for 0.1 sec
                    else begin
                        freq_counter <= 0;
                        counter <= 0;
                        state <= WAIT;
                    end
                    if (freq_counter < 82909) begin
                        freq_counter <= freq_counter + 1;
                    end
                    else begin
                        freq_counter <= 0;
                        sound <= ~sound;
                    end
                end
                default: state <= WAIT;
            endcase
    end
endmodule
```

Max Score

```
module max_score(
    input [15:0] current_score,
    input [15:0] score_from_flash,
    input clk,
    input ready,
    input busy,
    input reset_score,
    output reg reset,
    output reg writing,
    output reg reading,
    output reg up_reset,
    output [15:0] score,
    output reg [15:0] score_to_store,
    output [4:0] state_out);

    parameter STARTUP = 7'b00001;
    parameter READ_LOOP = 7'b00011;
    parameter CHECK = 7'b00010;
    parameter RESET = 7'b00100;
    parameter STORE = 7'b01000;
    parameter IDLE= 7'b10000;

    reg [4:0] state = STARTUP;
    reg [15:0] high_reg = 0;
    reg [8:0] counter = 0;
    reg old_reset_score;

    always @(posedge clk) begin
```

```
        old_reset_score <= reset_score;
        case (state)
               STARTUP: begin
                      reading <= 1;
                      state <= READ_LOOP;
               end
               READ_LOOP: begin
                      if (counter < 200) counter <= counter + 1;
                      else begin
                             counter <= 0;
                             state <= CHECK;
                             reading <= 0;
                      end
               end
               CHECK: begin
                      if (~busy) begin
                             high_reg <= score_from_flash;
                             state <= IDLE;
                      end
               end
               RESET: begin
                      up_reset <= 0;
                      reset <= 0;
                      if (~busy) begin
                             state <= STORE; // if done resetting, go to store state
                             writing <= 1;
                      end
               end
               STORE: begin
                      if (~busy) state <= IDLE; // done storing new score
               end
               IDLE: begin
                      if ((current_score > high_reg) && ready) begin
                             writing <= 0;
                             state <= RESET;
                             reset <= 1;
                             up_reset <=1;
                             score_to_store <= current_score;
                             high_reg <= current_score;
                      end
                      else if (~old_reset_score && reset_score) begin
                             writing <= 0;
                             state <= RESET;
                             reset<= 1;
                             up_reset <= 1;
                             score_to_store <= 0;
                             high_reg <= 0;
                      end
               end
               default: state <= IDLE;
        endcase
    end
    assign score = high_reg;
    assign state_out = state;
endmodule
```

Cursor Coordinates

```
module cursor_coords
       (input serial_data,
       input clk,
       output [15:0] x_coord,
       output [15:0] y_coord,
       output [7:0] button);

       parameter HIGH_STATE = 0;
       parameter FALLING = 1;
       parameter FIRST = 2;
       parameter DATA = 3;
       parameter DONE = 4;

       reg [17:0] high = 0; // hold all 20 ms between serial data pulses
       reg [12:0] counter = 0;
       reg [39:0] data = 0;
       reg [15:0] x = 507;
       reg [15:0] y = 379;
       reg [7:0] b;
       reg [2:0] state = 0;
       reg [3:0] num_bits;
       reg [2:0] num_bytes;

       always @(posedge clk) begin
              case (state)
                     HIGH_STATE: begin
                     if (high == 1300) begin
                            high <= 0;
                            state <= FALLING;
                     end
                     else begin
                            if (serial_data) high <= high + 1;
                            else high <= 0;
                     end
                     end
                     FALLING:
                     if (~serial_data) begin
                            state <= FIRST;
                            counter <= 0;
                     end
                     else begin
                            state <= FALLING;
                            num_bits <= 1; // have stored the start bit
                     end
                     FIRST:
                     if (counter == 3384) begin
                            state <= DATA;
                            counter <= 0;
                            num_bits <= 1; // have stored the start bit
                     end
```

```
                    else counter <= counter + 1;
                    DATA:
                    if (counter == 6769) begin // 16x 16 cycles
                           if (num_bits < 9) begin
                                  counter <= 0;
                                  data <= {data[38:0], serial_data};
                                  num_bits <= num_bits + 1;
                           end
                           else if (num_bits == 9) begin // have stored 9 bits
                                  if (num_bytes == 4) begin // about to get last byte
                                         state <= DONE;
                                         num_bytes <= 0;
                                  end
                           else begin
                                  state <= FALLING;
                                  num_bytes <= num_bytes + 1;
                           end
                    end
                    end
                    else counter <= counter + 1;
                    DONE: begin
                    state <= HIGH_STATE;
                    x <= {data[31:24],data[39:32]};
                    y <= {data[15:8], data[23:16]};
                    b <= data[7:0];
                    high <= 0;
                    end
                    default: state <= HIGH_STATE;
             endcase
      end
      assign x_coord = x;
      assign y_coord = y;
      assign button = b;
endmodule
```

Background

```
module bg(
    input clk,
       input vsync,
       output [23:0] pixel
       );
    wire [7:0] green = 164;
    wire [7:0] blue = 255;
    reg [7:0] red = 0;
    reg up = 1;
    reg old_vsync;

    always @(posedge clk) begin
        old_vsync <= vsync;
        if (~old_vsync && vsync) begin
               if (up) red <= red + 1;
               else red <= red - 1;
               if (red + 1 == 255 && up) up <= 0;
               else if (red == 1 && ~up) up <= 1;
        end
    end
    assign pixel = {red, green, blue};
endmodule
```

Bomb

```
module bomb
   #(parameter WIDTH = 150,  // default picture width
              HEIGHT = 150)         // default picture height
       (input pixel_clk,
       input [2:0] slice,
       input [10:0] x,hcount,
       input [9:0] y,vcount,
       input active,
       output reg [23:0] pixel);

   wire [14:0] image_addr;   // num of bits for 256*240 ROM
   wire [7:0] image_bits, red_mapped, green_mapped, blue_mapped;
   reg currentinactive;
   reg oldnotinair;

   // calculate rom address and read the location
   assign image_addr = (hcount-x) + (vcount-y) * WIDTH;
   bomb_rom bomb(.clka(pixel_clk), .addra(image_addr), .douta(image_bits));

   // use color map to create 8 bits R, 8 bits G, 8 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   // use color map to create 8bits R, 8bits G, 8 bits B;
   bomb_red_rom rcm (.clka(pixel_clk), .addra(image_bits), .douta(red_mapped));
   bomb_green_rom gcm (.clka(pixel_clk), .addra(image_bits), .douta(green_mapped));
   bomb_blue_rom bcm (.clka(pixel_clk), .addra(image_bits), .douta(blue_mapped));

    parameter SLICE_WIDTH = 10;
   // note the one clock cycle delay in pixel!
   always @ (posedge pixel_clk) begin
        if (active) begin
              if ((hcount >= x && hcount < (x+WIDTH)) && (vcount >= y && vcount <
              (y+HEIGHT)))
                     pixel = {red_mapped, green_mapped, blue_mapped};
               else pixel = 0;
        end
    end
endmodule
```

Apple (same for peach and orange with different ROMs)

```
module apple
   #(parameter WIDTH = 150,  // default picture width
              HEIGHT = 150)  // default picture height
       (input pixel_clk,
       input [2:0] slice,
       input [10:0] x, xslice, hcount,
       input [9:0] y, yslice, vcount,
       input active,
       output reg [23:0] pixel);

  reg [14:0] image_addr;
  wire [14:0] image_addr_1; wire [14:0] image_addr_2;   // num of bits for 256*240 ROM
  wire [7:0] image_bits, red_mapped, green_mapped, blue_mapped;
  reg currentinactive; reg oldnotinair;

  // calculate rom address and read the location
  assign image_addr_1 = (hcount-x) + (vcount-y) * WIDTH;
  rom2 apple_rom2(.clka(pixel_clk), .addra(image_addr), .douta(image_bits));
  assign image_addr_2 = (hcount-xslice) + (vcount-yslice) * WIDTH + (WIDTH * HEIGHT/2);

  // use color map to create 8 bits R, 8 bits G, 8 bits B
  // since the image is greyscale, just replicate the red pixels
  // and not bother with the other two color maps.
  // use color map to create 8bits R, 8bits G, 8 bits B;
  apple_red_rom rcm (.clka(pixel_clk), .addra(image_bits), .douta(red_mapped));
  apple_green_rom gcm (.clka(pixel_clk), .addra(image_bits), .douta(green_mapped));
  apple_blue_rom bcm (.clka(pixel_clk), .addra(image_bits), .douta(blue_mapped));

  parameter SLICE_WIDTH = 10;
  // note the one clock cycle delay in pixel!
  always @(posedge pixel_clk) begin
       if (active) begin
              if ((hcount >= x && hcount < (x+WIDTH)) && (vcount >= y && vcount <
              (y+HEIGHT)))
              case (slice)
                     0: begin
                            pixel <= {red_mapped,green_mapped, blue_mapped};
                            image_addr <= image_addr_1;
                     end
                     1: begin
                     if (((vcount >= y + HEIGHT/2 )) && (hcount >= x && hcount < (x+WIDTH)))
                            pixel <= 0;
                     else begin
                            pixel <= {red_mapped,green_mapped, blue_mapped};
                            image_addr <= image_addr_1;
                     end
                     end
                     default: begin
                            pixel <= {red_mapped,green_mapped, blue_mapped};
                            image_addr <= image_addr_1;
                     end
              endcase
              else pixel <= 0;

              if ((hcount >= xslice && hcount < (xslice+WIDTH)) && (vcount >= yslice &&
              vcount < (yslice+HEIGHT)))
                     if (slice == 1) begin
                            if (((vcount < yslice + HEIGHT/2) && (vcount > yslice)) &&
                            (hcount >= xslice && hcount < (xslice+WIDTH))) begin
                                   pixel <= {red_mapped,green_mapped,blue_mapped};
                                   image_addr <= image_addr_2;
                            end
                     end
              end
       else pixel <= 0;
   end
endmodule
```