Wednesday, December 12, 2012
6.111 Final Project Report

Augmented Reality Fruit Ninja™

Nathan Monroe, Drew Dennison, Isaac Evans

Abstract

Augmented Reality Fruit Ninja is a fully-functional system that lets people play the popular game *Fruit Ninja* using just their hands to play the game. The system works by using a video camera to track the players hands to move a sword around on screen. If the sword and a moving fruit intersect, the fruit is 'cut' and the player gains a point. Our implementation includes a special "cheat" mode and bombs that trigger sudden death if a player hits a bomb. This paper introduces our work, outsides a block diagram, covers the details of each module, and briefly covers a few ideas we have for future improvement. Finally, we have attached all of the code we wrote.

Table of Contents

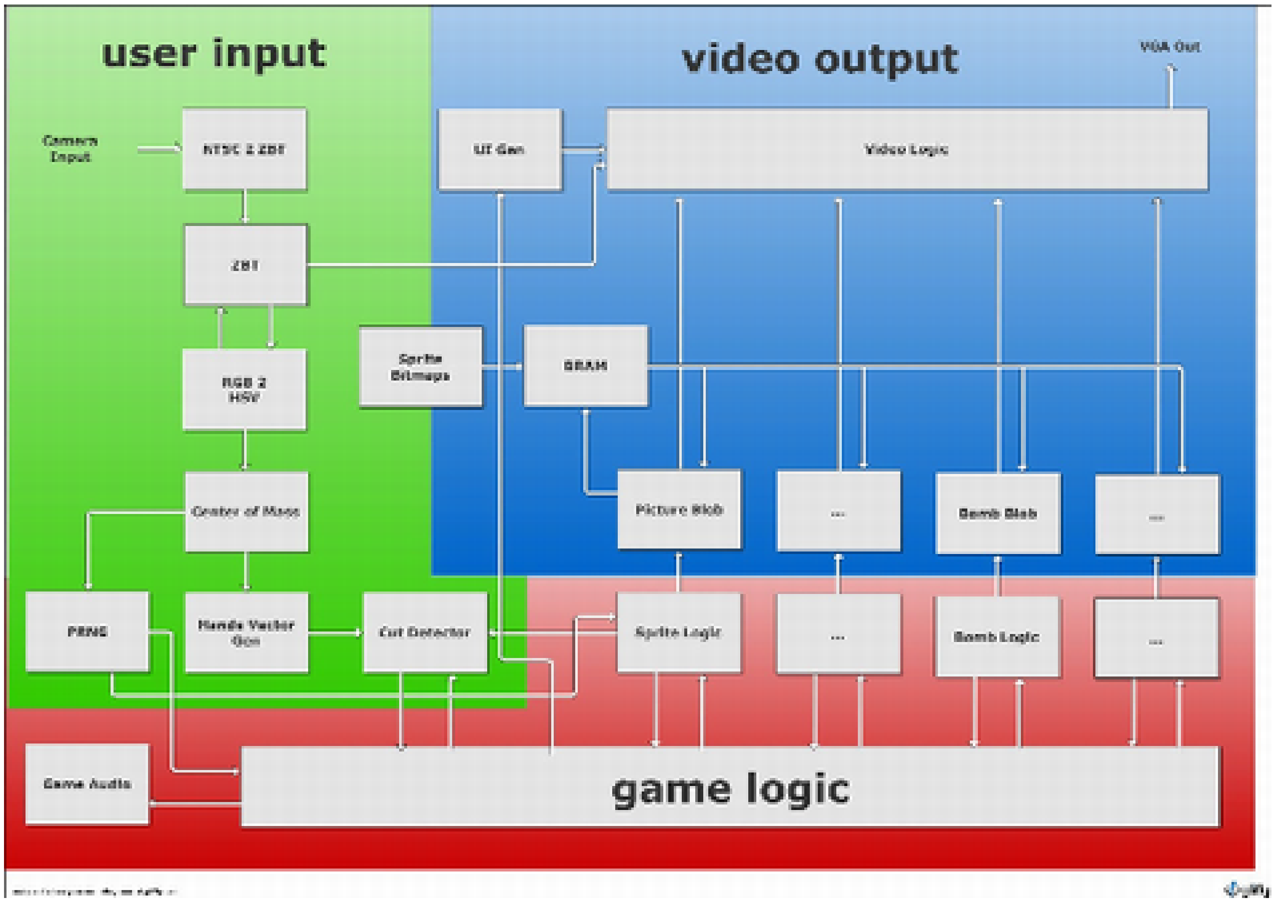
Start Screen of Augmented Reality Fruit Ninja

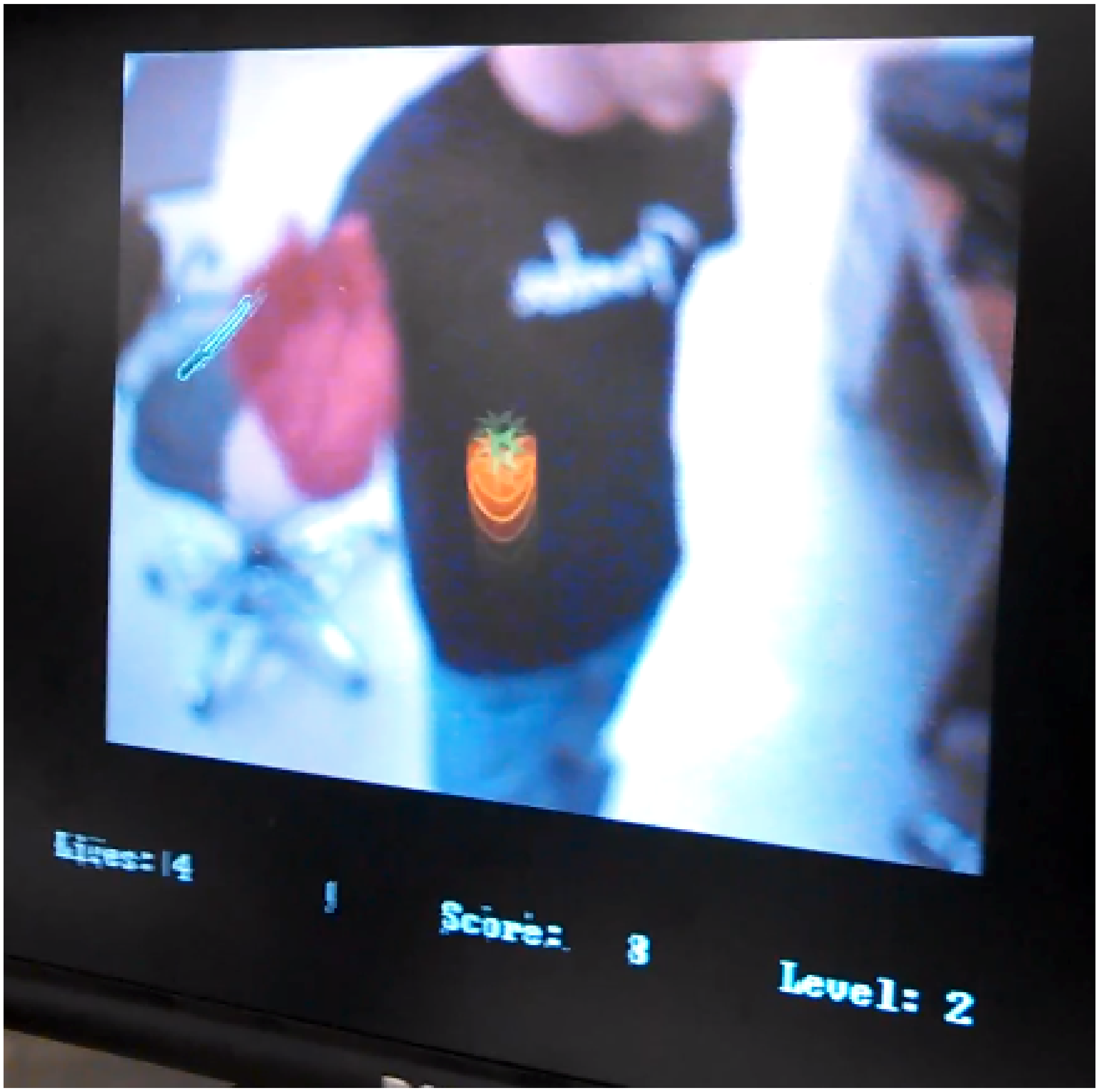
Overview

This project is inspired by Fruit Ninja™, a video game in which fruit appears on a touchscreen and a user must swipe across the fruit to cut it in half before it hits the bottom of the screen. We implemented augmented-reality Fruit Ninja™ on the FPGA labkit, using a NTSC camera to track the user's gloved hand which act as the controller for the game. We display a slightly blurred version of the the live input feed underneath our game overlay, which includes pieces of "fruit" or TAs' heads that the users must "cut" by waving their hands. The users wear a red glove to facilitate hand tracking. We implemented both fruits and TAs' heads, sound, and realistic gravity physics for the game to create a more delightful user experience. This project used the standard 6.111 labkit, as well as a NTSC camera and red gloves.

The basic functionality is that the FPGA generates fruits to be added to the game, with the frequency of new fruits increasing as the game progresses to higher levels. Fruit cutting events are calculated based on hand vector detection from camera input and calculated position of fruits. A "miss" event is calculated by the fruit reaching the bottom of the screen before being cut. With both the cut and miss events, the game score, level and number of lives left are updated accordingly, with the game ending after level 8 or if all lives are lost. Video modules create proper video signals based on fruit positions, as well as a user interface which displays information such as score, level, and number of lives left. Fruit sprites are generated from bitmap pixel data read from BRAM. The game also features six sound effetcs, which are stored as samples in the FPGA's block ram.

Block Diagram





Screen grab from video of actual game play by Nathan

Modules

The modules are hierarchically divided into three main groups: User Input, Game Logic, and Video Output. The logical flow of information is roughly the following. Video signals come in and are converted into frames by the NTSC2ZBT module. These frames are stored in the ZBT memory. Eventually these frames are read back out to the video output. For image processing, the frames are converted from pixels being represented in RGB (red, green, blue) representation to HSV (hue, saturation, value). This is done in the RGB2HSV module. This allows for easier image processing and mitigates issues due to noise and lighting conditions. From there, the Center of Mass module finds hand positions using a center of mass of the glove colors. The HandsVectorGenerator module reads hand positions over multiple frames to generate hand vectors. The GetoRandom module takes these hand positions and generates a pseudorandom value from the low-order hand-position bits. The CutDetector module determines if a fruit has been cut, based on hand vectors and fruit positions. The game logic module stores game state, game rules, and values such as score, lives left and level. Sprite logic modules calculates sprite position based on game physics. Picture Blob and bomb blob modules generate video signals for the sprite based on sprite position information and bitmap pixel data from BRAM. The UI gen module will generate video signals for a user interface based on signals for game score, level, and lives left. Also, a splash screen is overlayed on the video during initial game play. The top and UI Wrapper modules combine video signals from the UI, the fruit sprites, and the video frame into a single video output.

User Input

1. **NTSC2ZBT - Modified by Isaac**
   Takes in the NTSC camera data and stores it as a frame in the ZBT RAM. This module was already available from the course staff in black and white, so we modified it support color images. This was fairly straightforward, just involving expanding some of the register arrays. For space efficiency, we stored only the 6 high order bits of each byte, which allowed us to fit two RGB pixels into each 36 bit-wide ZBT memory location. There were a some glitches in the module that resulted in frame data being written to the first several pixels of the output image redundantly; this ended up not being an issue as we just clipped out those pixels when adding our UI screen.
   Inputs: Staff-provided NTSC decoder output
   Outputs: A 800x600 image frame in ZBT RAM.

1. **VRAM Display - Modified by Isaac**
   RAM storage for camera frame data and sprite bitmap data. Sprite bitmap data will be statically loaded at compile-time. The main modification to what was provided us by the course staff was adding color. We changed the hc4 variable, which previously switched between the four bytes (each byte representing a pixel) stored in a single ZBT location, to hc2, which switched between the two RGB pixels stored in a ZBT location. We then modified the address generation code to increment at double the previous rate since we were only reading two pixels instead of four; see the vram_addr variable in the vram_display module. Finally, we reversed the input *hcount* so that the when the user faced the screen, the user’s hand would move in the same direction on the screen as

well as in space (rather than being inverted). Additionally, we expanded the 6-bit values from ZBT RAM to 8 bit by appending 1's.
Inputs: NTSC2ZBT, precompiled bitmaps
Outputs: camera frame images, bitmaps

1. **RGB2HSV - Modified by Isaac**
   Converts RGB pixel map to HSV for better hand recognition. This module was provided to us by the course staff. One serious issue with the module is that there is a 22 clock cycle delay that must be accounted for when doing detection. We also had to convert the YCrCb input input, which is the NTSC format, to RGB before we could pass it into this module. Fortunately the code to do so was straightforward and provided.
   Inputs: R, G, B (red, green, blue) pixel color
   Outputs: H, S, V (hue, saturation, value) pixel color

1. **Center of Mass - Isaac**
   Generates hands positions as XY coordinates from frame data. The input "detected" bit which indicates whether the pixel in position (x, y) met our criteria for HSV detection ranges. The H range was implemented as the OR of two bounded ranges, matching between [0, *lo*) or (high, 255]. The S and V values were simple thresholds. Late in the project, we wired the detected wire to do its comparison with threshold registers which were could be modified by the up/down buttons on the labkit and had their values displayed on the labkit hex display. This way we could easily tune a range parameter while the system was running; this was key in honing our detection. Our final values for a red glove on the user's hand were as follows:

```
hue_thresh_low <= 8'h09;
hue_thresh_high <= 8'hec;
sat_thresh <= 8'hb7;
val_thresh <= 8'h22;
```

   Where the detected output was given by:

```
assign detected =
(H < hue_thresh_low || H > hue_thresh_high) &&
(S > sat_thresh) &&
(V > val_thresh);
```

   We distinguish between a pixel being "detected" and a pixel being "used." All pixels meeting the HSV thresholds were detected, but only pixels which had the previous 10 pixels in the VGA scan marked as detected are marked as "used."

When one of the labkit switches is enabled for debugging, we set the color of a “detected but not used” bit to green and the color of a “used” pixel to red. Displaying the output color diagnostics resulted in some interesting bugs. Of course, the color was shifted constantly to the right by 22 pixels off because of the 22 clock cycle delay in RGB2HSV (mentioned previously). Additionally, we had a problem where as soon as a red pixel was detected it would “bleed” all the way across the screen; this ended up being a loop we were accidentally generating by setting the pixel value to red for diagnostic but then that modified pixel value was passed into RGB2HSV, propagating it further through the image.

The final algorithm added all the x and y of pixels which were marked as “used” to x and y accumulators and then divided them by the used pixel count. We used the IPCore generator to build a pipelined (v3.0) divider that could handle the divisions quickly, and latched the new divisor output when the divider output was valid. The module was extremely fast and easily kept up with our frame rate.

Finally, the extrapolation bit, when set, causes the (x, y) position to be extrapolated out of the 800x600 camera image onto the 1024x768 VGA output coordinate space. We were able to achieve this with a simple bit shift and subtractions. Even though the mapping isn’t mathematically truly perfect, it is nearly impossible for the user to generate values at the extreme edges of the screen with the center of mass algorithm, so by using the multiplication by two and slowly tuning the subtracted values for x and y we were able to arrive at a good-enough solution which proved quite robust.

Inputs: A “detected” bit along with XY coordinates, and an extrapolation bit

Outputs: (x, y) coordinate pairs corresponding to the user’s hands, as well as 5 delayed coordinates of previous hand positions. Also outpus a “used” bit that indicates whether or not the pixel was used in the center of mass calculation; this is displayed on the screen as an output diagnostic.

1. **HandsVectorGenerator - Isaac**

   Combined into CenterOfMass for simplicity. See the xDelayLine and yDelayLine registers in CenterOfMass.

   Inputs: One (x, y) coordinate pair representing a hand position

   Outputs: An (x, y) vector representing the hand’s motion vector over the past 250 milliseconds.

1. **CutDetector - Isaac**

   Determines if any of the sprites have been “cut” by the hands. The cut detection is based on vector defined from the five points output from the CenterOfMass module. A cut is detected if and only if (a) one of the two outer points is *not* inside the sprite and (b) one of the three inner points *is* inside the sprite. A utility module called IsInside was written to make this easier to read.

Inputs: two hand vectors, 6 sprite positions and states, and two bomb positions and states
Outputs: cut boolean for each sprite

Game Logic

1. **GameLogic - Nathan**
The game logic records and manipulates the overall state of the game. It also stores the essential game parameters of Score, Level, and Lives Left. The module brings fruits into the game based on set delays (see appendix A). Fruits come more frequently at higher levels, making the game more difficult as it progresses. The module removes fruits from the game either when the cut detector tells them they have been cut, or when they reach the bottom of the screen, based on position input from the Sprite Logic. If a fruit has been cut, then the score and level values are updated accordingly. If a fruit reaches the bottom of the screen without being cut, the number of lives left is updated accordingly. There is a maximum of six fruits on the screen at any given time. For each of the six possible active fruits, whether a fruit is active in the game is determined by a single bit register, "Sprite State". The sprite state is a single boolean: 1 if the sprite is actively on the screen and in the game, or 0 if not. The Game Logic module also has functionality for bombs. They have the same functionality and implementation as the fruits, with the difference being that if a bomb is cut, the game is automatically lost, and if a bomb reaches the bottom of the screen it is removed from the game with no other effect. There are a maximum of two bombs in the game at a given time. Bombs are brought into the game beginning at level two. There are eight levels in the game, starting with level 1. After level 8, the game is won. Level 0 is the 'start state', where the game is not actively in progress. Level zero is the state used for the start screen logic.
The game logic module includes both the timer and time_divider modules which, as in lab 4, act as timers. They are used to activate fruits and bombs into the game
Real-world testing has shown that the game is very difficult, with the maximum score reached in our tests being 51 (level 5). The score to win the game is set at 400 (level 8). We feel that this increases the replayability value in the game because it really takes lots of playtime to win. Or just an actual ninja.

2. **GameLogic Test Bench - Nathan**
The game logic testbench module tests functionality of the game logic module, ensuring the following:
1. Start screen "level 0" functions properly
2. Fruits are brought into the game at the proper frequency, which increases with increasing level
3. Score is incremented properly upon fruit cut event

4. Lives is decremented properly upon fruits reaching the bottom of the screen without being cut
5. Level is updated properly according to score
6. Bombs are brought into the game at the correct frequency and desired level, with proper 'game over' functionality upon their cut
7. The game is lost when lives run out and game function ceases
8. The game is won when the 'win score' is reached
9. All values reset properly upon reset

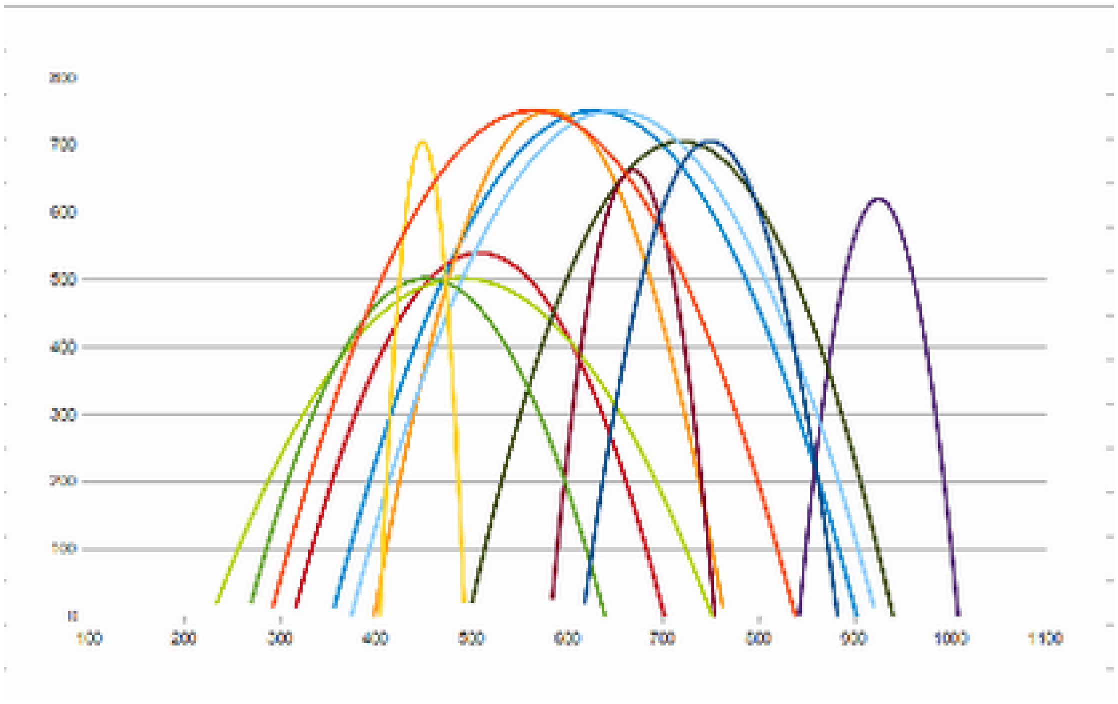
1. **SpriteLogic - Nathan**
   This module keeps track of the sprite position based on game physics, and input from the game logic for if that sprite should be on or not. Upon a sprite being brought into the game, it's X position is determined randomly, based on input from the randomizer. The Y position is also probabilistic, with it starting at the bottom of the screen 75% of the time, and the top of the screen 25% of the time.
   The initial Y (vertical) vector is probabilistic, based on input from the randomizer. If the sprite appears on the bottom of the screen, the Y vector is in the upwards direction, and is bounded such that it is between 15 and 21 pixels per second in the vertical direction. The randomization is centered around 18 pixels per second. These values, combined with the gravity discussed below, ensure that the sprites will never go above the top of the screen, and that at the very worst the sprites will apex at around halfway up the screen. If the sprite's starting position is at the top of the screen, which is 25% of the time, the initial Y vector is set at 1 pixel per frame in the downward direction.
   The game incorporates realistic gravity with a constant downward acceleration, such that the fruits initially flying upwards will arc and eventually fall downwards, and fruits initially at the top of the screen flying downwards will accelerate downwards. This is implemented in a manner similar to a floating point unit. There is a 6 bit register called (floatgrav). Upon each new frame, the floatgrav register is increased by a set value "gravity", which is between 0 and 63. Whenever the floatgrav register overflows, the Y vector is decremented by one. This will occur (gravity) / 64 of the time. Thus, the value of gravity in the game is equal to (gravity) / 64 frames per second per second, and is parametrized so it can be easily changed. For the values of Yvector listed above, the chosen gravity parameter was 22. This results in fruits being active in the game for approximately 5.5 seconds (or 2.75 seconds if the fruit falls downward from the top of the screen). This was determined empirically as being a reasonable amount of time for the user to cut the fruit.

   The X vector of the fruit was also determined probabilistically in a bounded fashion such that the fruits never fly off the left or right edges of the screen, which would be problematic. Based on a 5.5 second maximum time on the screen (or 165 frames), the X vector was bounded based on the starting X position such that the fruits were always on the active screen. If the fruit was initially in the left half of the screen, it had an X vector in the right direction, and vice versa. The vectors were bounded according to the following:

Starting X Position (pixels)	Maximum X vector (pixels per frame)
0-127	9 right
128-255	8 right
256-383	7 right
384-511	5 right
512-639	5 left
640-767	7 left
768-895	8 left
896-1023	9 left

The random X positions and Y positions as well as bounded random X and Y vectors combined with the gravity implementation to produce arcing fruits which always stayed within the screen, only leaving the screen upon a cut event or at the bottom edge if the fruit was missed by the user.

2. **SpriteLogic Test Bench - Nathan**
   The sprite logic testbench was used to produce simulations of arcing trajectories of the fruits in the game, based on randomly generated input. Plots were produced from multiple test runs to validate that the fruits never crossed the top of the screen, nor either of the side walls. This validation is impossible to prove with a finite number of test runs, but at the very least it was proven very unlikely with a very large sample of test runs.



MathLab plot of random outputs of x,y positions over time for sprites from Game Logic

3. **Geto Randomizer - Nathan**
   This module produces constantly updating numbers which appeared to be random. The numbers were the low-order bits of the hand position. Since the user moved their hands fairly rapidly, these numbers appeared to be random. This drove the random generation of fruit position and vectors.

4. **Game Audio - Nathan**
   The game audio module produces sound effects based on game events. Initially, sound effect samples were found online from a stock sound library. However, it was decided that for added character in the game, all sound effects should be recorded vocally. The game had six sound effects:

Event	Sound	Max Address (samples)	Priority
Fruit is cut	“Shink”	23386	1
Next Level	“Woohoo”	31996	2
Fruit is missed (lose a life)	“Ouch”	22254	3
Game is won	“You Win”	41806	4
Bomb is cut	“Boom”	42128	5
Game is lost	“You Fail”	50387	6

Sound effects were recorded vocally using a condenser microphone, being sampled at 48KHz. Samples were low-pass filtered at 6KHz to enable the possible future option of downsampling at 12KHz to save block ram space. Sound clips were filtered using Audacity audio software, and converted into .coe files using the TA-provided matlab script. Samples were stored with a resolution of 8 bits in the FPGA’s block ram. Block ram modules were produced using the built in Xilinx IP core generator.
Audio samples were passed to the ac97 module, as in lab 5. Each possible sound has a corresponding “ready” signal, which is turned on if that sound’s corresponding event occurs. The sounds are played based on a priority system, where only one of the sound’s corresponding block ram has control of the audio output at a given time. The audio output has an overall state which is “PLAYING” if a sound is playing, or “IDLE” otherwise. If the audio is in idle and a sound is ready, that sound will take control of the audio output, setting the overall state to “PLAYING” until that sound has reached it’s max address in block ram, indicating it is done playing.
The consequence of this implementation is that only one sound can be played at a time. For example, if two fruits are cut in less time than it takes to play the “cut” sound, the second “cut” sound will be delayed until the first is complete. The “cut” sound has the highest priority because it is likely to occur the most often. The current implementation features sound clips sampled at 48KHz, but due to the low pass filtering the clips could be easily downsampled to 12KHz if block ram space became a constraint.

Video Output

1. **Picture Blob - Drew**

Generates the pixels for all of the fruit sprites. This module takes as input the X and Y coordinates of 6 fruits and 2 bombs which is the maximum number of sprites needed for the hardest level. Each of the six fruits take a 3-bit selector wire so the Sprite Logic module can pick which fruit to use for each of the fruit sprites.

We implemented a "Cheat Mode" where if cheat=1, the fruits are replaced by graphics of the TAs and Writing Instructors' heads. We made Gim the bomb.
The technical underpinnings were 2 BRAMs of width 24 so each address was exactly 1 pixel. Each BRAM had to have space for 7, 32x32 sprites so the dimensions of the BRAM block memory layout generated by the IP Core utility in ISE were 24x7168 and required a 13-bit address bus.

This module also include a 3-bit level input and a 6-bit sprite_on input that uses 1 bit for each fruit to determine whether or not to display it. There is a corresponding 2-bit input signal that determines if either or both bombs should be displayed.

We had an input for a random number that would randomly pick which type of fruit to display when we activated a fruit sprite. However, we found that our random number generator module that uses the low-order bits of the hands was giving us too many 1s and 0s. Thus, we modified the module to simply cycle through each of the 6 possible types of fruit.

Every concept discussed above applies equally well in "cheat mode" with the staff heads instead of fruits and bombs.

The image address of the location to read from is calculated: image_addr = (1024*id) + ((hcount-x)/2) + ((vcount-y)/2) * 32 This address is correct because we scale the images up to 64x64 on the UI but the data in memory is only 32x32.

Each sprite is 1024 addresses apart (32x32) so we multiply the spriteID but 1024 to get the correct sprite from BRAM.

Fruit and staff sprites used. Staff images were produced by Nathan in photoshop. Images were taken from the course lecture slides.

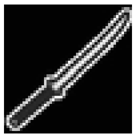
Inputs: sprite[n] (x, y), spriteID[n], spon, bombon, level
Outputs: 24-bit VGA video signal

2. **Level - Drew**
   Generates video signal for displaying the level the user is currently on.
   This module used the provided char_string_display and the accompanying font_rom.
   The input is a input is a 3-bit level signal that feeds into a case statement which concatenates the ASCII for “Level: “ with the ASCII for “0,1,...,7”
   Inputs: X and Y of upper-left corner, level
   Outputs: VGA output with “Level: X”

3. **Lives - Drew**
   Generates video signal for displaying the number of lives the user has remaining. Starts on 5 and counts down to 0 because that’s what the game logic module feeds it via the Lives input. This module used the provided char_string_display and the accompanying font_rom. The input is a input is a 3-bit lives signal that feeds into a case statement which concatenates the ASCII for “Lives: “ with the ASCII for “0,1,...,7”
   Inputs: X and Y of upper-left corner, lives
   Outputs: VGA output with “Lives: X”

4. **Score - Drew**
   Generates video signal for displaying the numerical score of the user. Starts at 0 and counts up to a possible 999. This module assumes that if the score changes, it must have increased by 1. The game logic module feeds the current game score via a 9-bit score input. This module uses the provided char_string_display and the accompanying font_rom. The input is a 9-bit score signal that feeds into a case statement which concatenates the ASCII for “Score: “ with the ASCII for 2 digits of “space,1,...,9” and 1 digit of ASCII for “0,1,...,9”
   Notice that the first two digits display a ‘space’ instead of 0 so if the score is 5 it displays as 5 and not 005.
   Inputs: X and Y of upper-left corner, score
   Outputs: VGA output with “Score: XXX”

5. **Sword - Drew**
   Generates video signals for a 64x64 image of a sword. This sword is used to point to the X, Y that is the center of mass of the user’s hand. This X,Y is feed into the system and 64 is subtracted from X so that similar code to the Picture Blob module can be used.



Bitmap of the sword. Produced by Nathan in photoshop.

Inputs: X and Y of the upper right corner
Outputs: VGA video signal

6. **UI Wrapper - Nathan**
   The UI wrapper module controls the sprite ID signals and the cheat signal. Most notably, it manipulates the signals during the splash screen in order to provide the desired effect of two options to start the game, either a watermelon, which if cut starts the game with fruit sprites (cheat=0), or a picture of Gim, which if cut starts the game with course staff sprites (cheat=1). The UI wrapper also rotates through the sprite ID's, so each possible fruit/TA is seen, as well as hard-coding the bombs in the game to appear as bombs.

7. **Top Module - Nathan**
   This module is the top level module for the game, and integrates all subsystems into a complete system. It features pushbuttons to change audio output volume. In integration, it also produces a reversed hcount that counts down from 1023 to zero, which is used to mirror the video feed horizontally, making it more natural for the user. The module also combines video signals from the following outputs:
   - -ZBT video pixels
   - -Sprite pixels
   - -Sword pixels
   - -Splash screen pixels
   - -Score UI pixels
   - -Level UI pixels
   - -Lives left UI pixels

   The ZBT video output from the camera has the lowest priority in video display, with everything else overlaying it. The module includes functionality to change the thresholds for detecting the user's glove. In addition, the switches turn on and off the following debugging info:
   - -Hand position mapping to entire screen
   - -Hilighting pixels within the color detection threshold
   - -Producing crosshairs representing hand position and vector
   - -Masking ZBT pixels surrounding valid video frame
   - -Master game reset

This module also incorporates muxes controlled by the level. If the level is zero, the 'sprite on' and 'sprite position' values are forced to produce the start screen. This module also accounts for the implementation detail that sprite position is given by the sprite logic as the center of the sprite, but taken in to the sprite blob generator as the upper left corner.

8. **Splash Screen - Drew**
   Generates video signals for a start screen. We were running low on available BRAM so we used a single bit for each of the 250x100 pixels to generate a mask that is ANDed with the video feed.

   Welcome to
FRUIT NINJA
   Start Screen Splash Mask

   Inputs: X and Y of top left corner
   Outputs: VGA video signal

9. **Python Helpers - Drew**
   We wrote 2 python scripts to help with this project. The first one serialized an image into a stream of 24 bit RGB values and transmitted this bitstream over a virtual serial-over-USB to a DLP-USB245M USB to 8-bit output. This python script worked perfectly and even though we ended up not using the compact flash, it was a significant portion of the overall work.

   The second python script was written after we made the design decision to switch to BRAM and we needed a way to generate .coe files from images. This bmp2coe script worked perfectly and it was faster to write a simple python script than try to modify the MatLab script from the website

10. **Unused CF code - Drew**
    A very significant portion of Drew's time was spent trying to get the compact flash and USB-to-fifo board working. I ran into a number of issues that I had to debug using the built-in display and the logic analyzer. Eventually, I got the code from the Conductor Hero team from 2007 working and modified for our purposes. Unfortunately, the compact flash was just too slow. It was an order of magnitude slower than we needed to refresh the pixels for VGA output. We would have had to cache portions of the CF to either ZBT or BRAM dynamically. We made the design decision to just switch to using BRAM

directly. ZBT was a strongly considered option but given the fact that would have to be pipelined to cope with the 2-cycle delay in read times, we chose to keep it simple and try the BRAM and only worry about ZBT or flash if we ran out of BRAM space. This was a good design decision and greatly reduced the number of places in our code that needed to be debugged.

Future Work

Even though our Fruit Ninja project met all of our expectations and most of our stretch goals, we know it could be even better with several additional features. The most impressive would be to add fruits that appear to be cut and have each part of the fruit continue its trajectory taking into account conservation of momentum instead of simply disappearing. The way we would accomplish this would be to have each fruit be composed of four independent quadrants for each image.

A really fun and definitely doable extension of our work would be add tracking for a second hand to enable two players to compete or cooperate. In addition, we would like to improve the user interface such that a trail follows the sword, giving a cutting effect and added feedback.
We would also like to implement a “high score” functionality, as well as some form of competition mode for multiple players.

Conclusion

Augmented Reality Fruit Ninja was a successful project that exceeded our expectations. We all learned a lot from this project and it was so much fun to build a recognizable game that everyone in lab enjoyed watching and playing.

One of the best lessons we learned as a team is that it is very difficult to debug hardware and we still have some unexplained glitches that will seemingly randomly appear that we can only attribute to subtle timing issues. Also, the project was at times frustrating because of ISE crashes, refusal to load files, and just the generally long compile times.

This was a fun, challenging project to design, program, test, debug, play, and complete. We will miss it!

Acknowledgements

Thank you Gim for a wonderful class that taught us the magic of programmable digital hardware!

We would like to thank all the TAs for their wonderful help, feedback, endless patience and advice. Best of luck in your future endeavors.

Team member Monroe would like to point out that he owes TA Devon Rosner a beer, as promised, in return for his help in debugging the game logic module.

Appendix A: Game parameters - Nathan

Level	Delay before next fruit (sec)	Score to next level	Delay before next bomb (fruits)	Notes
1	5	5	-	
2	4	15	7	Bombs now on
3	4	30	7	
4	3	50	7	
5	2	85	6	
6	1.5	125	6	
7	1	200	5	
8	0.5	400	4	Game won at score of 400

Appendix B: Fruit Ninja Verilog Code

NTSC2ZBT Verilog Code

```
// ntsc_2_zbt.v
// 6.111 final project
// Modified by Isaac Evans, ine@mit.edu

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);

  input     clk;   // system clock
  input     vclk;   // video clock from camera
  input [2:0]     fvh;
  input     dv;
  input [17:0]     din; // modification for b&w -> color
  output [18:0] ntsc_addr;
  output [35:0] ntsc_data;
  output     ntsc_we;    // write enable for NTSC data
  input     sw;       // switch which determines mode (for debugging)

  parameter     COL_START = 10'd150;
  parameter     ROW_START = 10'd50;

  // here put the luminance data from the ntsc decoder into the ram
  // this is for 1024 * 788 XGA display

  reg [9:0]     col = 0;
  reg [9:0]     row = 0;
  reg [17:0]     vdata = 0; // modification for b&w -> color
  reg        vwe;
  reg        old_dv;
  reg        old_frame;    // frames are even / odd interlaced
  reg        even_odd;    // decode interlaced frame to this wire

  wire     frame = fvh[2];
  wire     frame_edge = frame & ~old_frame;
```

```
always @ (posedge vclk) //LLC1 is reference
 begin
 old_dv <= dv;
 vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
 old_frame <= frame;
 even_odd = frame_edge ? ~even_odd : even_odd;

 if (!fvh[2])
  begin
     col <= fvh[0] ? COL_START :
       (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
     row <= fvh[1] ? ROW_START :
       (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
     // for b&w -> color - check this conditional?
     vdata <= (dv && !fvh[2]) ? din : vdata;
  end
 end

// synchronize with system clock

reg [9:0] x[1:0],y[1:0];
reg [17:0] data[1:0];  // modification for b&w -> color
reg      we[1:0];
reg        eo[1:0];

always @(posedge clk)
 begin
 {x[1],x[0]} <= {x[0],col};
 {y[1],y[0]} <= {y[0],row};
 {data[1],data[0]} <= {data[0],vdata};
 {we[1],we[0]} <= {we[0],vwe};
 {eo[1],eo[0]} <= {eo[0],even_odd};
 end

// edge detection on write enable signal
```

```
reg old_we;
wire we_edge = we[1] & ~old_we;
always @(posedge clk) old_we <= we[1];

// shift each set of four bytes into a large register for the ZBT

// mydata WILL go into the ZBT. ergo, nothing below this should be modified,
// except perhaps the clock rate
// reg [31:0] mydata; // modification for b&w -> color
reg [35:0] mydata; // modification for b&w -> color

always @(posedge clk)
 if (we_edge)
  begin
  //mydata <= { mydata[23:0], data[1] }; // modification for b&w -> color
 mydata <= { mydata[17:0], data[1] };
  end

// NOTICE : Here we have put 4 pixel delay on mydata. For example, when:
// (x[1], y[1]) = (60, 80) and eo[1] = 0, then:
// mydata[31:0] = ( pixel(56,160), pixel(57,160), pixel(58,160), pixel(59,160) )
// This is the root of the original addressing bug.


// NOTICE : Notice that we have decided to store mydata, which
//          contains pixel(56,160) to pixel(59,160) in address
//          (0, 160 (10 bits), 60 >> 2 = 15 (8 bits)).
//
//          This protocol is dangerous, because it means
//          pixel(0,0) to pixel(3,0) is NOT stored in address
//          (0, 0 (10 bits), 0 (8 bits)) but is rather stored
//          in address (0, 0 (10 bits), 4 >> 2 = 1 (8 bits)). This
//          calculation ignores COL_START & ROW_START.
//
//          4 pixels from the right side of the camera input will
//          be stored in address corresponding to x = 0.
//
```

```
  //        To fix, delay col & row by 4 clock cycles.
  //        Delay other signals as well.

  reg [39:0] x_delay;
  reg [39:0] y_delay;
  reg [3:0] we_delay;
  reg [3:0] eo_delay;

  always @ (posedge clk)
  begin
    x_delay <= {x_delay[29:0], x[1]};
    y_delay <= {y_delay[29:0], y[1]};
    we_delay <= {we_delay[2:0], we[1]};
    eo_delay <= {eo_delay[2:0], eo[1]};
  end

  // compute address to store data in
  wire [8:0] y_addr = y_delay[38:30];
  wire [9:0] x_addr = x_delay[39:30];

  // TODO - modify this?
  //wire [18:0] myaddr = {1'b0, y_addr[8:0], eo_delay[3], x_addr[9:2]};  // modification for b&w
-> color
  wire [18:0] myaddr = {y_addr[8:0], eo_delay[3], x_addr[9:1]};

  // Now address (0,0,0) contains pixel data(0,0) etc.

  // alternate (256x192) image data and address

  //wire [31:0] mydata2 = {data[0],data[1]};
  wire [35:0] mydata2 = {data[0],data[1]};  // modification for b&w -> color
  wire [18:0] myaddr2 = {1'b0, y_addr[8:0], eo_delay[3], x_addr[7:0]};
  //wire [18:0] myaddr2 = {y_addr[8:0], eo_delay[3], x_addr[9:1]};   // modification for b&w ->
color

  // update the output address and data only when four bytes ready
  reg [18:0] ntsc_addr;
  reg [35:0] ntsc_data;
//   wire       ntsc_we = sw ? we_edge : (we_edge & (x_delay[31:30]==2'b00)); // modification
for bw = color
    wire      ntsc_we = sw ? we_edge : (we_edge & (x_delay[30]==1'b0)); //

  always @(posedge clk)
//     if ( ntsc_we )
     begin
    ntsc_addr <= sw ? myaddr2 : myaddr;    // normal and expanded modes
    //ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};  // modification for b&w -> color
    ntsc_data <= sw ? mydata2 : mydata;
     end

endmodule // ntsc_to_zbt
```

Vram Display Verilog Code

```
// vram_display.v
// 6.111 final project
// Modified by Isaac Evans, ine@mit.edu
module vram_display(reset,clk,hcount,vcount,vr_pixel,
          vram_addr,vram_read_data);

  input reset, clk;
  input [10:0] hcount;
  input [9:0]     vcount;
  //   output [7:0] vr_pixel;       // modification for b&w -> color
  output [17:0] vr_pixel;

  output [18:0] vram_addr;
  input [35:0]  vram_read_data;

  //forecast hcount & vcount 8 clock cycles ahead to get data from ZBT
  wire [10:0]     hcount_f = (hcount >= 1048) ? (hcount - 1048) : (hcount + 8);
  wire [9:0]      vcount_f = (hcount >= 1048) ? ((vcount == 805) ? 0 : vcount + 1) : vcount;

  // wire [18:0] vram_addr = {1'b0, vcount_f, hcount_f[9:2]}; // modification for b&w -> color
  wire [18:0]     vram_addr = { vcount_f, hcount_f[9:1] }; // use all but the last bit of hcount

  // wire [1:0]      hc4 = hcount[1:0]; // modification for b&w -> color
  wire      hc2 = hcount[0] ; // hc = horizontal counter?

  // reg [7:0]  vr_pixel; // modification for b&w -> color
  reg [17:0]      vr_pixel;
  reg [35:0]      vr_data_latched;
  reg [35:0]      last_vr_data;

  always @(posedge clk)
    //last_vr_data <= (hc4==2'd3) ? vr_data_latched : last_vr_data; // modification for b&w ->
color
  last_vr_data <= (hc2) ? vr_data_latched : last_vr_data;

  always @(posedge clk)
  //vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched; // modification for
b&w -> color
  vr_data_latched <= (!hc2) ? vram_read_data : vr_data_latched;

  // modification for b&w -> color
  //   always @(*)        // each 36-bit word from RAM is decoded to 4 bytes
  //     case (hc4)
  //       2'd3: vr_pixel = last_vr_data[7:0];
  //       2'd2: vr_pixel = last_vr_data[7+8:0+8];
  //       2'd1: vr_pixel = last_vr_data[7+16:0+16];
  //       2'd0: vr_pixel = last_vr_data[7+24:0+24];
  //     endcase

  always @(*)
  case (hc2) // we now have two pixels stored in the zbt
  1'd1: vr_pixel = last_vr_data[17:0];
  1'd0: vr_pixel = last_vr_data[35:18];
  endcase
```

```
endmodule // vram_display
```

Center of Mass Verilog Code

```
// CenterOfMass.v
// 6.111 final project
// By Isaac Evans, ine@mit.edu

// calculates center of mass for a target hue within [target_hue_low,
// target_hue_high], or [0, target_hue_low] union [target_hue_high, 2^8] if
// invert is == 1.

module CenterOfMass #(parameter OLD_POINTS = 15)(inframe, clk, reset,
        ntsc_address, ntsc_we, x, y, H, S, V,
        target_hue_low, target_hue_high, comX, comY,
        detected, used, comXOld0, comYOld0, comXOld1,
        comYOld1, comXOld2, comYOld2, comXOld3, comYOld3,
        comXOld4, comYOld4, minX, minY, extrapolate);
  input wire extrapolate;
  input wire inframe;
  input wire clk;
  input wire reset;
  input wire [18:0] ntsc_address;
  input wire        ntsc_we;
  input wire        detected;
  input wire [10:0] x;
  input wire [9:0]  y;
  input wire [7:0]  H;
  input wire [7:0]  S;
  input wire [7:0]  V;
  input wire [7:0]  target_hue_low;
  input wire [7:0]  target_hue_high;
  output reg [10:0] comX;
  output reg [9:0]  comY;
  input wire [10:0] minX;
  input wire [9:0] minY;
```

```
    output reg          used;

//  reg [1024*768]    neighbors;
//  reg [1024]          last_scan_line;

    reg [10:0] xDelayLine [OLD_POINTS:0];
    reg [9:0] yDelayLine [OLD_POINTS:0];
    output wire [10:0] comXOld0;
    output wire [9:0]  comYOld0;
    output wire [10:0] comXOld1;
    output wire [9:0]  comYOld1;
    output wire [10:0] comXOld2;
    output wire [9:0]  comYOld2;
    output wire [10:0] comXOld3;
    output wire [9:0]  comYOld3;
    output wire [10:0] comXOld4;
    output wire [9:0]  comYOld4;

    // 28 bits for x and y accumulators because 28 = ceil(log2(1024*768*2^8))
    wire [27:0]         center_of_mass_div_x;
    wire [27:0]         center_of_mass_div_y;
    reg [27:0]          H_sum_x;
    reg [27:0]          H_sum_y;

    // appropriate for x in range [0, 1024] and y in range [0, 768]
    wire [10:0]         center_of_mass_x;
    wire [9:0]          center_of_mass_y;

    // pixel count accumulator is 21 bits = ceil(log2(1024*768))
    reg [20:0]          used_pixel_count;
    reg [9:0]          prev_detected;
    reg [20:0]          prev_detected_count;
    reg [20:0]          frameCount;
    // OLD_POINTS = 3
    // [0 1 2 3]
    //        ^ frameCount
    // [0 1 2 3]
    //  ^ frameCount
    //
    assign comXOld0 = xDelayLine[(frameCount + OLD_POINTS) % (OLD_POINTS + 1)];
    assign comYOld0 = yDelayLine[(frameCount + OLD_POINTS) % (OLD_POINTS + 1)];

    assign comXOld1 = xDelayLine[(frameCount + 12) % (OLD_POINTS + 1)];
    assign comYOld1 = yDelayLine[(frameCount + 12) % (OLD_POINTS + 1)];

    assign comXOld2 = xDelayLine[(frameCount + 9) % (OLD_POINTS + 1)];
    assign comYOld2 = yDelayLine[(frameCount + 9) % (OLD_POINTS + 1)];

    assign comXOld3 = xDelayLine[(frameCount + 6) % (OLD_POINTS + 1)];
    assign comYOld3 = yDelayLine[(frameCount + 6) % (OLD_POINTS + 1)];

    assign comXOld4 = xDelayLine[(frameCount + 3) % (OLD_POINTS + 1)];
    assign comYOld4 = yDelayLine[(frameCount + 3) % (OLD_POINTS + 1)];

    always @(posedge clk) begin
      // nearest neighbor stuff

      if (reset) begin // do initialization
     used_pixel_count <= 0;
     H_sum_x <= 0;
     H_sum_y <= 0;
     comX <= 0;
     comY <= 0;
     frameCount <= 0;
      end
      else if (x == 0 && y == 0) begin // next frame, latch output
     if (frameCount == 0)
       frameCount <= OLD_POINTS - 1;
     else
       frameCount <= frameCount - 1;
     xDelayLine[frameCount] <= comX;
     yDelayLine[frameCount] <= comY;
     if (extrapolate) begin
        comX <= ((center_of_mass_x - minX) << 1); //* 2;
```

```
      comY <= ((center_of_mass_y - minY) << 1); //* 2;
    end
    else begin
      comX <= ((center_of_mass_x));
      comY <= ((center_of_mass_y));
    end
    used_pixel_count <= 0;
    H_sum_x <= 0;
    H_sum_y <= 0;
     end
     else if (inframe && detected) begin  // ok b/c pixel is sync'd with clk
    if (prev_detected[9:0] == 10'b1111111111) begin
      used_pixel_count <= used_pixel_count + 1;
      H_sum_x <= H_sum_x + x;
      H_sum_y <= H_sum_y + y;
      used <= 1;
    end
    else begin
      used <= 0;
    end
    prev_detected = {prev_detected[8:0], detected};
     end
  end // always @ (posedge clk)

  wire rfdA;
  wire rfdB;

  div mydivX(.dividend(H_sum_x), .divisor(used_pixel_count),
       .quot(center_of_mass_div_x), .clk(clk), .rfd(rfdA));

  div mydivY(.dividend(H_sum_y), .divisor(used_pixel_count),
       .quot(center_of_mass_div_y), .clk(clk), .rfd(rfdB));

  assign center_of_mass_x = rfdA ? center_of_mass_div_x[10:0] : center_of_mass_x;
  assign center_of_mass_y = rfdB ? center_of_mass_div_y[9:0] : center_of_mass_y;

endmodule // CenterOfMass
```

Center of Mass Testbench Verilog Code

```
// test_center_of_mass.v
// 6.111 final project
// By Isaac Evans, ine@mit.edu
//Testbench for center of mass module

module test_center_of_mass;

   // Inputs
   reg clk;
   reg reset;
   reg [18:0] ntsc_address;
   reg         ntsc_we;
   reg [10:0] x;
   reg [9:0]  y;
   reg [7:0]  H;
   reg [7:0]  S;
   reg [7:0]  V;
   reg [7:0]  target_hue_low;
   reg [7:0]  target_hue_high;

   // Outputs
   wire [10:0] comX;
   wire [9:0]  comY;

   integer     fin, code;

   // Instantiate the Unit Under Test (UUT)
   CenterOfMass uut (
         .clk(clk),
         .reset(reset),
         .ntsc_address(ntsc_address),
         .ntsc_we(ntsc_we),
         .x(x),
         .y(y),
         .H(H),
         .S(S),
         .V(V),
         .target_hue_low(target_hue_low),
         .target_hue_high(target_hue_high),
         .comX(comX),
         .comY(comY)
         );

   initial begin
      fin = $fopen("h.jpg","r");
      if (fin == 0) begin
     $display("can't open file...");
     $stop;
      end

      // Initialize Inputs
      clk = 0;
      reset = 0;
      ntsc_address = 0;
      ntsc_we = 0;
      x = 0;
      y = 0;
      H = 0;
      S = 0;
      V = 0;
      target_hue_low = 3;
      target_hue_high = 15;

      // Wait 100 ns for global reset to finish
      #100;
      reset = 0;
      #10;
      reset = 1;
        #10;
        reset = 0;
   end // initial begin
```

```
always #5 clk = ~clk;

always @(posedge clk) begin
   code = $fscanf(fin,"%d", H);
   if (code != 1) begin
  $fclose(fin);
  $stop;
   end
   //sat = $fscanf(fin,"%d",x);
   //val = $fscanf(fin,"%d",x);

   ntsc_address <= ntsc_address + 1;
   ntsc_we = 1;

   if (x == 1024) begin
 x <= 0;
 y <= y + 1;
   end
   else if (y == 768) begin
  $stop;
   end
   else begin
 x <= x + 1;
   end

   /*
   if (cycle == 6'd63) begin
  // assert ready next cycle, read next sample from file
  ready <= 1;
  code = $fscanf(fin,"%d",x);
  // if we reach the end of the input file, we're done
  if (code != 1) begin
         $fclose(fout);
         $stop;
  end
   end
```

```
     else begin
    ready <= 0;
     end

     if (ready) begin
    // starting with sample 32, record results in output file
    if (scount > 31) $fdisplay(fout,"%d",y);
    scount <= scount + 1;
     end

     cycle <= cycle+1;
      */
  end

endmodule
```

Geto Cut Detector Verilog Code

```
// geto_cut_detector.v
// 6.111 final project
// By Isaac Evans, ine@mit.edu
//Detects if fruit has been cut based on hand vectors and fruit positions
module geto_cut_detector (
          input       clock,
          input       reset,
          input [9:0]      sp0y, //sprite Y positions
          input [9:0]      sp1y,
          input [9:0]      sp2y,
          input [9:0]      sp3y,
          input [9:0]      sp4y,
          input [9:0]      sp5y,
          input [10:0]     sp0x, //sprite X positions
          input [10:0]     sp1x,
          input [10:0]     sp2x,
          input [10:0]     sp3x,
          input [10:0]     sp4x,
          input [10:0]     sp5x,
        input [9:0]      b0y,
        input [9:0]      b1y,
        input [10:0]      b0x,
        input [10:0]      b1x,
        input [5:0]      spon,
        input [1:0]      bombon,
        //      input [2:0]     linemaker,
        input [10:0]      com_x_old0,
        input [9:0]      com_y_old0,
        input [10:0]      com_x_old1,
        input [9:0]      com_y_old1,
        input [10:0]      com_x_old2,
        input [9:0]      com_y_old2,
        input [10:0]      com_x_old3,
        input [9:0]      com_y_old3,
        input [10:0]      com_x_old4,
        input [9:0]      com_y_old4,
        output reg [5:0] cut,
        output reg [1:0] bombcut
        );

wire bomb0A;
wire bomb1A;

wire sp5A;
wire sp4A;
wire sp3A;
wire sp2A;
wire sp1A;
wire sp0A;

always @(posedge clock) begin
  if (cut[5]) cut[5] <= 0;
  if (cut[4]) cut[4] <= 0;
  if (cut[3]) cut[3] <= 0;
  if (cut[2]) cut[2] <= 0;
  if (cut[1]) cut[1] <= 0;
```

```
    if (cut[0]) cut[0] <= 0;
    if (bombcut[1]) bombcut[1] <= 0;
    if (bombcut[0]) bombcut[0] <= 0;

    if (bombon[0] && bomb0A) bombcut[0] <= 1;
    if (bombon[1] && bomb1A) bombcut[1] <= 1;

    if (spon[5] && sp5A) cut[5] <= 1;
    if (spon[4] && sp4A) cut[4] <= 1;
    if (spon[3] && sp3A) cut[3] <= 1;
    if (spon[2] && sp2A) cut[2] <= 1;
    if (spon[1] && sp1A) cut[1] <= 1;
    if (spon[0] && sp0A) cut[0] <= 1;

    /*
     wire spPROTOA; wire spPROTOB;
     isInside iiPROTOA(hand1X, hand1Y, spPROTOx, spPROTOy, 10'd64, 10'd64,
spPROTOA);
     isInside iiPROTOB(hand2X, hand2Y, spPROTOx, spPROTOy, 10'd64, 10'd64,
spPROTOB);
     always @(posedge clock) begin
     if (spon[PROTO] && (spPROTOA || spPROTOB)) cut[PROTO] <= 1;
     end
     */
  end

  // oldest vector
  isVectorCut iz0A(com_x_old0, com_x_old1, com_x_old2, com_x_old3, com_x_old4,
        com_y_old0, com_y_old1, com_y_old2, com_y_old3, com_y_old4,
        sp0x, sp0y, 10'd64, 10'd64, sp0A);

  isVectorCut iz1A(com_x_old0, com_x_old1, com_x_old2, com_x_old3, com_x_old4,
        com_y_old0, com_y_old1, com_y_old2, com_y_old3, com_y_old4,
        sp1x, sp1y, 10'd64, 10'd64, sp1A);

  isVectorCut iz2A(com_x_old0, com_x_old1, com_x_old2, com_x_old3, com_x_old4,
        com_y_old0, com_y_old1, com_y_old2, com_y_old3, com_y_old4,
        sp2x, sp2y, 10'd64, 10'd64, sp2A);

  isVectorCut iz3A(com_x_old0, com_x_old1, com_x_old2, com_x_old3, com_x_old4,
        com_y_old0, com_y_old1, com_y_old2, com_y_old3, com_y_old4,
        sp3x, sp3y, 10'd64, 10'd64, sp3A);

  isVectorCut iz4A(com_x_old0, com_x_old1, com_x_old2, com_x_old3, com_x_old4,
        com_y_old0, com_y_old1, com_y_old2, com_y_old3, com_y_old4,
        sp4x, sp4y, 10'd64, 10'd64, sp4A);

  isVectorCut iz5A(com_x_old0, com_x_old1, com_x_old2, com_x_old3, com_x_old4,
        com_y_old0, com_y_old1, com_y_old2, com_y_old3, com_y_old4,
        sp5x, sp5y, 10'd64, 10'd64, sp5A);

  isVectorCut iz6A(com_x_old0, com_x_old1, com_x_old2, com_x_old3, com_x_old4,
        com_y_old0, com_y_old1, com_y_old2, com_y_old3, com_y_old4,
        b0x, b0y, 10'd64, 10'd64, bomb0A);

  isVectorCut iz7A(com_x_old0, com_x_old1, com_x_old2, com_x_old3, com_x_old4,
        com_y_old0, com_y_old1, com_y_old2, com_y_old3, com_y_old4,
        b1x, b1y, 10'd64, 10'd64, bomb1A);

endmodule //geto_cut_detector
```

Is Vector Cut Verilog Code

```
// is_vector_cut.v
// 6.111 final project
// By Isaac Evans, ine@mit.edu
//determines if a vector is cutting a single sprite based on game rules

module isVectorCut(x0, x1, x2, x3, x4, y0, y1, y2, y3, y4,
         sprite_x, sprite_y, sprite_height, sprite_width, isCut);
  input wire [10:0] x0;
  input wire [10:0] x1;
  input wire [10:0] x2;
  input wire [10:0] x3;
  input wire [10:0] x4;
  input wire [9:0]  y0;
  input wire [9:0]  y1;
  input wire [9:0]  y2;
  input wire [9:0]  y3;
  input wire [9:0]  y4;
  output wire        isCut;
  input wire [10:0] sprite_x;
  input wire [9:0]  sprite_y;

  input wire [9:0]  sprite_height;
  input wire [9:0]  sprite_width;

  isInside i1 (x0, y0, sprite_x, sprite_y, sprite_height, sprite_width, in0);
  isInside i2 (x1, y1, sprite_x, sprite_y, sprite_height, sprite_width, in1);
  isInside i3 (x2, y2, sprite_x, sprite_y, sprite_height, sprite_width, in2);
  isInside i4 (x3, y3, sprite_x, sprite_y, sprite_height, sprite_width, in3);
  isInside i5 (x4, y4, sprite_x, sprite_y, sprite_height, sprite_width, in4);

  // vector
  //    in0 .... in1 ... in2 ... in3 ... in4 (oldest to newest)
  // want at least one of (in1, in2, in3) true
  // and at least one of (in0, in4) NOT true

  // harder
  assign isCut = (! in0 || ! in4) && (in1 || in2 || in3);
  // easier
  //assign isCut = (in0 || in4 || in1 || in2 || in3);

endmodule
```

Is Inside Verilog Code

```
// is_inside.v
// 6.111 final project
// By Isaac Evans, ine@mit.edu
//detects if a given point is within the sprite's area.
```

```
module isInside(x, y, sprite_x, sprite_y, sprite_height, sprite_width, isInside);
  input wire [10:0] x;
  input wire [9:0]  y;

  input wire [10:0] sprite_x;
  input wire [9:0]  sprite_y;

  input wire [9:0]  sprite_height;
  input wire [9:0]  sprite_width;
  output wire        isInside;

  assign isInside = (x >= (sprite_x - sprite_width) &&
           x <= (sprite_x + sprite_width) &&
           y >= (sprite_y - sprite_height) &&
           y <= (sprite_y + sprite_height));
endmodule
```

Geto Randomizer Verilog Code

```
// geto_randomizer.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
//generates pseudorandom numbers based on hand position

module geto_randomizer(input [10:0] xpos, input [9:0] ypos, output [9:0] rando);
assign rando = {xpos[4], ypos[4], xpos[3], ypos[3], xpos[2], ypos[2], xpos[1], ypos[1], xpos[0],
ypos[0]};
endmodule //geto_randomizer
```

Game Logic Verilog Code

```
// game_logic.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
//core game rules. Keeps track of score, lives, levels. Brings sprites into and out of the game.
```

```
module game_logic(
   input clock,
   input reset,
   input [5:0] cut, //up to 6 fruits simultaneously
   input [1:0] bombcut, //up to 2 bombs simultaneously
   input [9:0] sp0y, //sprite Y positions
   input [9:0] sp1y,
   input [9:0] sp2y,
   input [9:0] sp3y,
   input [9:0] sp4y,
   input [9:0] sp5y,
   input [9:0] b0y, //bomb Y positions
   input [9:0] b1y,

   output reg [5:0] spon, //sprite on
   output reg [1:0] bombon, //bomb on
   output reg [3:0] level, //8 levels plus an end level
   output reg [8:0] score, //max score of 512
   output reg [2:0] lives, //5 lives
   output reg gameon //1 bit signal for game state
);
//Parameters
parameter lv_1_delay = 4'd10; //10 half-seconds (5sec) between fruits for level 1

parameter lv_2_score = 9'd5; //5 points to get to level 2
parameter lv_2_delay = 4'd8; //8 half-seconds (4sec) between fruits for level 2

parameter lv_3_score = 9'd15; //15 points to get to level 3
parameter lv_3_delay = 4'd8; //8 half-seconds (4sec) between fruits for level 3

parameter lv_4_score = 9'd30; //30 points to get to level 4
parameter lv_4_delay = 4'd6; //6 half-seconds (3sec) between fruits for level 4
parameter bomb_delay_4 = 4'd7; //7 fruits per bomb

parameter lv_5_score = 9'd50; //50 points to get to level 5
parameter lv_5_delay = 4'd4; //4 half-seconds (2sec) between fruits for level 5
parameter bomb_delay_5 = 4'd6; //6 fruits per bomb
parameter lv_6_score = 9'd85; //85 points to get to level 6
parameter lv_6_delay = 4'd3; //3 half-seconds (1.5sec) between fruits for level 6
parameter bomb_delay_6 = 4'd6; //6 fruits per bomb

parameter lv_7_score = 9'd125; //125 points to get to level 7
parameter lv_7_delay = 4'd2; //2 half-seconds (1sec) between fruits for level 7
parameter bomb_delay_7 = 4'd5; //5 fruits per bomb

parameter lv_8_score = 9'd200; //200 points to get to level 8
parameter lv_8_delay = 4'd1; //1 half-seconds (0.5sec) between fruits for level 8
parameter bomb_delay_8 = 4'd4; //4 fruits per bomb

parameter win_score = 9'd400; //400 points to win the game

reg [3:0] timerval;
reg start_timer;
reg prevreset;
wire half_hz_enable, expired;
reg bomb_go;
reg [3:0] bomb_delay;
reg [3:0] bomb_counter;
reg bomb_ready;
reg [7:0] waitone;

initial begin
   waitone <= 0;
   timerval <= lv_1_delay;
   spon <= 6'd0;
   level <= 4'd0;
   score <= 9'd0;
   lives <= 3'b101;
   gameon <= 0;
   prevreset <= 0;
   start_timer <= 0;
   bomb_go <= 0;
   bomb_ready <= 0;
   bomb_delay <= bomb_delay_4; //bomb delay is the number of fruits that get sent for each
bomb
   bomb_counter <= 0; //this counts number of fruits sincs last bomb. Reloaded with
bomb_delay.
end //initial

always @(posedge clock) begin

   if (level == 4'd0) begin //start screen logic
      if (cut[0] || cut[1]) begin
         gameon <= 1'b1;
         level <= 4'd1;
         start_timer <= 1'b1;
      end //cut 0 or cut 1
   end

   if (gameon) begin
   if (start_timer) start_timer <= 0; //start timer only asserted for one cycle

////////////Fruit starting logic

   if (level == 4'd1) begin //level 1
      timerval <= lv_1_delay; //fruits come more frequently at higher levels
      if (score == lv_2_score) level <= 4'd2; //once you hit a certain score, go to the next level
   end //level 1

   if (level == 4'd2) begin //level 2
      timerval <= lv_2_delay;
      if (!bomb_go) begin
      bomb_go <= 1; //start sending bombs at level 4
      bomb_counter <= bomb_delay;
      end //if !bombgo

      if (score == lv_3_score) level <= 4'd3;
   end //level 2

   if (level == 4'd3) begin //level 3
```

```
        timerval <= lv_3_delay;
        if (score == lv_4_score) level <= 4'd4;
    end //level 3

    if (level == 4'd4) begin //level 4
        timerval <= lv_4_delay;
/*        if (!bomb_go) begin
        bomb_go <= 1; //start sending bombs at level 4
        bomb_counter <= bomb_delay;
        end //if !bombgo
*/        if (score == lv_5_score) level <= 4'd5;
    end //level 4

    if (level == 4'd5) begin //level 5
        timerval <= lv_5_delay;
        bomb_delay <= bomb_delay_5;
        if (score == lv_6_score) level <= 4'd6;
    end //level 5

    if (level == 4'd6) begin //level 6
        timerval <= lv_6_delay;
        bomb_delay <= bomb_delay_6;
        if (score == lv_7_score) level <= 4'd7;
    end //level 6

    if (level == 4'd7) begin //level 7
        timerval <= lv_7_delay;
        bomb_delay <= bomb_delay_7;
        if (score == lv_8_score) level <= 4'd8;
    end //level 7

    if (level == 4'd8) begin //level 8
        timerval <= lv_8_delay;
        bomb_delay <= bomb_delay_8;
        if (score == win_score) level <= 4'd9;
    end //level 8
    if (level == 4'd9) begin //endgame state
        spon <= 6'd0;
        gameon <= 0;
    end

    if (expired) begin //time to send a new sprite
        if (bomb_go) begin
            bomb_counter <= bomb_counter - 1'b1;
            if (bomb_counter == 4'd0) begin //time to send a new bomb
                bomb_ready <= 1;
                bomb_counter <= bomb_delay;
            end //if bomb counter expired
        end //if bomb go

        if (!(level == 4'd9) || !(level == 4'd0)) start_timer <= 1'b1; //start the timer for another sprite
if the game isn't won or hasn't started
        if (!(spon[0])) begin
        spon[0] <= 1'b1;
        waitone[0] <= 0;
        end
        else begin
            if (!(spon[1])) begin
            waitone[1] <= 0;
            spon[1] <= 1'b1;
            end
            else begin
                if ((!spon[2])) begin
                spon[2] <= 1'b1;
                waitone[2] <= 1'b0;
                end
                else begin
                    if ((!spon[3])) begin
                    spon[3] <= 1'b1;
                    waitone[3] <= 1'b0;
                    end
                    else begin
                        if ((!spon[4])) begin
            spon[4] <= 1'b1;
            waitone[4] <= 1'b0;
            end
            else begin
            spon[5] <= 1'b1;
            waitone[5] <= 1'b0;
            end
          end
        end
      end
    end


  end //if expired

///////////////Fruit ending logic

  if (spon[0]) begin
    waitone[0] <= 1'b1;
    if (waitone[0]) begin
    if (cut[0]) begin //if cut, increment score and turn off that fruit
      spon[0] <= 1'b0;
      score <= score + 1;
    end //cut logic

    if (sp0y > 767) begin //if you miss the fruit, decrement lives and turn it off
      spon[0] <= 1'b0;
      lives <= lives - 1;
    end //life lost logic
    end
  end //sprite 0 logic

  if (spon[1]) begin
    waitone[1] <= 1'b1;
    if (waitone[1]) begin
    if (cut[1]) begin
      spon[1] <= 1'b0;
```

```
        score <= score + 1;
    end //cut logic

    if (sp1y > 767) begin
        spon[1] <= 1'b0;
        lives <= lives - 1;
    end //life lost logic
    end
end //sprite 1 logic

if (spon[2]) begin
    waitone[2] <= 1'b1;
    if (waitone[2]) begin
    if (cut[2]) begin
        spon[2] <= 1'b0;
        score <= score + 1;
    end //cut logic

    if (sp2y > 767) begin
        spon[2] <= 1'b0;
        lives <= lives - 1;
    end //life lost logic
    end
end //sprite 2 logic

if (spon[3]) begin
    waitone[3] <= 1'b1;
    if (waitone[3]) begin
    if (cut[3]) begin
        spon[3] <= 1'b0;
        score <= score + 1;
    end //cut logic

    if (sp3y > 767) begin
        spon[3] <= 1'b0;
        lives <= lives - 1;
    end //life lost logic
```

```
        end
    end //sprite 3 logic

    if (spon[4]) begin
        waitone[4] <= 1'b1;
        if (waitone[4]) begin
        if (cut[4]) begin
            spon[4] <= 1'b0;
            score <= score + 1;
        end //cut logic

        if (sp4y > 767) begin
            spon[4] <= 1'b0;
            lives <= lives - 1;
        end //life lost logic
        end
    end //sprite 4 logic

    if (spon[5]) begin
        waitone[5] <= 1'b1;
        if (waitone[5]) begin
        if (cut[5]) begin
            spon[5] <= 1'b0;
            score <= score + 1;
        end //cut logic

        if (sp5y > 767) begin
            spon[5] <= 1'b0;
            lives <= lives - 1;
        end //life lost logic
        end
    end //sprite 5 logic

//////////////////////bomb logic
    if (bombon[0]) begin
        waitone[6] <= 1'b1;
        if (waitone[6]) begin
        if (bombcut[0]) begin
            bombon[0] <= 1'b0;
            lives <= 0;
        end //cut logic

        if (b0y > 767) begin
            bombon[0] <= 1'b0;
        end //bomb ending logic
        end
    end //bomb0 logic

    if (bombon[1]) begin
        waitone[7] <= 1'b1;
        if (waitone[7]) begin
        if (bombcut[1]) begin
            bombon[1] <= 1'b0;
            lives <= 0;
        end //cut logic

        if (b1y > 767) begin
            bombon[1] <= 1'b0;
        end //life lost logic
        end
    end //bomb1 logic

////////////////
    if (bomb_ready) begin
    if (half_hz_enable) begin
        bomb_ready <= 0;
        if ((!bombon[0])) begin
        bombon[0] <= 1;
        waitone[6] <= 1'b0;
        end
        else begin
        bombon[1] <= 1;
        waitone[7] <= 1'b0;
        end
    end //if half hz enable
    end //if bomb ready

if (lives == 4'd0) begin
    gameon <= 0;
end //game over scenario

end //gameon
/*if (!(reset) && prevreset) begin
    start_timer <= 1;
    gameon <= 1;
    level <= 1;
    prevreset <= 0;
end //letting go of reset, start the game
*/
    if (reset) begin
        start_timer <= 0;
        timerval <= lv_1_delay;
        spon <= 6'd0;
        score <= 9'd0;
        lives <= 3'b101;
        level <= 0;
        gameon <= 0;
        //prevreset <= 1;
        bombon <= 0;
        bomb_go <= 0;
        bomb_ready <= 0;
        bomb_delay <= bomb_delay_4; //bomb delay is the number of fruits that get sent for each
bomb
        bomb_counter <= 0; //this counts number of fruits sincs last bomb. Reloaded with
bomb_delay.
        waitone <= 0;

    end //reset

    if (!gameon) begin
        spon <= 6'd0;
        bombon <= 2'd0;
    end //if not game on
end //always block

time_divider div(clock, start_timer, reset, half_hz_enable);
timer tim(clock, half_hz_enable, start_timer, reset, timerval, expired);


endmodule //game_logic
```

Game Logic Testbench Verilog Code

```
// game_logic_tb.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
//tests game logic module

module game_logic_tb; //testbench for game logic module

reg clock, reset;
reg [3:0] cut;
reg [9:0] sp0y;
reg [9:0] sp1y;
reg [9:0] sp2y;
reg [9:0] sp3y;
wire [3:0] spon;
wire [3:0] level;
wire [8:0] score;
wire [2:0] lives;
wire gameon;

initial
  begin
    clock = 0;
    reset = 1;
    cut = 0;
    sp0y = 10'd500;
    sp1y = 10'd500;
    sp2y = 10'd500;
    sp3y = 10'd500;
    #50;
    reset = 0;
    #325; //all four should be activated at this point
    cut [3:0] = 4'd1; //should turn off sprite 0 and increment score
    #10
    sp0y = 769; //should do nothing
    #10
    sp1y = 769; //should reduce number of lives and turn off sprite 1
    #10
    cut [3:0] = 4'b1100; //should increment score by 2, turn off sprites 2 and 3
    #5
    cut [3:0] = 4'b0000;
    sp0y = 10'd500;
    #60; //sprite 0 should turn back on at 55ns into this
    cut[0] = 1; //increase score, turn off sprite 1
    #40
    cut[0] = 0;
    #40
    cut[0] = 1; //increase score, turn off sprite 1
    #40
    cut[0] = 0;
    #40
    cut[0] = 1; //increase score, turn off sprite 1 (should be in level 2 by now)
    #40
    cut[0] = 0;
    sp0y = 10'd800;
    sp1y = 10'd800;
    sp2y = 10'd800;
    sp3y = 10'd800;    //should eventually end the game with lost lives


  end //initial

  always #(1) clock = ~clock; //2ns clock period
```

```
game_logic dut(clock, reset, cut, sp0y, sp1y, sp2y, sp3y, spon, level, score, lives, gameon);
endmodule //game logic testbench
```

Time Divider Verilog Code

```
// time_divider.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
//outputs signal at a given frequency based on countval parameter

module time_divider (input clock, Start_Timer, reset, output reg half_hz_enable);
  //parameter countval = 25'd13_500_000;
  parameter countval = 25'd5_000_000; //FOR DEBUG TESTING timer = 5 for 10ns / 0.5s

  reg [24:0] count = countval;

  always @(posedge clock) begin

    count <= count-1; //decrement the count
    if (count == 0) begin
      count <= countval; //reset count, set the one_hz signal
      half_hz_enable <= 1'b1;
    end

    if (half_hz_enable == 1) half_hz_enable <= 0; //this is only 1 for one cycle out of
13,500,000

    if (reset | Start_Timer) begin //reset logic
      count <= countval;
      half_hz_enable <= 0;
    end //if reset
  end
endmodule //time_divider
```

Timer Verilog Code

```
// Timer.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
//keeps time based on input values
module timer (input clock, half_hz, start_timer, reset, input [3:0] value, output reg expired);
   reg running = 0; //the timer is running if signal running=1
   reg waitone = 0; //give 1 clock cycle delay
   reg [3:0] curr_count;
   always @(posedge clock) begin
      if (running) begin //if the timer is running
         if (curr_count == 0) begin
            waitone <= 1; //one cycle delay for timing reasons
            if (waitone) begin
            expired <= 1'b1; //after the delay, set the expired signal
            running <= 1'b0;
            end
         end //curr_count=0

         else begin
            if (half_hz) curr_count <= (curr_count-1);
         end //else

      end //if running

      if (~running & start_timer) begin
         running <= 1'b1; //start timer if it's not running and the start signal is asserted
         curr_count <= value; //load the count with input value
      end //not running

      if (expired) expired <= 1'b0; //expired only asserted for one clock cycle
      if (reset) begin
         expired <= 1'b0;
         running <= 1'b0;
      end //reset
   end //always block
```

```
endmodule //timer
```

Sprite Logic Verilog Code

```
// sprite_logic.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
//calculates sprite positions based on game physics

module sprite_logic(
   input clock,
   input reset,
   input vsync, //vsync of video signal
   input on, //sprite on
   input [9:0] rando,
   output reg [9:0] ypos, //sprite Y position
   output reg [9:0] xpos, //sprite x position
   output reg syncstate //sprite on synced to hsync
);
parameter gravity = 6'd22; //enter as integer. Gravity will be this number divided by 64 (frames
per second per second).
reg state;
reg [5:0] floatgrav; //used for fractional gravity
reg signed [11:0] xvector; //x component of fruit's vector
reg signed [10:0] yvector; //y component of fruit's vector max 63
reg prevvsync;

initial begin
   xpos <= 0;
   ypos <= 0;
   state <= 0;
   xvector <= 0;
   yvector <= 0;
   floatgrav <= 0;
   syncstate <= 0;
end //initial
```

```
always @(posedge clock) begin
   if (!state) begin
   ypos <= 10'd766;
   xpos <= 10'd500;
   xvector = 12'd0;
   yvector <= -10;
   end
   if ((!state) && on) begin  //indicates new sprite
      if (rando[6] && rando[4]) begin //25% of the time the fruit falls downwards from the top
      ypos <= 10'd1;
      yvector <= 1;
      end

      else begin
         ypos <= 10'd766; //otherwise it starts at the bottom, moving upwards between -26 and -32 
pix/frame upwards
         //if (rando[0]) yvector <= -29 + rando[2:1];
         //if (!rando[0]) yvector <=  -29 - rando[2:1];
         yvector <= -22;
         if (rando[0]) yvector <= -18 + rando[2:1];
         if (!rando[0])yvector <= -18 - rando[2:1];
      end
      xpos <= rando;
      case (rando[9:7]) //values selected based on X position so it never wraps around the screen
         3'b000 : xvector = {9'd0,rando[2:0]} + {11'd0,rando[4]} + {11'd0,rando[5]}; //max of 7 + 
1 + 1 = 9
         3'b001: xvector = {9'd0, rando[2:0]} + {11'd0,rando[4]}; //max of 7 + 1 = 8
         3'b010: xvector = {9'd0, rando[2:0]}; //max of 7
         3'b011: xvector = {10'd0,rando[1:0]} + {11'd0,rando[2]} + {11'd0,rando[3]}; //max of 3 + 
1 + 1 = 5
         3'b100: xvector = -1 * ({10'd0,rando[1:0]} + {11'd0,rando[2]} + {11'd0,rando[3]}); //max 
of 3 + 1 + 1 = -5
         3'b101: xvector = -1 * ({9'd0, rando[2:0]}); //max of -7
         3'b110: xvector = -1 * ({9'd0, rando[2:0]} + {11'd0,rando[4]}); //max of 7 + 1 = -8
         3'b111: xvector = -1 * ({9'd0,rando[2:0]} + {11'd0,rando[4]} + {11'd0,rando[5]}); //max 
of 7 + 1 + 1 = -9
```

```
/*
            3'b000 : xvector = {10'd0,rando[1:0]} + {10'd0,rando[3:2]}; //max of 7 + 1 + 1 = 9
            3'b001: xvector = {10'd0, rando[1:0]} + {11'd0,rando[2]} + {11'd0,rando[3]}; //max of 7
+ 1 = 8
            3'b010: xvector = {10'd0, rando[1:0]} + {11'd0,rando[2]} + {11'd0,rando[3]}; //max of 7
            3'b011: xvector = {10'd0,rando[1:0]} + {11'd0,rando[2]}; //max of 3 + 1 + 1 = 5
            3'b100: xvector = -1 * ({10'd0,rando[1:0]} + {11'd0,rando[2]}); //max of 3 + 1 + 1 = -5
            3'b101: xvector = -1 * ({10'd0, rando[1:0]} + {11'd0,rando[2]} + {11'd0,rando[3]}); //
max of -7
            3'b110: xvector = -1 * ({10'd0, rando[1:0]} + {11'd0,rando[2]} + {11'd0,rando[3]}); //
max of 7 + 1 = -8
            3'b111: xvector = -1 * ({10'd0,rando[1:0]} + {10'd0,rando[3:2]}); //max of 7 + 1 + 1 = -9
*/
            default: xvector = 0;
        endcase //rando
        state <= 1;
    end //start of new sprite
    if (vsync & !prevvsync) begin
        if (state) begin
            syncstate <= 1'b1; //state synchronized with the vsync signal
            //$display("%d, %d", xpos, ypos); //for debugging
            xpos <= xpos + xvector;
            ypos <= ypos + yvector;
            floatgrav <= floatgrav + gravity;
            if (floatgrav > (floatgrav + gravity)) yvector <= yvector + 1'b1; //this will be true gravity/
64 of the time. Poor man's floating point unit.
                if (!on) begin
                    state <= 0; //turn sprite off
                    syncstate <= 0;
                end
        end //sprite on
    end //vsync and not prevvsync
prevvsync <= vsync;
if (reset) begin
    xpos <= 0;
    ypos <= 0;
    state <= 0;
    yvector <= 0;
    xvector = 0;
    floatgrav <= 0;
    syncstate <= 0;
end //if reset
end //posedge clock

endmodule //sprite_logic
```

Sprite Logic Testbench Verilog Code

```
// sprite_logic_tb.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
//testbench for sprite logic module
module spritelogic_tb; //testbench for the sprite logic
reg clock, reset, vsync, on;
reg [9:0] rando;
wire [9:0] ypos;
wire [10:0] xpos;

initial begin
clock = 0;
vsync = 0;
rando = 10'b1110000110;
on = 0;
reset = 1;
#50
reset = 0;
#50
on = 1;

end //initial

always #(1) clock = ~clock; //2ns clock period
always #(15) vsync = ~vsync; //30ns vsync period
sprite_logic dut(clock, reset, vsync, on, rando, ypos, xpos);


endmodule //sprite logic testbench
```

Game Audio Verilog Code

```
// game_audio.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
///This module runs the game audio and plays sound effects based on game events

module game_audio(
  input wire clock,             // 27mhz system clock
  input wire reset,             // 1 to reset to initial state
  input wire ready,             // 1 when AC97 data is available
  input wire [5:0] cut,
  input wire [1:0] bombcut,
  input wire [2:0] lives,
  input wire [3:0] level,
  input wire [8:0] score,
  output reg [7:0] to_ac97_data    // 8-bit PCM data to headphone
);
parameter AUDIO_IDLE = 1'b0;
parameter AUDIO_PLAYING = 1'b1;
parameter cut_max_addr = 15'd23386; //goes with cutmem5
parameter newlevel_max_addr = 15'd31996; //goes with levelmem1
parameter lifelost_max_addr = 15'd22254; //goes with lifemem1
parameter win_max_addr = 16'd41806; //goes with winmem1
parameter fail_max_addr = 16'd50387; //goes with failmem1
parameter win_score = 9'd400; //score to win the game. Should match the one from game_logic
module.
parameter boom_max_addr = 16'd42128;

reg audiostate; //tells if any audio is playing or not
reg cut_ready; //tells if the corresponding sound is ready to play
reg newlevel_ready;
reg lifelost_ready;
reg win_ready;
reg fail_ready;
reg boom_ready;

reg cutplaying; //tells if the corresponding sound is playing
reg newlevelplaying;
reg lifelostplaying;
reg winplaying;
reg failplaying;
reg boomplaying;

reg [14:0] cut_addr; // address to corresponding memory brams. width based on cutmem5
reg [14:0] newlevel_addr; //width based on levelmem1
reg [14:0] lifelost_addr; //width based on lifemem1
reg [15:0] win_addr; //width based on winmem1
reg [15:0] fail_addr; //width based on failmem1
reg [15:0] boom_addr;

wire [7:0] cut_data; //output from corresponding memories
wire [7:0] newlevel_data;
wire [7:0] lifelost_data;
wire [7:0] win_data;
wire [7:0] fail_data;
wire [7:0] boom_data;

reg [3:0] curr_level; //current level
reg [2:0] curr_lives;
reg [8:0] curr_score;
reg [5:0] cutprev; //Previous state of cut. Used to tell if a new fruit has been cut
reg [1:0] bombcutprev;
initial begin
cut_ready <= 1'b0;
newlevel_ready <= 1'b0;
lifelost_ready <= 1'b0;
```

```
win_ready <= 1'b0;
fail_ready <= 1'b0;
boom_ready <= 1'b0;

cutprev <= 1'b0;
bombcutprev <= 2'b0;
audiostate <= AUDIO_IDLE;
curr_level <= level;
curr_lives <= lives;
curr_score <= score;

cutplaying <= 1'b0;
newlevelplaying <= 1'b0;
lifelostplaying <= 1'b0;
winplaying <= 1'b0;
failplaying <= 1'b0;
boomplaying <= 1'b0;

cut_addr <= 1'b0;
newlevel_addr <= 13'b0;
lifelost_addr <= 13'b0;
win_addr <= 0;
fail_addr <= 0;
boom_addr <= 0;

end //initial

  always @ (posedge clock) begin
   if (audiostate == AUDIO_IDLE) begin //this handles sounds waiting to play and makes sure
only one is playing at a time.
      if (cut_ready) begin
         cut_ready <= 1'b0;
         audiostate <= AUDIO_PLAYING;
         cutplaying <= 1'b1;
      end //if cut ready
      else if (newlevel_ready) begin
         newlevel_ready <= 1'b0;
```

```
            audiostate <= AUDIO_PLAYING;
            newlevelplaying <= 1'b1;
         end //if newlevel_ready

         else if (lifelost_ready) begin
            lifelost_ready <= 1'b0;
            audiostate <= AUDIO_PLAYING;
            lifelostplaying <= 1'b1;
         end //if lifelost_ready

         else if (win_ready) begin
            win_ready <= 1'b0;
            audiostate <= AUDIO_PLAYING;
            winplaying <= 1'b1;
         end //if win_ready

         else if (boom_ready) begin
            boom_ready <= 1'b0;
            audiostate <= AUDIO_PLAYING;
            boomplaying <= 1'b1;
         end //if boom_ready

         else if (fail_ready) begin
            fail_ready <= 1'b0;
            audiostate <= AUDIO_PLAYING;
            failplaying <= 1'b1;
         end //if fail_ready

      end //idle audio state
///////////////////////////turn on ready signals for different sounds based on input
   if((cut[0] && !cutprev[0]) || (cut[1] && !cutprev[1]) || (cut[2] && !cutprev[2]) || (cut[3] && !
cutprev[3]) || (cut[4] && !cutprev[4]) || (cut[5] && !cutprev[5])) cut_ready <= 1; //play the cut
sound if a new fruit has been cut
   cutprev <= cut;

   if ((level > curr_level) && (level != 4'b1)) newlevel_ready <= 1'b1; //play level sound if you
level up
```

```
   curr_level <= level;

   if (lives < curr_lives) begin
      if (lives == 3'd0) fail_ready <= 1'b1; //if you ran out of lives, play fail sound.
      else lifelost_ready <= 1'b1; //otherwise play the life lost sound
   end //life change
   curr_lives <= lives;

   if ((score == win_score) && (score > curr_score)) win_ready <= 1'b1; //if you got the score to
win, play the win sound.
   curr_score <= score;

   if ((bombcut[0] && !bombcutprev[0]) || (bombcut[1] && !bombcutprev[1])) boom_ready <=
1'b1;
   bombcutprev <= bombcut;
////////////////////////////manage output to AC97 based on which sound is playing
   if (audiostate == AUDIO_PLAYING) begin
      if (cutplaying) to_ac97_data <= cut_data;
      if (newlevelplaying) to_ac97_data <= newlevel_data;
      if (lifelostplaying) to_ac97_data <= lifelost_data;
      if (winplaying) to_ac97_data <= win_data;
      if (failplaying) to_ac97_data <= fail_data;
      if (boomplaying) to_ac97_data <= boom_data;
      //put other sounds here
   end //if audio is playing


///////////////////////////handles memory and state for which clip is playing
   if (ready) begin

   ////////////////////////////
   ///Cut Memory
   ////////////////////////////
      if (!cutplaying) cut_addr <= 13'b0;
      if (cutplaying) begin
         cut_addr <= cut_addr + 1'b1;
         if ((cut_addr+1'b1) == cut_max_addr) begin
```

```
            cutplaying <= 1'b0;
            audiostate <= AUDIO_IDLE;
         end //ending cut playing
      end //if cutplaying
///////////////////////////
///New Level Memory
///////////////////////////
      if (!newlevelplaying) newlevel_addr <= 13'b0;
      if (newlevelplaying) begin
         newlevel_addr <= newlevel_addr + 1'b1;
         if ((newlevel_addr + 1'b1) == newlevel_max_addr) begin
            newlevelplaying <= 1'b0;
            audiostate <= AUDIO_IDLE;
         end //ending new level playing
      end //if new level playing

///////////////////////////
///Life Lost Memory
///////////////////////////
      if (!lifelostplaying) lifelost_addr <= 13'b0;
      if (lifelostplaying) begin
         lifelost_addr <= lifelost_addr + 1'b1;
         if ((lifelost_addr + 1'b1) == lifelost_max_addr) begin
            lifelostplaying <= 1'b0;
            audiostate <= AUDIO_IDLE;
         end //ending life lost playing

      end //if life lost playing

///////////////////////////
///Win Memory
///////////////////////////
      if (!winplaying) win_addr <= 16'd0;
      if (winplaying) begin
         win_addr <= win_addr + 1'b1;
         if ((win_addr + 1'b1) == win_max_addr) begin
            winplaying <= 1'b0;
            audiostate <= AUDIO_IDLE;
        end //ending win playing

    end //if win playing

////////////////////////////
///Fail Memory
////////////////////////////
    if (!failplaying) fail_addr <= 16'b0;
    if (failplaying) begin
        fail_addr <= fail_addr + 1'b1;
        if ((fail_addr + 1'b1) == fail_max_addr) begin
            failplaying <= 1'b0;
            audiostate <= AUDIO_IDLE;
        end //ending fail playing

    end //if fail playing

////////////////////////////
///Boom Memory
////////////////////////////
    if (!boomplaying) boom_addr <= 16'b0;
    if (boomplaying) begin
        boom_addr <= boom_addr + 1'b1;
        if ((boom_addr + 1'b1) == boom_max_addr) begin
            boomplaying <= 1'b0;
            audiostate <= AUDIO_IDLE;
        end //ending boom playing

    end //if boom playing


end //if ready


if (reset) begin
    cut_ready <= 1'b0;
```

```
        newlevel_ready <= 1'b0;
        lifelost_ready <= 1'b0;
        win_ready <= 1'b0;
        fail_ready <= 1'b0;
        boom_ready <= 1'b0;

        bombcutprev <= 2'b0;
        cutprev <= 1'b0;
        audiostate <= AUDIO_IDLE;
        curr_level <= level;
        curr_lives <= lives;
        curr_score <= score;

        cutplaying <= 1'b0;
        newlevelplaying <= 1'b0;
        lifelostplaying <= 1'b0;
        winplaying <= 1'b0;
        failplaying <= 1'b0;
        boomplaying <= 1'b0;

        cut_addr <= 1'b0;
        newlevel_addr <= 13'b0;
        lifelost_addr <= 13'b0;
        win_addr <= 0;
        fail_addr <= 0;
        boom_addr <= 0;
     end //if reset
   end //posedge clock

cutmem5 cm(clock, {8'd0}, cut_addr, {1'b0}, cut_data);
levelmem1 newlevel(clock, {8'd0}, newlevel_addr, {1'b0}, newlevel_data);
lifemem1 lostlife(clock, {8'd0}, lifelost_addr, {1'b0}, lifelost_data);
failmem1 fail(clock, {8'd0}, fail_addr, {1'b0}, fail_data);
winmem1 win(clock, {8'd0}, win_addr, {1'b0}, win_data);
boommem1 boom(clock, {8'd0}, boom_addr, {1'b0}, boom_data);
```

```
endmodule  //game_audio
```

UI Wrapper Verilog Code

```
// UI_wrapper.v
// 6.111 final project
// By Nathan Monroe, monroe@mit.edu
//interfaces the sprite generating part of the UI with the rest of the system to provide a splash
screen with functionality to choose fruits or TA's as sprites. Also generates rotating sprites so
every possible sprite is displayed.

module UI_wrapper (
input clock, reset,
input [3:0] level,
input [10:0] hcount,
input [5:0] spon,
input [3:0] rando,
input [1:0] cut,
output reg [2:0] s0, s1, s2, s3, s4, s5, s6, s7,
output reg cheat
);
reg [5:0] prevspon;
reg cheatstate = 0;
reg [2:0] curr_sprite = 0;
always @(posedge clock) begin
if (!cheatstate) begin
   if (cut[0]) begin
          cheatstate <= 1;
          cheat <= 1;
   end //if cut 0

   if (cut[1]) begin
          cheatstate <= 1;
          cheat <= 0;
```

```
    end //if cut 1
end //if not cheat state

    if (level == 4'd0) begin
            s0 <= 3'd3;
            s1 <= 3'd6;

            if (!cut[0] && !cut[1]) begin
                    if (hcount[9:0] > 10'd400) cheat <= 0;
                    else cheat <= 1;
            end //if not cut 0 and not cut 1
    end //if level 0

    else begin
            if (spon[0] && !prevspon[0]) begin
                    s0 <= curr_sprite;
                    if ((curr_sprite + 1) > 3'd5) curr_sprite <= 0;
                    else curr_sprite <= curr_sprite + 1; //cycle through all sprites
            end
            if (spon[1] && !prevspon[1]) begin
                                    s1 <= curr_sprite;
                    if ((curr_sprite + 1) > 3'd5) curr_sprite <= 0;
                    else curr_sprite <= curr_sprite + 1; //cycle through all sprites
            end
            if (spon[2] && !prevspon[2]) begin
                                    s2 <= curr_sprite;
                    if ((curr_sprite + 1) > 3'd5) curr_sprite <= 0;
                    else curr_sprite <= curr_sprite + 1; //cycle through all sprites
            end
                            if (spon[3] && !prevspon[3]) begin
                                    s3 <= curr_sprite;
                    if ((curr_sprite + 1) > 3'd5) curr_sprite <= 0;
                    else curr_sprite <= curr_sprite + 1; //cycle through all sprites
            end
            if (spon[4] && !prevspon[4]) begin
                                    s0 <= curr_sprite;
                    if ((curr_sprite + 1) > 3'd5) curr_sprite <= 0;
            else curr_sprite <= curr_sprite + 1; //cycle through all sprites
        end
        if (spon[5] && !prevspon[5]) begin
                    s5 <= curr_sprite;
            if ((curr_sprite + 1) > 3'd5) curr_sprite <= 0;
            else curr_sprite <= curr_sprite + 1; //cycle through all sprites
        end

   end //if level is not 0
prevspon <= spon;
s6 <= 3'd6;
s7 <= 3'd6;

if (reset) begin
   prevspon <= 0;
   cheatstate <= 0;
end //if reset
end //posedge clock



endmodule
```

Picture Blob Verilog Code

```
// picture_blob.v
// 6.111 final project
// By Drew Dennison and Nathan Monroe, dennison@mit.edu, monroe@mit.edu
//displays a picture at the desired position on the screen
module picture_blob
  #(parameter DIM = 32) // default picture width and height
  (input pixel_clk, cheat,
   input [10:0]     x0,x1,x2,x3,x4,x5,bomb0x,bomb1x,hcount,
   input [9:0]      y0,y1,y2,y3,y4,y5,bomb0y,bomb1y,vcount,
   input [2:0]      s0,s1,s2,s3,s4,s5,s6,s7,
   input [3:0]      level,
   input [3:0]      rando,
   input [5:0]       spon,
   input [1:0]       bombon,
   input          hsync, vsync,
   output reg          phsync, pvsync,
   output reg [23:0] pixel);

  wire [12:0]         image_addr; // num of bits for 32*32*24*7 ROM
  wire [23:0]         image_bits;
  wire [23:0]         cheat_bits;
  wire [23:0]         myimage_bits;
  wire [23:0]         mycheat_bits;

  reg [10:0]          addr_x = 0;
  reg [9:0]          addr_y = 0;
  reg [2:0]          id = 0;
wire [23:0] blob1out;
 wire [23:0] blob2out;

assign cheat_bits = (level==3'd0) ? (mycheat_bits & (blob1out | blob2out)) : mycheat_bits;
assign image_bits = (level==3'd0) ? (myimage_bits & (blob1out | blob2out)) : myimage_bits;
  // note the one clock cycle delay in pixel!

hackblob blob1(.x(11'd303-32),.y(11'd500-32),.hcount(hcount),.vcount(vcount), .level(level),
        .mypixel(blob1out));

hackblob blob2(.x(11'd703-32),.y(11'd500-32),.hcount(hcount),.vcount(vcount), .level(level),
        .mypixel(blob2out));

  always @ (posedge pixel_clk) begin
    phsync <= hsync;
    pvsync <= vsync;

    pixel <= cheat ? image_bits : cheat_bits;

    if (spon[0] &&((hcount >= x0 && hcount < (x0+DIM*2)) &&
    (vcount >= y0 && vcount < (y0+DIM*2)))) begin
   addr_x <= x0;
```

```
addr_y <= y0;
id <= s0;
 end

 else if (spon[1] && ((hcount >= x1 && hcount < (x1+DIM*2)) &&
      (vcount >= y1 && vcount < (y1+DIM*2)))) begin
addr_x <= x1;
addr_y <= y1;
id <= s1;
 end

 else if (spon[2] && ((hcount >= x2 && hcount < (x2+DIM*2)) &&
      (vcount >= y2 && vcount < (y2+DIM*2)))) begin
addr_x <= x2;
addr_y <= y2;
id <= s2;
 end

 else if (spon[3] && ((hcount+10 >= x3 && hcount < (x3+DIM*2)) &&
      (vcount >= y3 && vcount < (y3+DIM*2)))) begin
addr_x <= x3;
addr_y <= y3;
id <= s3;
 end

 else if (spon[4] && ((hcount >= x4 && hcount < (x4+DIM*2)) &&
      (vcount >= y4 && vcount < (y4+DIM*2)))) begin
addr_x <= x4;
addr_y <= y4;
id <= s4;
 end

 else if (spon[5] && ((hcount >= x5 && hcount < (x5+DIM*2)) &&
      (vcount >= y5 && vcount < (y5+DIM*2)))) begin
addr_x <= x5;
addr_y <= y5;
id <= s5;
```

```
      end

      else if (bombon[0] && ((hcount >= bomb0x && hcount < (bomb0x+DIM*2)) &&
           (vcount >= bomb0y && vcount < (bomb0y+DIM*2)))) begin
     addr_x <= bomb0x;
     addr_y <= bomb0y;
     id <= s6;
      end

      else if (bombon[1] && ((hcount >= bomb1x && hcount < (bomb1x+DIM*2)) &&
           (vcount >= bomb1y && vcount < (bomb1y+DIM*2)))) begin
     addr_x <= bomb1x;
     addr_y <= bomb1y;
     id <= s7;
      end

      else
    pixel <= 0;

   end
   // calculate rom address and read the location


   assign image_addr = (1024*id) + ((hcount-addr_x)/2) + ((vcount-addr_y)/2) * DIM;
   sprites rom1(pixel_clk,24'b0, image_addr, 1'b0,myimage_bits);
   ta_sprites rom2(pixel_clk,24'b0, image_addr, 1'b0,mycheat_bits);
endmodule
```

Level Verilog Code

```
// level.v
// 6.111 final project
// By Drew Dennison, dennison@mit.edu
//generates video signals to display level on the UI
module level(input vclock,
        input [10:0]  x,hcount,
        input [9:0]   y,vcount,
      input [2:0]  level, // 0-7
      output [23:0] pixel
   );

  wire [63:0]           cstring;
  wire [23:0] mypixel;
  wire [55:0] text = 56'b01001100011001010111011001100101011011000011101000100000;
  reg [7:0]   asciiLevel;

  always @ (posedge vclock) begin
    case (level)
   0:
    asciiLevel <= 8'b00110000;
   1:
    asciiLevel <= 8'b00110001;
   2:
    asciiLevel <= 8'b00110010;
   3:
    asciiLevel <= 8'b00110011;
   4:
    asciiLevel <= 8'b00110100;
   5:
    asciiLevel <= 8'b00110101;
   6:
    asciiLevel <= 8'b00110110;
   7:
    asciiLevel <= 8'b00110111;
   default:
    asciiLevel <= 8'b00110000; // zero
    endcase // case (level)

  end // always @ (posedge vclock)
  assign cstring = {text,asciiLevel};
  assign pixel = (level==3'd0) ? 24'd0 : mypixel;
  char_string_display
level_text(.vclock(vclock),.hcount(hcount),.vcount(vcount),.pixel(mypixel),.cstring(cstring),.cx(
x),.cy(y));
```

```
endmodule
```

Lives Verilog Code

```
// lives.v
// 6.111 final project
// By Drew Dennison, dennison@mit.edu
//generates proper signals to display number of lives left on the UI
module lives(input vclock,
        input [10:0]  x,hcount,
        input [9:0]   y,vcount,
        input [2:0]   lives, level, // 0-7
        output [23:0] pixel
  );
  wire [23:0] mypixel;
  wire [63:0]          cstring;

  wire [55:0] text = 56'b01001100011010010111011001100101011100110011101000100000;
  reg [7:0]   asciiLives;

  always @ (posedge vclock) begin
    case (lives)
   0:
    asciiLives <= 8'b00110000;
   1:
    asciiLives <= 8'b00110001;
   2:
    asciiLives <= 8'b00110010;
   3:
    asciiLives <= 8'b00110011;
   4:
    asciiLives <= 8'b00110100;
   5:
    asciiLives <= 8'b00110101;
   6:
    asciiLives <= 8'b00110110;
   7:
     asciiLives <= 8'b00110111;
   default:
     asciiLives <= 8'b00110000; // zero
     endcase // case (lives)

  end // always @ (posedge vclock)
  assign cstring = {text,asciiLives};
  assign pixel = (level==3'd0) ? 24'd0 : mypixel;
  char_string_display
lives_text(.vclock(vclock),.hcount(hcount),.vcount(vcount),.pixel(mypixel),.cstring(cstring),.cx(
x),.cy(y));
endmodule
```

Score Verilog Code

```
// score.v
// 6.111 final project
// By Drew Dennison, dennison@mit.edu
//generates proper video signals to display current score on the UI
module score(input vclock, slowclock,
        input [10:0]  x,hcount,
        input [9:0]   y,vcount,
        input [8:0]   score, // 0-400
        output [23:0] pixel,
        input       reset,
         input [2:0] level
        );

  wire [127:0]          cstring; // Score: 000 = 16 chars

  wire [63:0]          text =
63'b01010011011000110110111101110010011001010011101000100000;
  reg [7:0]          asciiHundreds = 0;
  reg [7:0]          asciiTens = 0;
  reg [7:0]          asciiOnes = 0;
  reg [3:0]          hundreds = 0;
  reg [3:0]          tens = 0;
  reg [3:0]          ones = 0;

  reg [8:0]          old_score = 0;
  wire [23:0] mypixel;

   assign pixel = (level == 3'd0) ? 24'd0 : mypixel;
  initial begin
     ones <= 0;
     tens <= 0;
     hundreds <= 0;
  end

  always @ (posedge slowclock) begin
     if(reset) begin
    ones <= 0;
    tens <= 0;
    hundreds <= 0;
     end

     old_score <= score;
     if (score >  old_score) begin
    ones <= ones + 1;
    if (ones >= 9) begin
       ones <= 0;
       tens <= tens + 1;
       if (tens >= 9) begin
          tens <= 0;
          hundreds <= hundreds + 1;
          if (hundreds >= 9) begin
         hundreds <= 0;
         tens <= 0;
         ones <= 0;
          end
       end
    end // if (ones > 9)
    end
    case (hundreds)
  0:
    asciiHundreds <= 8'b00100000; // space
  1:
    asciiHundreds <= 8'b00110001;
  2:
    asciiHundreds <= 8'b00110010;
  3:
    asciiHundreds <= 8'b00110011;
  4:
    asciiHundreds <= 8'b00110100;
  5:
    asciiHundreds <= 8'b00110101;
  6:
    asciiHundreds <= 8'b00110110;
  7:
    asciiHundreds <= 8'b00110111;
  8:
    asciiHundreds <= 8'b00111000;
  9:
    asciiHundreds <= 8'b00111001;
  default:
    asciiHundreds <= 8'b00100000; // space
    endcase // case (hundreds)

    case (tens)
  0:
    asciiTens <= 8'b00100000; // space
  1:
    asciiTens <= 8'b00110001;
  2:
    asciiTens <= 8'b00110010;
  3:
    asciiTens <= 8'b00110011;
  4:
    asciiTens <= 8'b00110100;
```

```
5:
  asciiTens <= 8'b00110101;
6:
  asciiTens <= 8'b00110110;
7:
  asciiTens <= 8'b00110111;
8:
  asciiTens <= 8'b00111000;
9:
  asciiTens <= 8'b00111001;

default:
  asciiTens <= 8'b00100000; // space
  endcase // case (tens)

  case (ones)
0:
  asciiOnes <= 8'b00110000;
1:
  asciiOnes <= 8'b00110001;
2:
  asciiOnes <= 8'b00110010;
3:
  asciiOnes <= 8'b00110011;
4:
  asciiOnes <= 8'b00110100;
5:
  asciiOnes <= 8'b00110101;
6:
  asciiOnes <= 8'b00110110;
7:
  asciiOnes <= 8'b00110111;
8:
  asciiOnes <= 8'b00111000;
9:
  asciiOnes <= 8'b00111001;
  default:
   asciiOnes <= 8'b00110000; // zero
   endcase // case (one)
  end // always @ (posedge vclock)

  assign cstring = {text,asciiHundreds,asciiTens,asciiOnes};
  char_string_display #(.NCHAR(16), .NCHAR_BITS(4))

score_text(.vclock(vclock),.hcount(hcount),.vcount(vcount),.pixel(mypixel),.cstring(cstring),.cx(
x),.cy(y));
endmodule
```

Splash Screen Verilog Code

```
// splash_screen.v
// 6.111 final project
// By Drew Dennison, dennison@mit.edu
//Generates proper video signals for splash screen at the beginning of the game
module splash_screen
  #(parameter WIDTH = 250,
   parameter HEIGHT= 100) // default picture width and height
  (input pixel_clk,
   input [10:0]      x,hcount,
   input [9:0]       y,vcount,
    input [2:0] level,
   output wire [23:0] mypixel);
  wire [14:0]          image_addr; // num of bits for 250*100*1 ROM
  wire          image_bit;
  reg [23:0] pixel;
   assign mypixel = (level == 3'd0) ? pixel : 24'hFFFFFF;
  // note the one clock cycle delay in pixel!
  always @ (posedge pixel_clk) begin

   if ((hcount >= x && hcount < (x+WIDTH*2) &&
        (vcount >= y && vcount < (y+HEIGHT*2)))) begin
    pixel <= image_bit ? 0 : 24'hFFFFFF;
   end
```

```
    else
   pixel <= 24'hFFFFFF;

  end
  // calculate rom address and read the location

  assign image_addr = (hcount-x)/2 + ((vcount-y)/2 * WIDTH)
;
  splash rom1(pixel_clk,24'b0, image_addr, 1'b0,image_bit);
endmodule
```

Sword Verilog Code

```
// sword.v
// 6.111 final project
// By Drew Dennison, dennison@mit.edu
//generates video signals to display sword on the screen at the hand position
module sword
  #(parameter DIM = 64) // default picture width and height
  (input pixel_clk,
   input [10:0]      myx,hcount,
   input [9:0]       y,vcount,
   output reg [23:0] pixel);
  wire [11:0]        image_addr; // num of bits for 64*64*24 ROM
  wire [23:0]        image_bits;
  wire [10:0] x;
   assign x = myx - 11'd64;
  // note the one clock cycle delay in pixel!
  always @ (posedge pixel_clk) begin

    if ((hcount >= x && hcount < (x +DIM)) &&
        (vcount >= y && vcount < (y+DIM))) begin
   pixel <= image_bits;
    end

    else
   pixel <= 0;

  end
  // calculate rom address and read the location


  assign image_addr = (hcount-x) + (vcount-y) * DIM;
  sword_sprite rom1(pixel_clk,24'b0, image_addr, 1'b0,image_bits);
endmodule
```

Top Module Verilog Code

```
// top_module.v
// 6.111 final project
// By Nathan Monroe, Isaac Evans, Drew Dennison, monroe@mit.edu, ine@mit.edu, //
dennison@mit.edu

module top_module(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'b1;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b1;
//assign tv_in_reset_b = 1'b0;
assign tv_in_clock = clock_27mhz;//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

/* change lines below to enable ZBT RAM bank0 */

/*
 assign ram0_data = 36'hZ;
 assign ram0_address = 19'h0;
 assign ram0_clk = 1'b0;
 assign ram0_we_b = 1'b1;
 assign ram0_cen_b = 1'b0;    // clock enable
 */

/* enable RAM pins */

assign ram0_ce_b = 1'b0;
assign ram0_oe_b = 1'b0;
assign ram0_adv_ld = 1'b0;
assign ram0_bwe_b = 4'h0;

/**********/

assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;

//These values has to be set to 0 like ram0 if ram1 is used.
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;

// clock_feedback_out will be assigned by ramclock
// assign clock_feedback_out = 1'b0;  //2011-Nov-10
// 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;

////////////////////////////////////////////////////////////////////////////
// Demonstration of ZBT RAM as video memory

// 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));

//   wire clk = clock_65mhz;  // gph 2011-Nov-10

/*   ////////////////////////////////////////////////////////////////////////////
 // Demonstration of ZBT RAM as video memory

 // use FPGA's digital clock manager to produce a
 // 40MHz clock (actually 40.5MHz)
 wire clock_40mhz_unbuf,clock_40mhz;
 DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));
 // synthesis attribute CLKFX_DIVIDE of vclk1 is 2
 // synthesis attribute CLKFX_MULTIPLY of vclk1 is 3
 // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
 // synthesis attribute CLKIN_PERIOD of vclk1 is 37
 BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));

 wire clk = clock_40mhz;
 */
wire    locked;
//assign clock_feedback_out = 0; // gph 2011-Nov-10
wire clk;
ramclock rc(.ref_clock(clock_65mhz), .fpga_clock(clk),
        .ram0_clock(ram0_clk),
        //.ram1_clock(ram1_clk),   //uncomment if ram1 is used
        .clock_feedback_in(clock_feedback_in),
        .clock_feedback_out(clock_feedback_out), .locked(locked));


// power-on reset generation
wire    power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clk), .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(power_on_reset, clk, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;

// display module for debugging

reg [63:0]    dispdata;
display_16hex hexdisp1(reset, clk, dispdata,
          disp_blank, disp_clock, disp_rs, disp_ce_b,
          disp_reset_b, disp_data_out);
```

```
// generate basic XVGA video signals
wire [10:0]     hcount;
wire [9:0]     vcount;
wire [10:0]     reversed_hcount;
assign reversed_hcount = 11'd1023 - hcount;

wire     hsync,vsync,blank;
xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);

// wire up to ZBT ram

wire [35:0]     vram_write_data;
wire [35:0]     vram_read_data;
wire [18:0]     vram_addr;
wire     vram_we;

wire     ram0_clk_not_used;
zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
    vram_write_data, vram_read_data,
    ram0_clk_not_used,   //to get good timing, don't connect ram_clk to zbt_6111
    ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

// generate pixel value from reading ZBT memory
//wire [17:0] vr_pixel;
wire [17:0]     vr_pixel; // modification for b&w -> color
wire [18:0]     vram_addr1;

vram_display vd1(reset,clk,reversed_hcount,vcount,vr_pixel,
      vram_addr1,vram_read_data);

// ADV7185 NTSC decoder interface code
// adv7185 initialization module
adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
        .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
        .tv_in_i2c_clock(tv_in_i2c_clock),
        .tv_in_i2c_data(tv_in_i2c_data));
```

```
  wire [29:0]     ycrcb;    // video data (luminance, chrominance)
  wire [2:0]     fvh;    // sync for field, vertical, horizontal
  wire      dv;    // data valid

  ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
            .tv_in_ycrcb(tv_in_ycrcb[19:10]),
            .ycrcb(ycrcb), .f(fvh[2]),
            .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

  // convert the output to RGB
  wire [23:0]     rgb;
  YCrCb2RGB ycrcb2rgb( .R(rgb[23:16]), .G(rgb[15:8]), .B(rgb[7:0]), .clk(tv_in_line_clock1),
.rst(reset),
          .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]), .Cb(ycrcb[9:0]));


  // code to write NTSC data to video memory

  wire [18:0]     ntsc_addr;
  wire [35:0]     ntsc_data;
  wire      ntsc_we;

  // extract the 6 hi order bits from the R, G, and B bytes as we pass them in
  ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, {rgb[23:18], rgb[15:10], rgb[7:2]}, //
modification for b&w -> color
          //ycrcb[29:22],
          ntsc_addr, ntsc_data, ntsc_we, 1'b0); //switch[6]);

  // code to write pattern to ZBT memory
  reg [31:0]     count;
  always @(posedge clk) count <= reset ? 0 : count + 1;

  wire [18:0]     vram_addr2 = count[0+18:0];
  //   wire [35:0]     vpat = ( switch[1] ? {4{count[3+3:3],4'b0}}
  //             :
  wire [35:0]     vpat = {4{count[3+4:4],4'b0}};
```

```
// mux selecting read/write to memory based on which write-enable is chosen

//   wire     sw_ntsc = ~switch[7];
wire     sw_ntsc = 1'b1;
//wire     my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank; // modification for b&w -> color
wire     my_we = sw_ntsc ? (reversed_hcount[0]==1'd1) : blank;

wire [18:0]     write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
wire [35:0]     write_data = sw_ntsc ? ntsc_data : vpat;

//   wire     write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//   assign     vram_addr = write_enable ? write_addr : vram_addr1;
//   assign     vram_we = write_enable;

assign     vram_addr = my_we ? write_addr : vram_addr1;
assign     vram_we = my_we;
assign     vram_write_data = write_data;


wire [10:0]     com_x;
wire [9:0]     com_y;
wire [10:0]     com_x_old0;
wire [9:0]     com_y_old0;
wire [10:0]     com_x_old1;
wire [9:0]     com_y_old1;
wire [10:0]     com_x_old2;
wire [9:0]     com_y_old2;
wire [10:0]     com_x_old3;
wire [9:0]     com_y_old3;
wire [10:0]     com_x_old4;
wire [9:0]     com_y_old4;
wire [7:0]     hue;
wire [7:0]     saturation;
wire [7:0]     value;

wire [7:0]     isaac_vga_red;
```

```
  wire [7:0]     isaac_vga_green;
  wire [7:0]     isaac_vga_blue;
  wire [7:0]     vr_vga_red;
  wire [7:0]     vr_vga_green;
  wire [7:0]     vr_vga_blue;

  // select output pixel data

  //reg [7:0]     pixel; // modification for b&w -> color
  reg [17:0]     pixel;
  reg        b,hs,vs;

  //   wire [17:0] delay_pixel;
  //delay the hcount and vcount by 22 clock cycles to match the rgb2hsv delay
  //   delayN #(.NDELAY(30),.SIZE(18)) delayx(.clk(clk), .in(pixel), .out(delay_pixel));
  //   delayN #(.NDELAY(22),.SIZE(10)) delayy(.clk(clk), .in(vcount), .out(vcount_filter));

  // select output pixel data
  assign isaac_vga_red = {pixel[17:12], 2'd1};
  assign isaac_vga_green = {pixel[11:6], 2'd1};
  assign isaac_vga_blue = {pixel[5:0], 2'd1};

  assign vr_vga_red = {vr_pixel[17:12], 2'd1};
  assign vr_vga_green = {vr_pixel[11:6], 2'd1};
  assign vr_vga_blue = {vr_pixel[5:0], 2'd1};

  wire     inframe;
  // was 523 for vcount
  //assign inframe = (hcount >= 80 && hcount <= 719 && vcount >= 76 && vcount <= 565) ||
switch[2];
  //assign inframe = (hcount <= 880 && hcount >= 195 && vcount >= 126 && vcount <= 605) ||
switch[2];
  assign inframe = (hcount <= 880 && hcount >= 200 && vcount >= 126 && vcount <= 605) ||
switch[2];

  wire     detected;
  //assign detected = (hue < 8 || hue > 248) && (saturation > 125) && (value > 120);
  //assign detected = (hue < 2 || hue > 253) && (saturation > 130) && (value > 100);
//   assign detected = switch[7] ? (hue < 8 || hue > 248) && (saturation > 125) && (value >
switch[7:0]) : (hue < 8 || hue > 248) && (saturation > switch[7:0]) && (value > 100);

  reg [7:0]     hue_thresh_high;
  reg [7:0]     hue_thresh_low;
  reg [7:0]     sat_thresh;
  reg [7:0]     val_thresh;

  assign detected = (hue < hue_thresh_low || hue > hue_thresh_high) && (saturation >
sat_thresh) && (value > val_thresh);

  wire     used;
  // if you see lines running down the image, restart the FPGA dev kit
  // completely and the camera

  CenterOfMass com1 (.inframe(inframe), .clk(clk), .reset(reset),
         .x(hcount), .y(vcount), .H(hue),
         .S(saturation), .V(value), .target_hue_low(7'b0),
         .target_hue_high(switch[7:0]), .comX(com_x),
         .comY(com_y), .detected(detected), .used(used),
         .comXOld0(com_x_old0), .comYOld0(com_y_old0),
           .comXOld1(com_x_old1), .comYOld1(com_y_old1),
           .comXOld2(com_x_old2), .comYOld2(com_y_old2),
           .comXOld3(com_x_old3), .comYOld3(com_y_old3),
           .comXOld4(com_x_old4), .comYOld4(com_y_old4),
         .minX(11'd245), .minY(10'd176), .extrapolate(switch[4]));

  rgb2hsv
myrgb2hsv(.clock(clk), .reset(reset), .r(vr_vga_red), .g(vr_vga_green), .b(vr_vga_blue), .h(hue), .
s(saturation), .v(value));

  wire     clock_250_clock;
  five_time_divider ftd(clk, reset, clock_250_clock);

  initial begin
  hue_thresh_low <= 8'h09;
  hue_thresh_high <= 8'hec;
  sat_thresh <= 8'hb7;
  val_thresh <= 8'h22;
  val_thresh <= 8'h22;
end

always @(posedge clock_250_clock) begin
   if (!button3 && !button_left)
 hue_thresh_low <= hue_thresh_low - 1;
   else if (!button3 && !button_right)
 hue_thresh_low <= hue_thresh_low + 1;
   else if (!button2 && !button_left)
 hue_thresh_high <= hue_thresh_high - 1;
   else if (!button2 && !button_right)
 hue_thresh_high <= hue_thresh_high + 1;
   else if (!button1 && !button_left)
 sat_thresh <= sat_thresh - 1;
   else if (!button1 && !button_right)
 sat_thresh <= sat_thresh + 1;
   else if (!button0 && !button_left)
 val_thresh <= val_thresh - 1;
   else if (!button0 && !button_right)
 val_thresh <= val_thresh + 1;
end

always @(posedge clk) begin

   // need to fix 22 pixel offset
   if (switch[0] && (hcount == (com_x - 22) || vcount == (com_y )))
 pixel <= 18'b111111111111111111; // white
   else if (switch[0] &&
        ((hcount == (com_x_old0 - 22) && vcount == (com_y_old0 )) ||
     (hcount == (com_x_old1 - 22) && vcount == (com_y_old1 )) ||
     (hcount == (com_x_old2 - 22) && vcount == (com_y_old2 )) ||
     (hcount == (com_x_old3 - 22) && vcount == (com_y_old3 )) ||
     (hcount == (com_x_old4 - 22) && vcount == (com_y_old4 ))))
 pixel <= 18'b111111111111111111; // white
   else if (switch[0] && (hcount == (com_x_old0 - 22) || vcount == (com_y_old0 )))
 pixel <= 18'b011111011111011111; // gray
   //   else if (!button_left && (hue <= switch[7:0]) && inframe)
   //     pixel <= 18'b000000111111000000; // solid green
   //   else if (!button_right && (hue >= switch[7:0]) && inframe)
   //     pixel <= 18'b000000111111000000; // solid green
   else if (switch[1] && detected && inframe) //origsat 120
 pixel <= used ? 18'b111111000000000000 : 18'b000000111111000000;
   else if (inframe)
 pixel <= vr_pixel;
   else
 pixel <= 18'b000000000000000000; // solid black
   b <= blank;
   hs <= hsync;
   vs <= vsync;
end

// VGA Output.  In order to meet the setup and hold times of the
// AD7125, we send it ~clk.
// these three lines: modification for b&w -> color
//   assign vga_out_red = vga_red;
//   assign vga_out_green = vga_green;
//   assign vga_out_blue = vga_blue;
assign vga_out_sync_b = 1'b1;    // not used
assign vga_out_pixel_clock = ~clk;
assign vga_out_blank_b = ~b;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;

// debugging

//assign led = ~{vram_addr[18:13],reset,1};
//assign led = {com_x, com

always @(posedge clk)
  // dispdata <= {vram_read_data,9'b0,vram_addr};
  //dispdata <= {ntsc_data,9'b0,ntsc_addr};
  //dispdata <= {12'b0,switch[7:0],1'b0,com_x,2'b0,com_y};
  //dispdata <= {12'b0,switch[7:0],1'b0,com_x,2'b0,com_y};
  dispdata <= {12'b0,hue_thresh_low,hue_thresh_high,sat_thresh,val_thresh};

//////////////////////NATHAN
  ///////////////////////////////////////////////////////////////////////////
  wire [7:0] from_ac97_data, to_ac97_data;
wire     ready;

// allow user to adjust volume
wire     vup,vdown;
reg        old_vup,old_vdown;
debounce bup(.reset(reset),.clk(clock_27mhz),.noisy(~button_up),.clean(vup));
debounce bdown(.reset(reset),.clk(clock_27mhz),.noisy(~button_down),.clean(vdown));
reg [4:0]     volume;
always @ (posedge clock_27mhz) begin
   if (reset) volume <= 5'd22;
   else begin
  if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
  if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
   end
   old_vup <= vup;
   old_vdown <= vdown;
end
wire myreset;
wire myresetinv;
assign myreset = ~myresetinv;
// AC97 driver

wire [5:0] cut;
wire [2:0] lives;
wire [3:0] level;
wire [2:0] linemaker; //generates cuts from buttons
//wire [10:0] hcount;
//wire [9:0]  vcount;
//wire hsync,vsync,blank;
```

```
wire      gameon;
wire [8:0] score;
wire [9:0] sp0x; //sprite X positions
wire [9:0] sp1x;
wire [9:0] sp2x;
wire [9:0] sp3x;
wire [9:0] sp4x;
wire [9:0] sp5x;
wire [9:0] sp0y; //sprite Y positions
wire [9:0] sp1y;
wire [9:0] sp2y;
wire [9:0] sp3y;
wire [9:0] sp4y;
wire [9:0] sp5y;

wire [9:0] mysp0x; //used for start screen generation
wire [9:0] mysp1x;
wire [9:0] mysp0y;
wire [9:0] mysp1y;
wire [1:0] myspon;
wire [1:0] mysponsync;

wire [23:0] livespixel, levelpixel, scorepixel;

wire [5:0] spon; //sprites on
wire [9:0] rando; //Random number GENERATE THIS
wire [5:0] sponsync;

wire [1:0] bombcut;
wire [9:0] bomb0y;
wire [9:0] bomb1y;
wire [9:0] bomb0x;
wire [9:0] bomb1x;
wire [1:0] bombon;
wire [1:0] bombonsync;

// output useful things to the logic analyzer connectors
//   assign analyzer1_clock = ac97_bit_clock;
//   assign analyzer1_data[0] = audio_reset_b;
//   assign analyzer1_data[1] = ac97_sdata_out;
//   assign analyzer1_data[2] = ac97_sdata_in;
//   assign analyzer1_data[3] = ac97_synch;
//   assign analyzer1_data[15:4] = 0;
//assign led = {8'd0};
//   assign analyzer3_clock = ready;
//   assign analyzer3_data = {from_ac97_data, to_ac97_data};

/*  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));
 */
//    reg [23:0] rgb;
//   reg b,hs,vs;
reg [23:0] nathanpixelreg;
wire [23:0] nathanpixel1, nathanpixel2, splashpixel;
wire       phsync,pvsync,pblank;
always @(posedge clock_65mhz) begin
   nathanpixelreg <= nathanpixel1 | nathanpixel2;
   //     hs <= phsync;
   //     vs <= pvsync;
   //     b <= pblank;
end

//start screen logic
assign myspon[0] = (!(|level)) ? 1'b1 : spon[0]; //force proper signals for start screen
assign myspon[1] = (!(|level)) ? 1'b1 : spon[1];
assign mysp0x = (!(|level)) ? 10'd300 : sp0x;
assign mysp0y = (!(|level)) ? 10'd500 : sp0y;
assign mysp1x = (!(|level)) ? 10'd700 : sp1x;
```

```
  assign mysp1y = (!(|level)) ? 10'd500 : sp1y;
  assign mysponsync[0] = (!(|level)) ? 1'b1 : sponsync[0];
  assign mysponsync[1] = (!(|level)) ? 1'b1 : sponsync[1];


  //
  /*
   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;
   */
  //assign led = {~(score[7:0])};
  assign led = 8'b11111010;
  // ~rando[7:0];

  assign user1= {31'hZ, ready};
  //isaac_vga_red
  assign vga_out_red = (|(nathanpixelreg[23:16]) ? nathanpixelreg[23:16] : (isaac_vga_red &
splashpixel[23:16])) | (livespixel[23:16] | levelpixel[23:16] | scorepixel[23:16]) ;
  assign vga_out_green = (|(nathanpixelreg[15:8]) ? nathanpixelreg[15:8] : (isaac_vga_green &
splashpixel[15:8])) | (livespixel[15:8] | levelpixel[15:8] | scorepixel[15:8]) ;
  assign vga_out_blue = (|(nathanpixelreg[7:0]) ? nathanpixelreg[7:0] : (isaac_vga_blue &
splashpixel[7:0])) | (livespixel[7:0] | levelpixel[7:0] | scorepixel[7:0]) ;

  debounce b1(.reset(myreset),.clk(clock_27mhz),.noisy(button0),.clean(linemaker[0]));
  debounce b2(.reset(myreset),.clk(clock_27mhz),.noisy(button1),.clean(linemaker[1]));
  debounce b3(.reset(myreset),.clk(clock_27mhz),.noisy(button2),.clean(linemaker[2]));
  debounce bent(.reset(myreset),.clk(clock_27mhz),.noisy(button_enter),.clean(myresetinv));

  game_audio
sound(.clock(clock_27mhz),.reset(myreset), .ready(ready), .cut(cut), .bombcut(bombcut), .lives(li
ves), .level(level), .score(score), .to_ac97_data(to_ac97_data));
```

```
  lab5audio a(clock_27mhz, myreset, volume, from_ac97_data, to_ac97_data, ready,
         audio_reset_b, ac97_sdata_out, ac97_sdata_in,
         ac97_synch, ac97_bit_clock);

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

/*   game_video pg(.vclock(clock_65mhz),.reset(myreset),
        .hcount(hcount),.vcount(vcount),
               .hsync(hsync),.vsync(vsync),.blank(blank),
        .sp0x(mysp0x), .sp1x(mysp1x), .sp2x(sp2x), .sp3x(sp3x), .sp4x(sp4x), .sp5x(sp5x),

.sp0y(mysp0y), .sp1y(mysp1y), .sp2y(sp2y), .sp3y(sp3y), .sp4y(sp4y), .sp5y(sp5y), .b0x(bomb0x
), .b0y(bomb0y), .b1x(bomb1x), .b1y(bomb1y), .bombon(bombonsync), .spon({sponsync[5:2],
mysponsync}), .comx(com_x), .comy(com_y), .linemaker({~linemaker[2], ~linemaker[1],
~linemaker[0]}),
        .pixel(nathanpixel));
*/
wire [2:0] s0,s1,s2,s3,s4,s5,s6,s7;
wire cheat;

UI_wrapper dw(.clock(clock_27mhz), .reset(myreset),.level(level),.hcount(hcount),
.spon(spon),.rando(rando[3:0]
),.s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),.cheat(cheat), .cut(cut[1:0]));

sword swd(.pixel_clk(clock_65mhz),.myx(com_x - 22),.hcount(hcount),.y(com_y
),.vcount(vcount),.pixel(nathanpixel2));

lives
liv(.vclock(clock_65mhz),.x(11'd150),.hcount(hcount),.y(10'd675),.vcount(vcount),.lives(lives), .
level(level), .pixel(livespixel));

level
lev(.vclock(clock_65mhz),.x(11'd750),.hcount(hcount),.y(10'd675),.vcount(vcount),.level(level),.
pixel(levelpixel));
```

```
score
sco(.vclock(clock_65mhz), .slowclock(clock_27mhz),.x(11'd400),.hcount(hcount),.y(10'd675),.v
count(vcount),.score(score),.pixel(scorepixel),.reset(myreset),.level(level));

splash_screen
spl(.pixel_clk(clock_65mhz), .level(level),.x(11'd262),.hcount(hcount),.y(10'd200),.vcount(vcoun
t),.mypixel(splashpixel));

picture_blob pb(.pixel_clk(clock_65mhz),.cheat(cheat), .level(level),
.x0(mysp0x-32), .x1(mysp1x-32), .x2(sp2x-32), .x3(sp3x-32), .x4(sp4x-32), .x5(sp5x-
32), .bomb0x(bomb0x-32), .bomb1x(bomb1x-32), .hcount(hcount),
.y0(mysp0y-32), .y1(mysp1y-32), .y2(sp2y-32), .y3(sp3y-32), .y4(sp4y-32), .y5(sp5y-
32), .bomb0y(bomb0y-32), .bomb1y(bomb1y-32),
.vcount(vcount), .spon({sponsync[5:2], mysponsync}
), .bombon(bombonsync), .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
.pixel(nathanpixel1), .hsync(hsync), .vsync(vsync), .phsync(phsync), .pvsync(pvsync));



  geto_cut_detector gcd(.clock(clock_27mhz), .reset(myreset),
        .sp0y(mysp0y), .sp1y(mysp1y), .sp2y(sp2y), .sp3y(sp3y), .sp4y(sp4y), .sp5y(sp5y),
        .sp0x(mysp0x), .sp1x(mysp1x), .sp2x(sp2x), .sp3x(sp3x), .sp4x(sp4x), .sp5x(sp5x),
        .b0y(bomb0y), .b1y(bomb1y),
        .b0x(bomb0x), .b1x(bomb1x),
        .spon({spon[5:2], myspon}), .bombon(bombon),
        .cut(cut), .bombcut(bombcut),
        .com_x_old0(com_x_old0 - 22), .com_y_old0(com_y_old0 ),
        .com_x_old1(com_x_old1 - 22), .com_y_old1(com_y_old1 ),
        .com_x_old2(com_x_old2 - 22), .com_y_old2(com_y_old2 ),
        .com_x_old3(com_x_old3 - 22), .com_y_old3(com_y_old3 ),
        .com_x_old4(com_x_old4 - 22), .com_y_old4(com_y_old4 ));


  game_logic
gl(.clock(clock_27mhz), .reset(myreset), .cut(cut), .bombcut(bombcut), .sp0y(sp0y), .sp1y(sp1y),
 .sp2y(sp2y), .sp3y(sp3y), .sp4y(sp4y), .sp5y(sp5y), .b0y(bomb0y), .b1y(bomb1y), .spon(spon), .
bombon(bombon), .level(level), .score(score), .lives(lives), .gameon(gameon));
```

```
   sprite_logic sl0(.clock(clock_27mhz), .reset(myreset), .vsync(vsync), .on(spon[0]
), .rando(rando), .ypos(sp0y), .xpos(sp0x), .syncstate(sponsync[0]));
   sprite_logic sl1(.clock(clock_27mhz), .reset(myreset), .vsync(vsync), .on(spon[1]
), .rando(rando), .ypos(sp1y), .xpos(sp1x), .syncstate(sponsync[1]));
   sprite_logic sl2(.clock(clock_27mhz), .reset(myreset), .vsync(vsync), .on(spon[2]
), .rando(rando), .ypos(sp2y), .xpos(sp2x), .syncstate(sponsync[2]));
   sprite_logic sl3(.clock(clock_27mhz), .reset(myreset), .vsync(vsync), .on(spon[3]
), .rando(rando), .ypos(sp3y), .xpos(sp3x), .syncstate(sponsync[3]));
   sprite_logic sl4(.clock(clock_27mhz), .reset(myreset), .vsync(vsync), .on(spon[4]
), .rando(rando), .ypos(sp4y), .xpos(sp4x), .syncstate(sponsync[4]));
   sprite_logic sl5(.clock(clock_27mhz), .reset(myreset), .vsync(vsync), .on(spon[5]
), .rando(rando), .ypos(sp5y), .xpos(sp5x), .syncstate(sponsync[5]));
   sprite_logic bomb0(.clock(clock_27mhz), .reset(myreset), .vsync(vsync), .on(bombon[0]
), .rando(rando), .ypos(bomb0y), .xpos(bomb0x), .syncstate(bombonsync[0]));
   sprite_logic bomb1(.clock(clock_27mhz), .reset(myreset), .vsync(vsync), .on(bombon[1]
), .rando(rando), .ypos(bomb1y), .xpos(bomb1x), .syncstate(bombonsync[1]));



   geto_randomizer random(.xpos(com_x), .ypos(com_y), .rando(rando));


endmodule
```

Python Serializer (Drew)

```
from PIL import Image
import struct, serial


def img2list(filename):
	# extracts the raw values (0-255) from an image as a list of 3-tuples
	im = Image.open(filename)
	return list(im.getdata())

def unpack(l):
	# flattens a list
	return [item for sublist in l for item in sublist]


def img2flat(filename):
	return repr(''.join([struct.pack("B", i) for i in unpack(img2list(filename))]))

def send(s, always=False):
	ser = serial.Serial('/dev/tty.usbserial-DPE0AJZN') # open the serial port
	ser.write(s)
	while always:
	    ser.write(s)
	ser.close()

s = img2flat("test_cf.bmp")
print "Raw hex stream:"
#s
= "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x
61" * 200
#s = "\x62" * 6400
```

```
print s
print "length: ", len(s)
print "Sending via serial..."
send(s)
print "Data sent"
```

Python: BMP2COE (Drew)

```
from PIL import Image
import struct, serial


def img2list(filename):
	# extracts the raw values (0-255) from an image as a list of 3-tuples
	im = Image.open(filename)
	return list(im.getdata())

def unpack(l):
	# flattens a list
	return [item for sublist in l for item in sublist]

def _bin(x, width):
    return ''.join(str((x>>i)&1) for i in xrange(width-1,-1,-1))

def img2flat(filename):
    f = open(filename[:-4] + '.coe', "w")
    f.write("memory_initialization_radix=2;\nmemory_initialization_vector=\n")
    for i in img2list(filename):
        print i[0], i[1], i[2]
        f.write("%s%s%s,\n" %  (_bin(i[0],8), _bin(i[1],8), _bin(i[2],8)))
    f.write(";");
    f.close()

img2flat('sword.bmp')
```