6.111 Final Project: The Dueling Club

Marayna Martinez
Lotta Blumberg

Abstract:

The Harry Potter universe has captivated readers and moviegoers for two decades. A huge component of that fascination is thanks to JK Rowling's phenomenal descriptions of magic. For our final project, we decided to that magic to the real world in the form of a spell casting game.

Basically, you should be able to make gestures with a wand in front of a camera and certain things will happen on a computer screen depending on your gestures. Our "wand" will be and IR LED and wand positions over time will dictate what spells get cast. The virtual drum kit project served as inspiration for our game. We also chose this idea because it is very buildable. At minimum, we can have a single player make gestures to do things like levitate objects on the screen. We can make the project more complex but adding in a second player and making the spell interactions more complicated.

Overview:

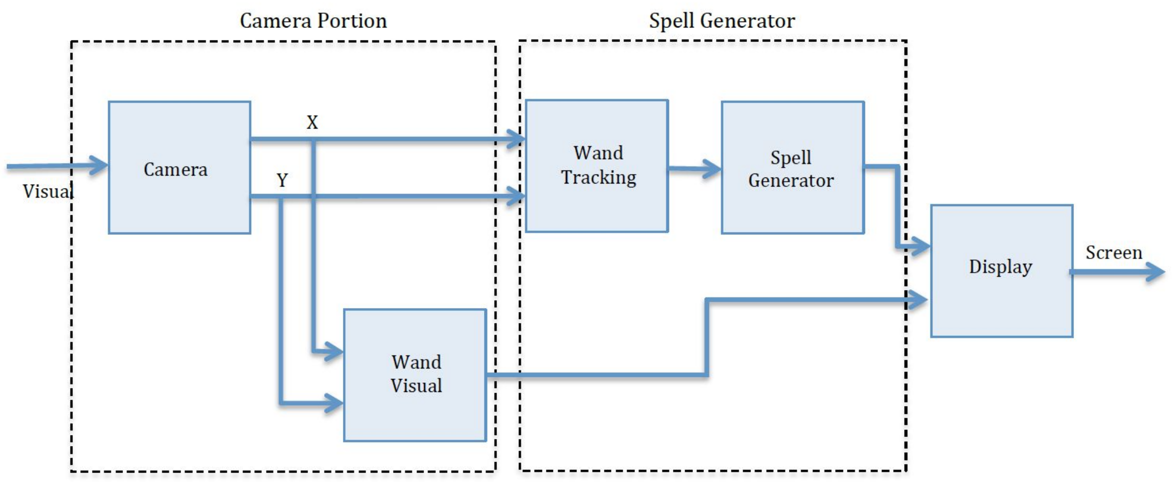
Our final project features two modes: a single player mode and a two player battle mode. In single player mode, a player can cast up to six spells that manipulate sprites on screen as well as lights on the labkit. In two player mode, each player can cast up to eight destructive spells to inflict damage on their opponent. The hex display on the labkit will indicate the winner of each round.

In terms of implementation, a camera with a floppy disk over it views an IR LED representing the wand tip. Modules provide the wand position and track it over time to distinguish between different spells in both modes. Depending on the spell cast, different modules are triggered dictating the computer response. A user interface displays the placement of the wand, the spell responses, and the two player game set up.

Overall, the player can wave an IR LED in front of a camera in two different modes and different things will happen as a response.

Overview of Implementation:

Our project as basically divided into two parts: the camera and the spell generator. The camera portion of the project required taking in data from the camera and processing it to achieve an (x,y) coordinate pair. Then, the wand position would be displayed on screen. The spell generator portion of the project involved tracking the movements of the wand over time and triggering different spells when the wand travels of a specific path.

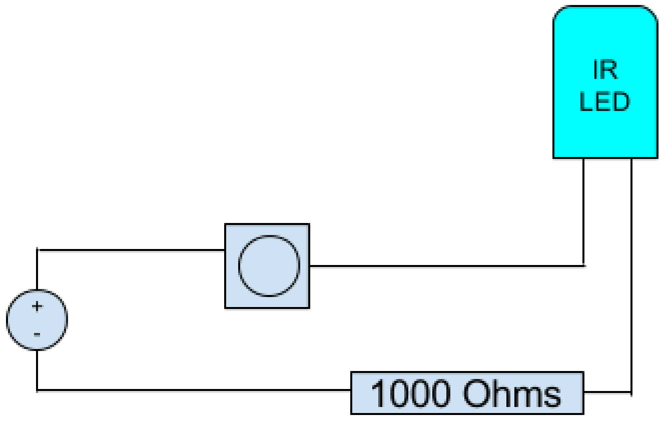


Marayna was in charge of the spell generator portion of the project and Lotta focused on the camera portion. The single player spells were split between Marayna and Lotta.

Module Name	Written By
aparecium	Marayna
accio	Marayna
blockTracker	Marayna
division	Marayna
twoPlayerSetup	Marayna
healthBars	Marayna
hitmaker	Marayna
healthbarChanger	Marayna
blockTrackerTP	Marayna
divisionTPRight/divisionTPLeft	Marayna
Star	Lotta
Wand	Lotta
Winner	Lotta
Average (left and right)	Lotta
Lumos (nox)	Lotta
Wingardium Levisoa	Lotta
Engorgio	Lotta

Tracking the Wand with the Camera: (Lotta)

Hardware:
The wand were built using a simple circuit with a button that turns on an IR LED. Here is a diagram:



The infrared light then passed through a floppy disk, which filters out visible light but lets through the infrared light. This floppy disk was mounted onto a camera.

For the camera, we used an NTSC camera with a range that goes well into the infrared spectrum.

Finding the Wand:

Once the we had an input from the camera, I used the sample NTSC verilog online to figure out where the wand is. Running that code gives a mostly dark image with a bright spot wherever the wand is. To clean this up, I made a threshold at around 50% brightness so that every pixel would either be black or white and the white pixels would all be around where the wand is.

Average Module:

This is the module that actually takes in an image and outputs an x and y coordinate for the wand location.

Inputs:

- clock: standard clock
- hcount: says the horizontal coordinate of the current pixel
- vcount: says the vertical coordinate of the current pixel
- is_bright: one bit that says whether the pixel at the current hcount and vcount satisfies the threshold from earlier

Outputs:

- avg_x: the x coordinate of the wand
- avg_y: the y coordinate of the wand

How it works:
This module tries to find the left-most, right-most, top-most, and bottom-most bright spots on the screen. The way it does this is by going through the hcounts and vcounts that are within the camera image and then storing in memory the location of the pixel most in each direction so far. It compares each of these to the current pixel and stores the more extreme coordinate. Once every pixel has been looked at (hcount and vcount are back at 0), the program outputs avg_x as the average of the left-most and right-most pixels and avg_y as the average of the top-most and bottom-most pixels. It then also resets to look at the next cycle. The program also keeps track of whether there is a light on at all, and if there are no bright pixels, it will output an avg_x and avg_y off screen. Note that in order to have the wand location be a mirror of what the player does, the avg_x output had to be 1024 - the average value.

Design Choice:
In designing the module, there was a choice to be made as to whether to just take the extreme coordinates of the light or to try to take the center of mass. In the end, I decided that taking the average of the extremes, though more prone to outliers, would be faster and more efficient. I think this was a reasonable design choice because it worked well. There never was a problem where outliers messed with the wand location.

Two-Player version:
For the two player version, I made another two versions of this module. The main difference was that in these, the range of hcount in which you look at the pixel was half the size so that you only look at the half of the screen that is relevant.

Displaying the Wand Sprite: (Lotta)

The visual for the wand sprite was a magic wand that had a handle and a star on top. This was made with two modules, a star module and a wand module.

Star:

This module draws a star on the display

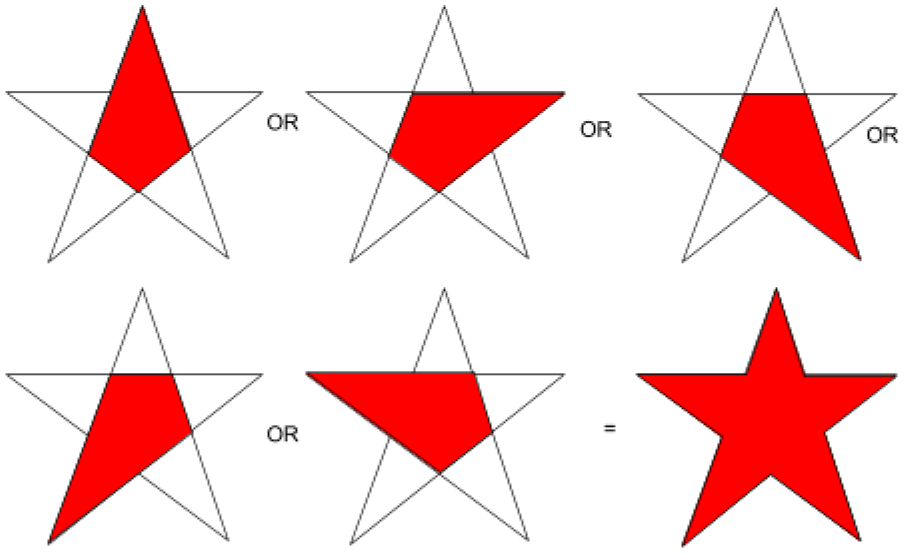
Inputs:

- clock: standard clock
- x: the center of the star along the x axis
- hcount: the horizontal coordinate of the current pixel
- y: the center of the star along the y axis
- vcount: the vertical coordinate of the current pixel
- color: the color of the star
- size: roughly the diameter of the star

Output:

- pixel: the color of the current pixel (either black if it is not part of the star or color if it is part of the star)

How it works:
This module creates 5 equations that form the outline of a star. It then looks for 5 regions, as shown, and colors in any pixel that falls into either of those 5 regions.



Design Choice:
This sprite could have been created by simply assigning pixels to make a star. This would have been faster and less computationally intensive. However, it would have also been a bit tedious to do and far less fun. I basically ended up designing the star in this way because thinking about the math and equations that I would use was more entertaining and interesting for me.

Challenges:
Because I went with the more complicated but fun approach, I ran into some timing issues once the project was all put together. This is a bug that didn't exist with just the one-player mode, but adding the two-player was too much. The bug had the star flicker in and out and created vertical lines outside the star that should not have existed. This was eventually fixed by having the equations be generated within a clocked always block instead of being assigned outside of it.

Wand:

This module draws a wand

Inputs:

- clock: standard clock
- x: the center of the wand along the x axis
- hcount: the horizontal coordinate of the current pixel
- y: the center of the wand along the y axis
- vcount: the vertical coordinate of the current pixel
- color: the color of the star
- size: roughly the diameter of the star on top of the wand

Outputs:

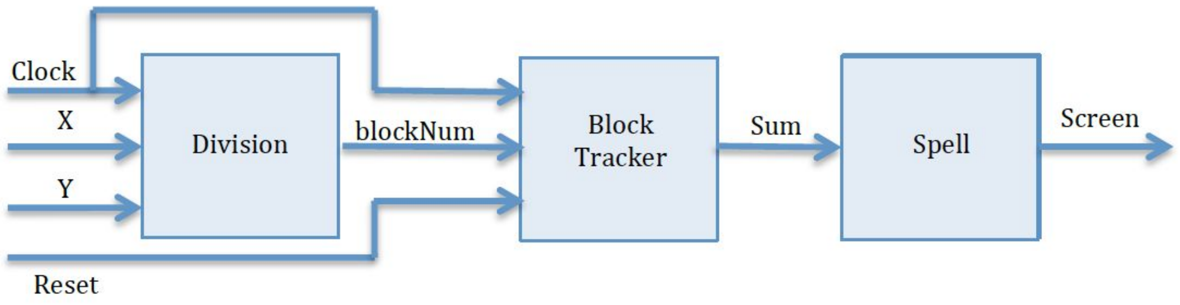
- pixel: the color of the current pixel (either black if it is not part of the wand or color if it is part of the wand)

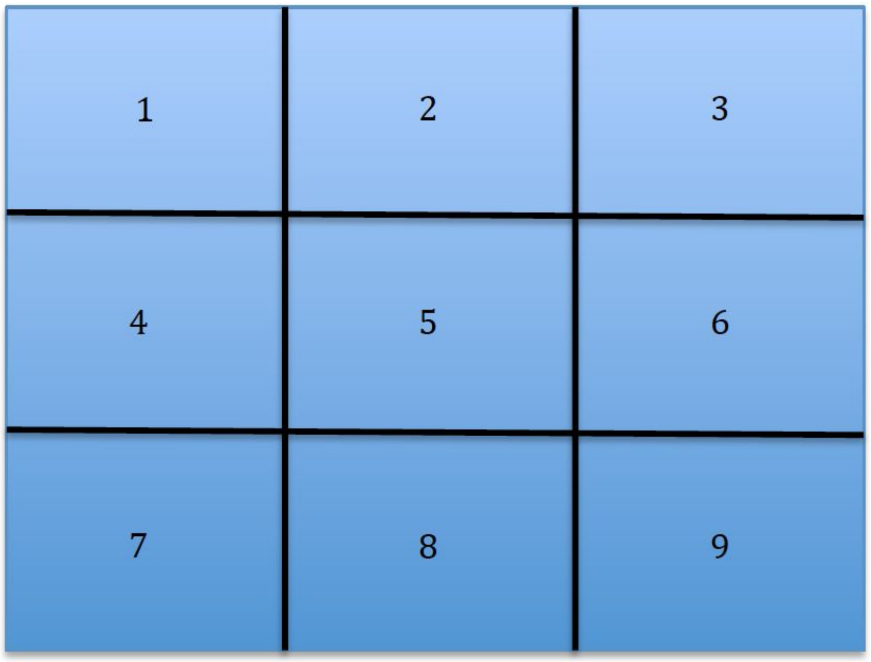
How it works:
This module is fairly simple. It uses the star module to draw a star. It then calls the bloc module (provided in Lab 3) to draw a handle. Finally, it ORs the pixels from each output to create the new pixel output. The main design choice here involves playing around with sizes and shapes until it looks pretty.

Single Player Spell Generation: (Marayna)

Overview:
Conceptually, in order to cast a spell, you need the tip of the wand to trace a certain pattern. In order to do this, we need to be able to characterize the location of the wand tip, then track the location overtime. We accomplished this by dividing the screen into different areas and numbering each one to characterize location. Then, we kept a memory of the areas the wand had been in.

Block Diagram for Spell Generator:





Single Player Layout

Division:

This module divides our computer screen into nine distinct blocks that will be used to track position. Each block is them numbered in order to distinguish between them.

Inputs:

- X: x position of the wand
- Y: y position of the wand
- Clock: standard clock from the labkit

Outputs:

- blockNum: the number of the block that the wand is in

How it works:
At the positive edge of every clock cycle, the module takes in the x and y coordinate of the wand. Using a series of if-statements, the module outputs the number of the block that the wand is in. For example, if the x position is between 0 and 340 and the y positions is between 0 and 256, then the module will output the number “1”. I did a proof of concept with a 2x2 block screen. I used the hex display module to display the block number that was being output. After I was satisfied that the 4 block version worked, I edited the module to the 9 block version that we used in the final product. I tested the module using the game controller concept from the lab 3 pong game.

Block Tracker:

This module tracks what blocks the wand has been in as it moves around the screen. Using a 3 slot array, it keeps track of the last three blocks the wand had been in. Not only did it keep track of position, but order too.

Inputs:

- Clock: standard clock from the labkit, the same clock fed into division so everything stays on the same clock cycle
- Reset: the enter button on the labkit
- blockNum: the 4 bit decimal number ranging from 1 to 9 output by division

Outputs:

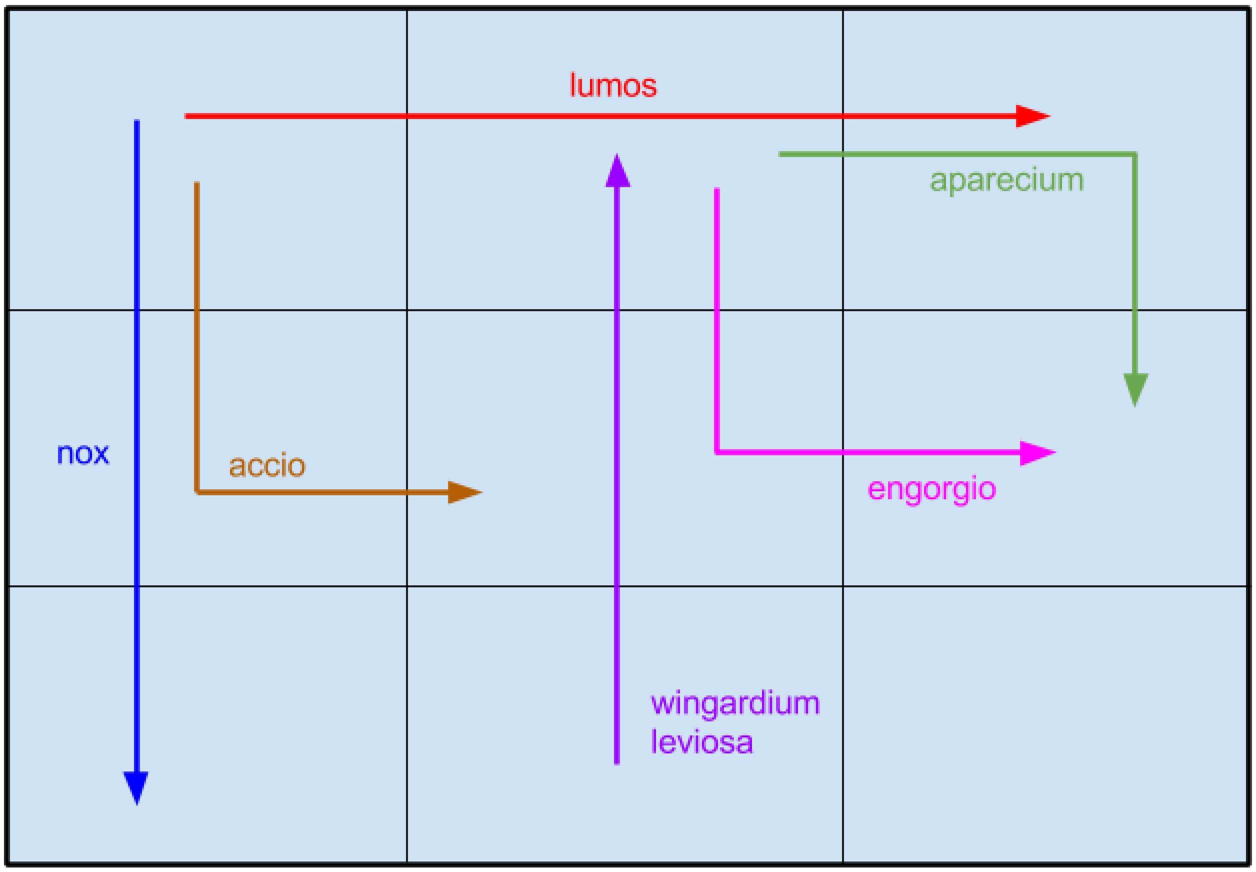
- Sum: a modified sum of the 3 last visited blocks, this acts as the spell number, each spell has a unique sum output

How it works:
This module creates a 3 slot array of 4 bits each. At each clock cycle, the module checks if block number is the same as the one logged at index 2. If the block number has changed, then index 2 is set to equal the current block number, index 1 is set to equal index 2, and index 0 is set to each index 1. Essentially, anytime the wand moves to a new block, the memory left shifts everything to only track the current location and the previous two locations. Then index 0 is multiplied by 100, index 1 is multiplied by 10, and index 2 is multiplied by 1. These adjusted values are summed to equal the output sum. Now, every sequence of 3 blocks has a unique sum. For example, if you traced along the top row of blocks, the sum output would be 123. When I begin designing the single player spell generator, I worked with a 2x2 divided screen. In that case, all the possible sums of 3 blocks was unique. When I expanded the single player spell generator to a 3x3 screen, the sums were no longer unique, I toyed with the idea of just constructing a memory block, then checking all three slots every clock cycle to see if the right pattern had been traced. But, my method of summing the modified indices was more efficient and easier to check for spells.

Single Player Spells: (Marayna and Lotta)

Overview:

In the Single-Player version of this game, you can move a wand around to interact with various objects on screen and even turn a light on and off on the labkit. Each spell is triggered by running the wand through a certain set of blocks on the screen. The patterns are drawn below:



Accio: (Marayna)

This module casts the accio spell, which summons a block sprite to the position of the wand.

Inputs:

- ➔ hcount
- ➔ vcount

- ➔ Sum: the output sum from blockTracker, essentially the unique spell number
- ➔ wandX: the x coordinate of the wand tip
- ➔ wandY the y coordinate of the wand tip

Outputs:

- ➔ newX: the new x coordinate to be delivered to an arbitrary sprite
- ➔ newY: the new y coordinate to be delivered to an arbitrary sprite

How it works:
Conceptually, accio is a summoning spell which should move a sprite to the location of the wand, then let it fall back. I started by updating the position of the sprite in a very binary matter. While the input sum is equal to 145, set newX and new Y equal to wandX and wandY. If the sum is anything else, newX defaults to 750 and newY defaults to 703. The next step was to add in a way for the sprite to move smoothly towards the wand position. I decided that I wanted the movement to happen over the course of 64 frames. So, in order to determine the proper speed, I took the difference in x and y coordinates and divided them by 64. Next, using a counter, I slowly updated the newX and newY using my calculated speeds over the course of 64 frames. This system was not perfect for several reasons. First, replicating the process backwards was extremely difficult and glitchy mostly because the wandX and wandY coordinates can change drastically while still executing the same spell. The speed would get very messed up and the sprite would not end up back at (750, 703) as intended. Second, the speed updates every clock cycle, so as the position of the wand changes, the speed also changes. Because the position of the sprite at rest is always constant, this poses less of an issue than the first question. At most, the sprite slows down as it approaches the target, but the change isn't detrimental to the movement of the sprite.

Aparecium: (Marayna)

This module casts the aparecium spell, which reveals a secret message ("Hi!") on screen.

Input:

- ➔ hcount
- ➔ vcount
- ➔ Sum: the output sum from blockTracker, essentially the unique spell number
- ➔ Color: the target color of the message

Output:

- ➔ Pixel: the data for the pixel coloration of the "secret message"

How it works:
I started with a binary on off switch for the message. When sum equals 236, the pixels that compose the word “Hi!” are set to whatever the input color is. When the sum is anything else, the pixels are set to black. Next, I worked on having the message fade in. I used the alpha blending technique from lab 3. I right shifted the RBG components less and less until they reached the full input color. Using a counter, I lessened the right shift by one integer value every 25000000 clock cycles.

Lumos and Nox: (Lotta)

This module covers two spells. Lumos turns on an LED light on the labkit. Nox turns the light off.

Inputs:
- sum: the last three blocks visited
- clock: standard clock

Output:
- led_on: stores the state of the led of the labkit

How it works:
This is very straightforward module. Basically, the last three blocks corresponds to lumos, the led is turned on. If the last three blocks corresponds to nox, the led is turned off.

Wingardium Leviosa: (Lotta)

This module takes a block on screen, lifts it up, and then lowers it down.

Inputs:
- sum: the last three blocks visited
- clock: standard clock

Output:
- y: stores the y-coordinate of the block on the screen

How it works:
Whenever the spell is triggered by sum, this module increases the y and then decreases it in a little animation. While the animation is playing, the spell cannot be triggered again.

This module uses a timer and moves the y coordinate up a pixel at each timer reset and then back down once the y has reached its top value.

Engorgio: (Lotta)

This module takes a block on screen, makes it larger, and then shrinks it back down.

Inputs:

- sum: the last three blocks visited
- clock: standard clock

Output:

- x: stores the x and y coordinates of the block on the screen
- size: stores the size of the block

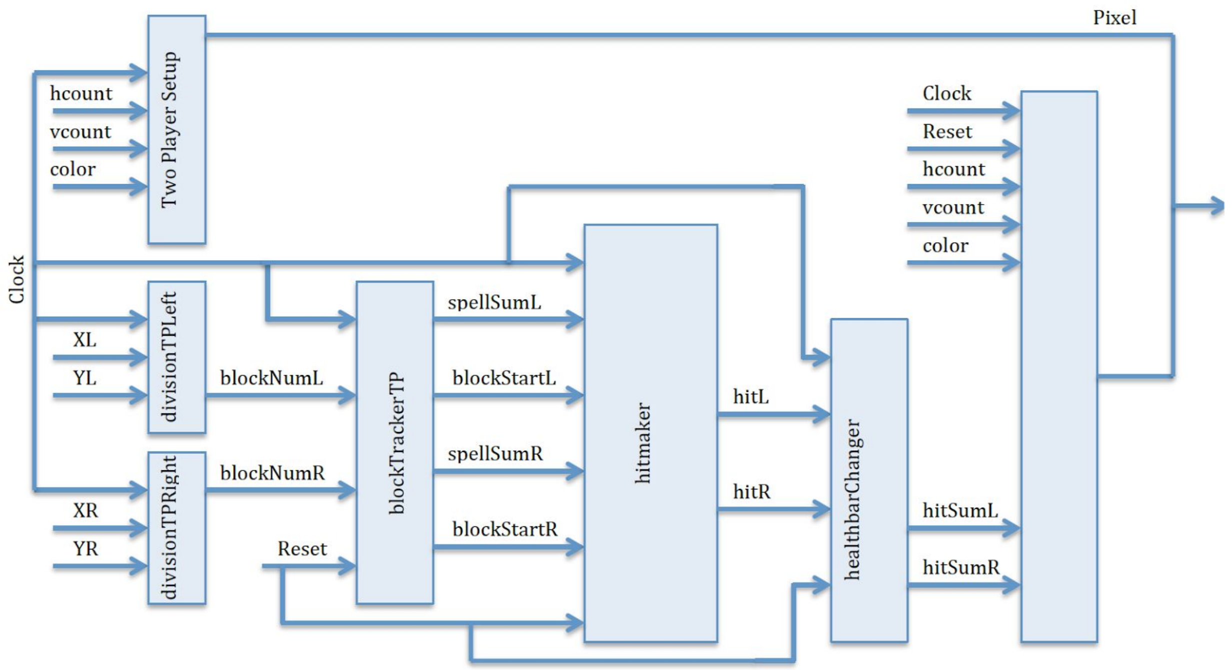
How it works:
Whenever the spell is triggered by sum, this module decrease the x and increases the size. This is done in a 1:2 ratio so that the block's center of mass stays constant as it grows and then shrinks. It then undoes this, increasing x and decreasing size, again in a 1:2 ratio. This all happens in a little animation. While the animation is playing, the spell cannot be triggered again. This module uses a timer for the animation.

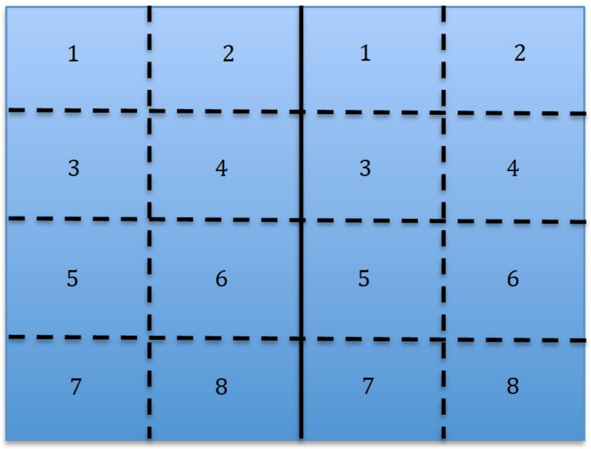
Two-Player Mode: (Mostly Marayna)

Overview:

Two player mode operates under the same principles that govern the single player mode. But, in this mode, each side of the screen is separated into two smaller versions of the single player mode (8 blocks instead of 9). Each spell is still composed of a three block sequence. Another memory block tracks all of the spells cast and corresponding hits to the opponent's health bar. This memory is used to calculate the state of the health bar.

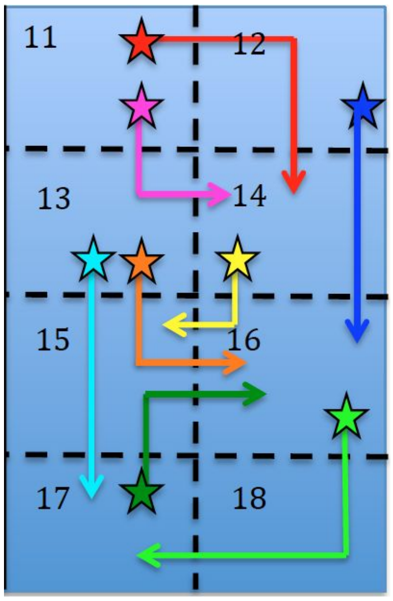
Block Diagram of Two Player Mode:





Screen Layout

Two Player Spells Diagram



Starting Block	Spell Sum	Hit
11	37	20
13	44	30
14	45	40
16	51	50
17	48	45
13	45	35
12	44	12
11	38	23

Chart of starting block, spell sum, and inflicted hit

twoPlayerSetup:

This module sets up the screen for two player mode. It divides the screen in half and creates the outline of the health bars at the top of each side of the screen.

Inputs:

- ➔ Hcount
- ➔ Vcount
- ➔ Color: the color of the set up for the two player version

Outputs:

- ➔ Pixel: the data for the pixel coloration of the setup of the two player mode

How it works:
It basically adds a white line down the center of the screen and the adds white outlined rectangles to the top of each side as an outline for the health bars.

divisionTPRight and divisionTPLeft:

These modules works the same way as division in the single player mode. These are two modules that separate each side of the screen into 8 labelled blocks.

Inputs:

- ➔ X: x position of the wand
- ➔ Y: y position of the wand
- ➔ Clock: standard clock from the labkit

Outputs:

- ➔ blockNum: the number of the block that the wand is in

How it works:
This module works in the exact same way divisionTP. It is just separated into two modules, one for each side of the screen.

blockTrackerTP:

This module works the same way as the block tracker in the single player mode. Two instances of this single module are used to track spells on the left and right sides of the screen.

Inputs:

- ➔ Clock: standard clock from the labkit
- ➔ Reset: the enter button on the labkit
- ➔ blockNum: the number of the block that the wand is in

Outputs:

- ➔ Sum: a sum of the 3 last visited blocks, this acts as the spell number, unlike the single player version each sum is not unique
- ➔ blockStart: this is the block number of the first block in the three block sequence

How it works:
This module works the same way as blockTracker but with two main differences. First, the sum is not modified. Instead of multiplying certain indices, I just add indices 0 through 2. Second, I output index 0 as the blockStart. I made these design choices because my one player method was more computation heavy. I did not want to use the computation heavy method when I had two sets of data to keep track of.

Hitmaker:

This module translates each spell into a “hit”. A hit is basically the number of pixels the health bar will be decreased by. It takes in the spell number output by blockTrackerTP and it feeds it into healthbarChanger.

Inputs:

- ➔ Clock: standard clock from the labkit
- ➔ Reset: the enter button on the labkit
- ➔ Sum: a sum of the 3 last visited blocks, this acts as the spell number, unlike the single player version each sum is not unique
- ➔ blockStart: this is the block number of the first block in the three block sequence

Outputs:

- ➔ Hit: the number of pixels the opponent's health bar will be depleted by when a spell is cast

How it works:
In two player mode, each spell has two identifying factors: the sum of each of the blocks and the block that each spell originated in. Based on the spell characteristics, the hit value is assigned through a series of if/else if statements.

healthbarChanger:

This module sums up the individual hits for an overall sum of the damage inflicted by a player. This sum can then be passed into the health bar module in order to physically display the damage that has been inflicted.

Inputs:

- ➔ Clock: standard clock from the labkit
- ➔ Reset: the enter button on the labkit
- ➔ Hit: the number of pixels the opponent's health bar will be depleted by when a spell is cast

Outputs:

- ➔ Sum: the sum of all the hits you inflict on your opponent

How it works:

The less effective hit is inflicts 12 pixels worth of damage. At worst, only inflicting 12 pixels worth of damage, it would take 40 moves to defeat your opponent. So, I created a 40 slot memory to track all of the hits. Then, every clock cycle, I sum all of the hits to find the total damage that should be reflected in the health bar. Two instances of this module are used for the left and right sides of the screen. The hit outputs are placed into the left and right inputs of the health bar module to accurately reflect damage.

Health Bars: (Marayna and Lotta)

This module displays the healthbars.

Inputs:

- clock: standard clock
- reset: button reset
- hcount: horizontal coordinate of current pixel
- vcount: vertical coordinate of current pixel
- left: hit points taken by left player
- right: hit points taken by right player

Outputs:

- pixel: color of the current pixel

How it works:
This module colors the screen wherever the health bars exist. The left, top, and bottom boundaries of the health bar are constant. The right boundary moves down as left and right increase. When either health bar runs out, the health bars stop displaying until the game is reset. Upon reset, the healthbars appear full again. The colors of the health bars change as a function of how large it is. So, when the health bar is full, the bar is green. As it moves down, it becomes less and less green and more and more red.

Winner: (Lotta)

This module is used to track who won the duel so that it can be displayed on the labkit.

Inputs:

- clock: standard clock
- reset: allows you to restart the game
- left: the number of hit points taken on the left

- right: the number of hit points taken on the right

Outputs:

- led_left: is 1 if the left player wins. Otherwise 0.
- led_right: is 1 if the right player wins. Otherwise 0.

How it works:
This module keeps track of whether the game is over yet or not. If the game is over, it does nothing until the game is reset. If the game is still going, then it checks to see of left or right are over 475 which is the total number of hit points each player has. If so, it assigns the other player's led to winning. Once a player wins, the game is over.

The Process:

How Time was Spent:

Week	Marayna	Lotta
Oct 24 - Oct 30	Project planning	Project Planning
Oct 31 - Nov 6	Working on single player version	Learning the camera code
Nov 7 - Nov 13	Debugging single player version, picking spells, planning two player version	Wand Tracking and building wands
Nov 14 - Nov 20	Expanding single player version, starting single player spells	Creating single player spells
Nov 21 - Nov 27	Thanksgiving	Thanksgiving
Nov 28 - Dec 4	Finishing single player spells, two player version	Spells and integration
Dec 5 - Dec 11	Debugging two player health bars, debugging	Two-Player Mode integration

What to do better:

In general, given a bit more time, I would have liked to add the Harry Potter theme song to the background of our game, improved our graphics, and added more spells. We could also make the graphics on the spells prettier. For the two-player version, we could add more types of things to do such as healing your own health bar or putting up a shield that protects you from future spells for a few seconds.

Appendix Verilog Modules:

Here are the modules that we wrote.

```
////////////////////////////////////////////////////////////////////
//
// blob: generate rectangle on screen
//
////////////////////////////////////////////////////////////////////
module blob
   (input [10:0] x,hcount,
        input [9:0] y,vcount,
         input [23:0] color,
         input [10:0] width, height,
        output reg [23:0] pixel);

   always @ * begin
        if ((hcount >= x && hcount < (x+width)) &&
               (vcount >= y && vcount < (y+height)))
                      pixel = color;
        else pixel = 0;
   end
endmodule


//////////////////////////////////////////////////////////////////////
//
// star: generate a star on screen!
// This is done by splitting the star into 5 sections and adding them together
//
///////////////////////////////////////////////////////////////////////

module star
        (input [10:0] x,hcount,
         input clock,
        input [9:0] y,vcount,
         input [23:0] color,
         input signed [10:0] size,
        output reg [23:0] pixel);

         reg signed [15:0] x_coord;
         reg signed [15:0] y_coord;

         reg eq1;
         reg eq2;
         reg eq3;
         reg eq4;
         reg eq5;


         always @(posedge clock) begin
                x_coord <= hcount - x; // normalize variables
                y_coord <= vcount - y;

                eq1 <= (y_coord+2*size> 3*x_coord); // equations for the star
                eq2 <= (y_coord+3*x_coord+2*size > 0);
                eq3 <= (y_coord+x_coord < size);
                eq4 <= (y_coord < x_coord+size);
                eq5 <= (y_coord+size/2 > 0);

                if (eq2 && eq3 && eq4 && eq5) pixel <= color; // Test that 4 out of 5 equations
are satisfied
                else if (eq1 && eq3 && eq4 && eq5) pixel <= color;
                else if (eq1 && eq2 && eq4 && eq5) pixel <= color;
                else if (eq1 && eq2 && eq3 && eq5) pixel <= color;
                else if (eq1 && eq2 && eq3 && eq4) pixel <= color;
                else pixel <= 0;
         end
endmodule


////////////////////////////////////////////////////////////////////////
//
// wand_sprite
//
////////////////////////////////////////////////////////////////////////

module wand_sprite
        (input [10:0] x,hcount,
        input [9:0] y,vcount,
         input clock,
         input [23:0] color,
         input signed [10:0] size,
        output [23:0] pixel);

         wire [23:0] star_pixel;
         wire [23:0] handle_pixel;

         star tip(.x(x),.y(y),.clock(clock),.hcount(hcount),.vcount(vcount),
                              .pixel(star_pixel),.color(color),.size(10));

         parameter WHITE = 24'hFFFFFF;

         blob handle(.x(x),.y(y),.hcount(hcount),.vcount(vcount),
                              .pixel(handle_pixel),.color(color),.width(6),.height(60));

         assign pixel = star_pixel|handle_pixel;

endmodule


////////////////////////////////////////////////////////////////////////
//
// average value
//
////////////////////////////////////////////////////////////////////////


module average(clock, hcount, vcount, is_bright, avg_x, avg_y);
        input clock;
        input [10:0] hcount;
        input [9:0] vcount;
```

```
        input is_bright;
        output reg [10:0] avg_x;
        output reg [9:0] avg_y;

        reg [10:0] min_hcount = 1024;
        reg [10:0] max_hcount = 0;
        reg [9:0] min_vcount = 768;
        reg [9:0] max_vcount = 0;

        reg exists_light = 0;

        parameter SCREENH = 1024;
        parameter SCREENV = 768;

        parameter MINH = 36;
        parameter MAXH = MINH+SCREENH/2;//750;
        parameter MINV = 74;
        parameter MAXV = MINV+SCREENV/2;//566;




        always @(posedge clock) begin
                if (hcount ==0 && vcount == 0) begin
                        min_hcount <= SCREENH; // reset values
                        max_hcount <= 0;
                        min_vcount <= SCREENV;
                        max_vcount <= 0;
                        exists_light <= 0;
                        if (exists_light) begin // give an average only if there is a light
                                avg_x <= SCREENH-((min_hcount+max_hcount)/2-MINH)*2; // find
average
                                avg_y <= ((min_vcount+max_vcount)/2-MINV)*2;
                        end
                        else begin // if there is no light, put the point somewhere off screen
                                avg_x <= 1200;
                                avg_y <= 800;
                        end
                end
                if (is_bright && hcount > MINH && hcount < MAXH && vcount < MAXV && vcount >
MINV) begin
                        if (hcount < min_hcount) min_hcount <= hcount;
                        if (hcount > max_hcount) max_hcount <= hcount;
                        if (vcount < min_vcount) min_vcount <= vcount;
                        if (vcount > max_vcount) max_vcount <= vcount;
                        exists_light <= 1;
                end
        end

endmodule


////////////////////////////////////////////////////////////////////////
//
// average value for left player
//
////////////////////////////////////////////////////////////////////////
```

```
module average_r(clock, hcount, vcount, is_bright, avg_x, avg_y);
	input clock;
	input [10:0] hcount;
	input [9:0] vcount;
	input is_bright;
	output reg [10:0] avg_x;
	output reg [9:0] avg_y;

	reg [10:0] min_hcount = 1024;
	reg [10:0] max_hcount = 0;
	reg [9:0] min_vcount = 768;
	reg [9:0] max_vcount = 0;

	reg exists_light = 0;

	parameter SCREENH = 1024;
	parameter SCREENV = 768;

	parameter MINH = 36;
	parameter MAXH = MINH+(SCREENH/4);
	parameter MINV = 74;
	parameter MAXV = MINV+SCREENV/2;




	always @(posedge clock) begin
		if (hcount ==0 && vcount == 0) begin
			min_hcount <= SCREENH; // reset values
			max_hcount <= 0;
			min_vcount <= SCREENV;
			max_vcount <= 0;
			exists_light <= 0;
			if (exists_light) begin // give an average only if there is a light
				avg_x <= SCREENH-((min_hcount+max_hcount)/2-MINH)*2; // find
average
				avg_y <= ((min_vcount+max_vcount)/2-MINV)*2;
			end
			else begin // if there is no light, put the point somewhere off screen
				avg_x <= 1200;
				avg_y <= 800;
			end
		end
		if (is_bright && hcount > MINH && hcount < MAXH && vcount < MAXV && vcount >
MINV) begin
			if (hcount < min_hcount) min_hcount <= hcount;
			if (hcount > max_hcount) max_hcount <= hcount;
			if (vcount < min_vcount) min_vcount <= vcount;
			if (vcount > max_vcount) max_vcount <= vcount;
			exists_light <= 1;
		end
	end

endmodule
```

```
////////////////////////////////////////////////////////////////////////
//
// average value for right player
//
////////////////////////////////////////////////////////////////////////


module average_l(clock, hcount, vcount, is_bright, avg_x, avg_y);
	input clock;
	input [10:0] hcount;
	input [9:0] vcount;
	input is_bright;
	output reg [10:0] avg_x;
	output reg [9:0] avg_y;

	reg [10:0] min_hcount = 1024;
	reg [10:0] max_hcount = 0;
	reg [9:0] min_vcount = 768;
	reg [9:0] max_vcount = 0;

	reg exists_light = 0;

	parameter SCREENH = 1024;
	parameter SCREENV = 768;

	parameter MINH = 36+(SCREENH/4);
	parameter MAXH = MINH+(SCREENH/4);
	parameter MINV = 74;
	parameter MAXV = MINV+SCREENV/2;




	always @(posedge clock) begin
		if (hcount ==0 && vcount == 0) begin
			min_hcount <= SCREENH; // reset values
			max_hcount <= 0;
			min_vcount <= SCREENV;
			max_vcount <= 0;
			exists_light <= 0;
			if (exists_light) begin // give an average only if there is a light
				avg_x <= SCREENH/2-((min_hcount+max_hcount)/2-MINH)*2; // find
average
				avg_y <= ((min_vcount+max_vcount)/2-MINV)*2;
			end
			else begin // if there is no light, put the point somewhere off screen
				avg_x <= 1200;
				avg_y <= 800;
			end
		end
		if (is_bright && hcount > MINH && hcount < MAXH && vcount < MAXV && vcount >
MINV) begin
			if (hcount < min_hcount) min_hcount <= hcount;
			if (hcount > max_hcount) max_hcount <= hcount;
			if (vcount < min_vcount) min_vcount <= vcount;
			if (vcount > max_vcount) max_vcount <= vcount;
```

```
                        exists_light <= 1;
                end
        end

endmodule

////////////////////////////////////////////////////////////////////////////////
//
//memory of blocks visited, to calculate spell number
//
///////////////////////////////////////////////////////////////////////////////

module blockTracker
   (input clock,
        input reset,
        input [3:0] blockNum,
        output reg[9:0] sum);

        reg [3:0] block[0:2];

   always @ (posedge clock) begin
        if (reset) begin
                block[0] <= 0;
                block[1] <= 0;
                block[2] <= 0;
        end
        if (block[2] !== blockNum)begin
                block[2] <= blockNum;
                block[1] <= block[2];
                block[0] <= block[1];
        end
        sum <= 100*block[0] + 10*block[1] + block[2];
        end
endmodule


/////////////////////////////////////////////////////////////////////////////
//
//divide the screen, tell me where we are, right now we have 9 blocks
//
/////////////////////////////////////////////////////////////////////////////

module division
   (input [10:0] x,
        input [9:0] y,
        input clock,
        output reg [3:0] blockNum);

   always @ (posedge clock) begin
        if (x >= 0 && x <= 340 && y >= 0 && y <= 256) begin
                blockNum <= 1;
        end
        else if (x >= 341 && x <= 681 && y >= 0 && y <= 256) begin
                blockNum <= 2;
        end
        else if (x >= 682 && x <= 1023 && y >= 0 && y <= 256) begin
                blockNum <= 3;
        end
        else if (x >= 0 && x <= 340 && y >= 257 && y <= 512) begin
                blockNum <= 4;
        end
        else if (x >= 341 && x <= 681 && y >= 257 && y <= 512) begin
                blockNum <= 5;
        end
        else if (x >= 682 && x <= 1023 && y >= 257 && y <= 512) begin
                blockNum <= 6;
        end
        else if (x >= 0 && x <= 340 && y >= 513 && y <= 767) begin
                blockNum <= 7;
        end
        else if (x >= 341 && x <= 681 && y >= 513 && y <= 767) begin
                blockNum <= 8;
        end
        else if (x >= 682 && x <= 1023 && y >= 513 && y <= 767) begin
                blockNum <= 9;
        end
        end
endmodule


///////////////////////////////////////////////////////////////////////////////
//
// Lumos and Knox. Toggle an LED switch each time the sum is 1
//
///////////////////////////////////////////////////////////////////////////////

module lumos
        (input [9:0] sum,
        input clock,
        output reg led_on);

        reg [9:0] prev_sum = 0;

        parameter LUMOS_SUM = 123;
        parameter KNOX_SUM = 147;

        always @(posedge clock) begin
                prev_sum <= sum;
                if ((sum == LUMOS_SUM) && (prev_sum != LUMOS_SUM)) led_on <= 0;
                else if ((sum == KNOX_SUM) && (prev_sum != KNOX_SUM)) led_on <= 1;

        end

endmodule



////////////////////////////////////////////////////////////////////////////////
//
// Wingardium Leviosa. Make object move up and then have it fall
//
////////////////////////////////////////////////////////////////////////////////

module leviosa
```

```
        (input [9:0] sum,
        input clock,
        output reg [9:0] y);

        reg [9:0] prev_sum = 0;
        reg animation = 0; // keeps track of whether the animation is currently playing
        reg [15:0] timer = 0;
        reg up = 1;
        parameter LEVIOSA_SUM = 852;

        always @(posedge clock) begin
                prev_sum <= sum;
                if ((sum == LEVIOSA_SUM) && (prev_sum != LEVIOSA_SUM) && (animation == 0))
begin
                        animation <= 1;
                        up <= 1;
                end
                if (animation == 1) begin
                        timer <= timer+1;
                        if ((timer == 1) && (up == 1)) y <= y-1;
                        else if ((timer == 1) && (up == 0)) y <= y+1;
                        if (y == 100) up <= 0;
                        else if (y == 701) animation <= 0;
                end
                else y <= 700;
        end
endmodule



////////////////////////////////////////////////////////////////////////////////
//
// Engorgio: Make a block bigger
//
////////////////////////////////////////////////////////////////////////////////

module engorgio
        (input [9:0] sum,
        input clock,
        output reg [9:0] x,
        output reg [9:0] size);

        reg [9:0] prev_sum = 0;
        reg animation = 0; // keeps track of whether the animation is currently playing
        reg [16:0] timer = 0;
        reg bigger = 1;
        parameter ENGORGIO_SUM = 256;

        always @(posedge clock) begin
                prev_sum <= sum;
                if ((sum == ENGORGIO_SUM ) && (prev_sum != ENGORGIO_SUM ) && (animation == 0))
begin
                        animation <= 1;
                        bigger <= 1;
                end
                if (animation == 1) begin
                        timer <= timer+1;
                        if ((timer == 1) && (bigger == 1)) begin
                                x <= x-1;
                                size <= size+2;
                        end
                        else if ((timer == 1) && (bigger == 0)) begin
                                x <= x+1;
                                size <= size-2;
                        end
                        if (x == 400) bigger <= 0;
                        else if (x == 501) animation <= 0;
                end
                else begin
                        x <= 500;
                        size <= 20;
                end
        end
endmodule

////////////////////////////////////////////////////////////////////////////////////
//
//Aparecium spell for "Hi!"
//
/////////////////////////////////////////////////////////////////////////////////////

module aparecium
   (input clock,
        input [31:0] sum,
        input [10:0] hcount,
        input [9:0] vcount,
        input [23:0] color,
        output reg [23:0] pixel);

        reg [7:0] R;
        reg [7:0] B;
        reg [7:0] G;
        reg [31:0] counter = 0;
        reg [31:0] alpha;

         parameter APARECIUM_SUM = 236;

   always @ (posedge clock) begin
        if (sum == APARECIUM_SUM) begin
                counter <= counter + 1;
                if (((hcount >= 479 && hcount <= 483) &&
                (vcount >= 367 && vcount <= 399)) |
                ((hcount >= 463 && hcount <= 467) &&
                (vcount >= 367 && vcount <= 399)) |
                ((hcount >= 467 && hcount <= 479) &&
                (vcount >= 381 && vcount <= 385)) |
                ((hcount >= 489 && hcount <= 493) &&
                (vcount >= 381 && vcount <= 399)) |
                ((hcount >= 489 && hcount <= 493) &&
                (vcount >= 367 && vcount <= 371)) |
                ((hcount >= 499 && hcount <= 503) &&
                (vcount >= 367 && vcount <= 389)) |
                ((hcount >= 499 && hcount <= 503) &&
                (vcount >= 395 && vcount <= 399))) begin
```

```
                        if (counter <= 25000000) begin
                        alpha <= 4;
                        end
                        else if (counter <= 50000000) begin
                        alpha <= 3;
                        end
                        else if (counter <= 75000000) begin
                        alpha <= 2;
                        end
                        else if (counter <= 100000000) begin
                        alpha <= 1;
                        end
                        else if (counter <= 125000000) begin
                        alpha <= 0;
                        end

                        R <= color[23:15] >> (2*alpha);
                        G <= color[15:7] >> (3*alpha);
                        B <= color[7:0] >> alpha;
                        pixel <= {R,G,B};
                end
                else pixel <= 0;
        end
        else begin
                pixel <= 0;
                counter <= 0;
        end
        end
endmodule



////////////////////////////////////////////////////////////////////////////////////
//
//Accio Spell
//
////////////////////////////////////////////////////////////////////////////////////

module accio
   (input clock,
        input [10:0] hcount,
        input [9:0] vcount,
        input [31:0] sum,
        input [10:0] wandX,
        input [9:0] wandY,
        output reg [10:0] newX = 750,
        output reg [9:0] newY = 703);

        reg [15:0] xspeed = 4;
        reg [15:0] yspeed = 4;
        reg [15:0] counter;
        reg [15:0] counter2;

         parameter ACCIO_SUM = 145;

   always @ (posedge clock) begin
```

```
	if (sum == ACCIO_SUM) begin
		xspeed <= (750 - wandX) >>> 6;
		yspeed <= (703 - wandY) >>> 6;
		if (hcount == 1024 && vcount == 768 && counter < 65) begin
		counter <= counter + 1;
		newX <= newX - xspeed;
		newY <= newY - yspeed;
		end
	end
	else begin
		counter <= 0;
		newX <= 750;
		newY <= 703;
	end
	end
endmodule



////////////////////////////////////////////////////////////////////////////////////
//
// Two-Player Setup
//
/////////////////////////////////////////////////////////////////////////////////////

module twoPlayerSetup
   (input clock,
	input [10:0] hcount,
	input [9:0] vcount,
	input [23:0] color,
	output reg [23:0] pixel);

	always @ (posedge clock) begin
	if (((hcount >= 510 && hcount <= 512) && //Middle bar
		(vcount >= 0 && vcount <= 767))|

		((hcount >= 15 && hcount <= 495)&&   //Top bars
		(vcount >= 15 && vcount <= 16))|
		((hcount >= 527 && hcount <= 1008)&&
		(vcount >= 15 && vcount <= 16))|

		((hcount >= 15 && hcount <= 495)&&   //Bottom bars
		(vcount >= 30 && vcount <= 31))|
		((hcount >= 527 && hcount <= 1008)&&
		(vcount >= 30 && vcount <= 31))|

		((hcount >= 15 && hcount <= 16)&&    //left, down bars
		(vcount >= 15 && vcount <= 30))|
		((hcount >= 494 && hcount <= 495)&&
		(vcount >= 15 && vcount <= 30))|

		((hcount >= 527 && hcount <= 528)&&  //right, down bars
		(vcount >= 15 && vcount <= 30))|
		((hcount >= 1007 && hcount <= 1008)&&
		(vcount >= 15 && vcount <= 30))) begin

		pixel <= color;
```

```
        end
        else pixel <= 0;
        end
endmodule

////////////////////////////////////////////////////////////////////
//
// Display LEDs to decide winner
//
////////////////////////////////////////////////////////////////////

module winner
        (input clock, reset,
         input [31:0] left, right,
         output reg led_left,
         output reg led_right);

         reg game_over = 0;

         always @(posedge clock) begin
                if (reset) begin
                        game_over <= 0;
                        led_left <= 0;
                        led_right <= 0;
                        end
                else if ((game_over == 0) && (left >= 475)) begin
                        led_left <= 1;
                        game_over <= 1;
                        end
                else if ((game_over == 0) && (right >= 475)) begin
                        led_right <= 1;
                        game_over <= 1;
                        end
         end

endmodule


///////////////////////////////////////////////////////////////////
//
// Health Bars: this is where we will change the size of the health bars
//
/////////////////////////////////////////////////////////////////

module healthBars
   (input clock,
          input reset,
        input [10:0] hcount,
        input [9:0] vcount,
        input [23:0] color,
        input [31:0] left,
        input [31:0] right,
        output reg [23:0] pixel);


         // Color the health bars to change color as life decreases!
        wire [7:0] left_red = left[8:1];
```

```
         wire [7:0] left_green = 255-left[8:1];
         wire [23:0] left_color = {left_red,left_green,8'h00};

         wire [7:0] right_red = right[8:1];
         wire [7:0] right_green = 255-right[8:1];
         wire [23:0] right_color = {right_red,right_green,8'h00};

         always @ (posedge clock) begin
                 //reset game
                 if (reset)begin
                         if ((hcount >= 18 && hcount <= (492)) && //left bar
                                        (vcount >= 18 && vcount <= 28)) pixel <= left_color;

                         else if ((hcount >= 530 && hcount <= (1005))&&     //right bar
                                        (vcount >= 18 && vcount <= 28)) pixel <= right_color;
                         else pixel <= 0;
                 end

        //end game
        else if (left >= 492) begin
                if ((hcount >= 18 && hcount <= 492) && //left bar
                                        (vcount >= 18 && vcount <= 28)) pixel <= 0;
        end
        else if (right >= 495) begin
                if ((hcount >= 530 && hcount <= (1005))&&    //right bar
                                        (vcount >= 18 && vcount <= 28)) pixel <= 0;
        end

        //update as we take hits
        else if ((hcount >= 18 && hcount <= (492-left)) && //left bar
                (vcount >= 18 && vcount <= 28)) pixel <= left_color;

        else if ((hcount >= 530 && hcount <= (1005-right))&&        //right bar
                (vcount >= 18 && vcount <= 28)) pixel <= right_color;

        else pixel <= 0;
        end
endmodule


/////////////////////////////////////////////////////////////
//
// Hit! Given the spell sum, tell me what hit the opponent takes
//
////////////////////////////////////////////////////////////

module hitmaker
   (input clock,
         input reset,
        input [15:0] sum,
        input [15:0] blockStart,
        output reg [31:0] hit = 0);

        always @ (posedge clock) begin
                 if (reset) begin
                              hit <= 0;
                 end
        else if (sum == 37 && blockStart == 11) begin
               hit <= 20;
        end
        else if (sum == 44 && blockStart == 13) begin
               hit <= 30;
        end
        else if (sum == 45 && blockStart == 14) begin
               hit <= 40;
        end
        else if (sum == 51 && blockStart == 16) begin
               hit <= 50;
        end
        else if (sum == 48 && blockStart == 17) begin
               hit <= 45;
        end
        else if (sum == 45 && blockStart == 13) begin
               hit <= 35;
        end
        else if (sum == 42 && blockStart == 12) begin
               hit <= 12;
        end
        else if (sum == 38 && blockStart == 11) begin
               hit <= 23;
        end
        end

endmodule



////////////////////////////////////////////////////////////////////////////////
//
// memory of hits taken, to calculate what the healthbar should look like
//
//////////////////////////////////////////////////////////////////////////////////

module healthbarChanger
   (input clock,
        input reset,
        input [31:0] hit,
        output reg [31:0] sum);

        reg [31:0] block[0:39];
        reg [15:0] a;

   always @ (posedge clock) begin
        if (reset) begin
               block[39] <= 0;
               block[38] <= 0;
               block[37] <= 0;
               block[36] <= 0;
               block[35] <= 0;
               block[34] <= 0;
               block[33] <= 0;
               block[32] <= 0;
               block[31] <= 0;
               block[30] <= 0;
               block[29] <= 0;
               block[28] <= 0;
               block[27] <= 0;
               block[26] <= 0;
               block[25] <= 0;
               block[24] <= 0;
               block[23] <= 0;
               block[22] <= 0;
               block[21] <= 0;
               block[20] <= 0;
               block[19] <= 0;
               block[18] <= 0;
               block[17] <= 0;
               block[16] <= 0;
               block[15] <= 0;
               block[14] <= 0;
               block[13] <= 0;
               block[12] <= 0;
               block[11] <= 0;
               block[10] <= 0;
               block[9] <= 0;
               block[8] <= 0;
               block[7] <= 0;
               block[6] <= 0;
               block[5] <= 0;
               block[4] <= 0;
               block[3] <= 0;
               block[2] <= 0;
               block[1] <= 0;
               block[0] <= 0;
                               sum <= 0;
        end
        if (block[0] !== hit)begin
               block[39] <= block[38];
               block[38] <= block[37];
               block[37] <= block[36];
               block[36] <= block[35];
               block[35] <= block[34];
               block[34] <= block[33];
               block[33] <= block[32];
               block[32] <= block[31];
               block[31] <= block[30];
               block[30] <= block[29];
               block[29] <= block[28];
               block[28] <= block[27];
               block[27] <= block[26];
               block[26] <= block[25];
               block[25] <= block[24];
               block[24] <= block[23];
               block[23] <= block[22];
               block[22] <= block[21];
               block[21] <= block[20];
               block[20] <= block[19];
               block[19] <= block[18];
               block[18] <= block[17];
               block[17] <= block[16];
               block[16] <= block[15];
```

```
                block[15] <= block[14];
                block[14] <= block[13];
                block[13] <= block[12];
                block[12] <= block[11];
                block[11] <= block[10];
                block[10] <= block[9];
                block[9] <= block[8];
                block[8] <= block[7];
                block[7] <= block[6];
                block[6] <= block[5];
                block[5] <= block[4];
                block[4] <= block[3];
                block[3] <= block[2];
                block[2] <= block[1];
                block[1] <= block[0];
                block[0] <= hit;
        end

        sum <= block[0] + block[1] + block[2] + block[3] + block[4] +
                        block[5] + block[6] + block[7] + block[8] + block[9] +
                        block[10] + block[11] + block[12] + block[13] + block[14] +
                        block[15] + block[16] + block[17] + block[18] + block[19] +
                        block[20] + block[21] + block[22] + block[23] + block[24] +
                        block[25] + block[26] + block[27] + block[28] + block[29] +
                        block[30] + block[31] + block[32] + block[33] + block[34] +
                        block[35] + block[36] + block[37] + block[38] + block[39];
        end
endmodule


////////////////////////////////////////////////////////////////
//
// memory of blocks visited, to calculate spell number
//
////////////////////////////////////////////////////////////////

module blockTrackerTP
   (input clock,
        input reset,
        input [15:0] blockNum,
        output reg [15:0] sum,
        output reg [15:0] blockStart);

        reg [15:0] block[0:2];

   always @ (posedge clock) begin
        if (reset) begin
                block[0] <= 0;
                block[1] <= 0;
                block[2] <= 0;
        end
        if (block[2] !== blockNum)begin
                block[2] <= blockNum;
                block[1] <= block[2];
                block[0] <= block[1];
        end
        sum <= block[0] + block[1] + block[2];
        blockStart <= block[0];
        end
endmodule


//////////////////////////////////////////////////////////////////////////
//
// These modules divide eahc half of the screen
//
///////////////////////////////////////////////////////////////////////////

module divisionTPLeft
   (input [10:0] x,
        input [9:0] y,
        input clock,
        output reg [15:0] blockNum);

   always @ (posedge clock) begin
        if (x >= 0 && x <= 255 && y >= 0 && y <= 191) begin
               blockNum <= 11;
        end
        else if (x >= 0 && x <= 255 && y >= 192 && y <= 383) begin
               blockNum <= 13;
        end
        else if (x >= 0 && x <= 255 && y >= 384 && y <= 575) begin
               blockNum <= 15;
        end
        else if (x >= 0 && x <= 255 && y >= 576 && y <= 767) begin
               blockNum <= 17;
        end
        else if (x >= 256 && x <= 510 && y >= 0 && y <= 191) begin
               blockNum <= 12;
        end
        else if (x >= 256 && x <= 510 && y >= 192 && y <= 383) begin
               blockNum <= 14;
        end
        else if (x >= 256 && x <= 510 && y >= 384 && y <= 575) begin
               blockNum <= 16;
        end
        else if (x >= 256 && x <= 510 && y >= 576 && y <= 767) begin
               blockNum <= 18;
        end
        else blockNum <= 0;
        end
endmodule

module divisionTPRight
   (input [10:0] x,
        input [9:0] y,
        input clock,
        output reg [15:0] blockNum);

   always @ (posedge clock) begin
        if (x >= 512 && x <= 767 && y >= 0 && y <= 191) begin
               blockNum <= 11;
        end
        else if (x >= 512 && x <= 767 && y >= 192 && y <= 383) begin
                blockNum <= 13;
        end
        else if (x >= 512 && x <= 767 && y >= 384 && y <= 575) begin
                blockNum <= 15;
        end
        else if (x >= 512 && x <= 767 && y >= 576 && y <= 767) begin
                blockNum <= 17;
        end
        else if (x >= 768 && x <= 1023 && y >= 0 && y <= 191) begin
                blockNum <= 12;
        end
        else if (x >= 768 && x <= 1023 && y >= 192 && y <= 383) begin
                blockNum <= 14;
        end
        else if (x >= 768 && x <= 1023 && y >= 384 && y <= 575) begin
                blockNum <= 16;
        end
        else if (x >= 768 && x <= 1023 && y >= 576 && y <= 767) begin
                blockNum <= 18;
        end
        else blockNum <= 0;
        end
endmodule
```

Appendix Verilog Integration Code:

Here is some of the code used to integrate the modules.

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

       // One-Player Mode!


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

       // average module
       wire [10:0] avg_x;
       wire [9:0] avg_y;
       average avg(clk, hcount, vcount, is_bright, avg_x, avg_y);

       parameter RED = 24'hFF0000; // Define a bunch of colors for convenience
       parameter GREEN = 24'h00FF00;
       parameter BLUE = 24'h0000FF;
       parameter WHITE = 24'hFFFFFF;
       parameter BLACK = 24'h000000;

       wire [23:0] wand_pixel;
       wand_sprite twinkle(.x(avg_x),.y(avg_y),.clock(clk),.hcount(hcount),.vcount(vcount),
                              .pixel(wand_pixel),.color(RED+GREEN),.size(10));



       // divide screen and track spell
       wire [3:0] blockNum;
       wire [9:0] sum;
       division div(avg_x, avg_y, clk, blockNum);
       blockTracker track(.clock(clk), .reset(reset), .blockNum(blockNum), .sum(sum));

       // Lumos and Knox
       lumos light(sum, clk, led);

       // Wingardium Leviosa
       wire[9:0] wing_y;
       wire [23:0] wing_pixel;
       blob
wing(.x(600),.y(wing_y),.hcount(hcount),.vcount(vcount),.color(BLUE),.pixel(wing_pixel),.width
(40),.height(40));
       leviosa wingardium(sum, clk, wing_y);

       // Engorgio
       wire [9:0] pos;
       wire [9:0] size;
       wire [23:0] eng_pixel;
```

```
	blob
big(.x(pos),.y(pos),.hcount(hcount),.vcount(vcount),.color(RED),.pixel(eng_pixel),.width(size)
,.height(size));
	engorgio eng(sum, clk, pos, size);

	// Aparecium
	wire [23:0] apar_pixel;
	aparecium
apa(.clock(clk),.sum(sum),.hcount(hcount),.vcount(vcount),.color(GREEN),.pixel(apar_pixel));

   // Accio
	wire [23:0] acc_pixel;
	wire [10:0] acc_x;
	wire [9:0] acc_y;
	blob
object(.x(acc_x),.y(acc_y),.hcount(hcount),.vcount(vcount),.color(RED+GREEN),.pixel(acc_pixel)
,.width(40),.height(40));
	accio
acc(.clock(clk),.hcount(hcount),.vcount(vcount),.sum(sum),.wandX(avg_x),.wandY(avg_y),.newX(ac
c_x),.newY(acc_y));


	// Display all pixels
	wire [23:0] vr_pixel = wand_pixel | wing_pixel | eng_pixel | apar_pixel | acc_pixel;



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

	// Two-Player Mode!


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

	// Find Wands
	wire [10:0] l_x;
	wire [9:0] l_y;
	average_l avg_l(clk, hcount, vcount, is_bright, l_x, l_y);

	wire [10:0] r_x;
	wire [9:0] r_y;
	average_r avg_r(clk, hcount, vcount, is_bright, r_x, r_y);


	// Create Wands
	wire [23:0] l_wand_pixel;
	wand_sprite l_twinkle(.x(l_x),.y(l_y),.clock(clk),.hcount(hcount),.vcount(vcount),
			      .pixel(l_wand_pixel),.color(RED),.size(10));

	wire [23:0] r_wand_pixel;
	wand_sprite r_twinkle(.x(r_x),.y(r_y),.clock(clk),.hcount(hcount),.vcount(vcount),
			      .pixel(r_wand_pixel),.color(BLUE),.size(10));


	// Setup Two-Player Screen
	wire [23:0] setup_pixel;
	twoPlayerSetup
set(.clock(clk),.hcount(hcount),.vcount(vcount),.color(WHITE),.pixel(setup_pixel));


	// Get block numbers from x and y values
	wire [15:0] l_block;
	wire [15:0] r_block;
	divisionTPLeft divl(.x(l_x),.y(l_y),.clock(clk),.blockNum(l_block));
	divisionTPRight divr(.x(r_x),.y(r_y),.clock(clk),.blockNum(r_block));


	// Track Blocks to get sums
	wire [15:0] lsum;
	wire [15:0] rsum;
	wire [15:0] lblockstart;
	wire [15:0] rblockstart;
	blockTrackerTP
ltrack(.clock(clk),.reset(reset),.blockNum(l_block),.sum(lsum),.blockStart(lblockstart));
	blockTrackerTP
rtrack(.clock(clk),.reset(reset),.blockNum(r_block),.sum(rsum),.blockStart(rblockstart));

	// Make the hits!
	wire [31:0] l_hit;
	wire [31:0] r_hit;
	hitmaker
lhit(.clock(clk),.reset(reset),.sum(lsum),.blockStart(lblockstart),.hit(l_hit));
	hitmaker
rhit(.clock(clk),.reset(reset),.sum(rsum),.blockStart(rblockstart),.hit(r_hit));


	// Calculate healthbars
	wire [31:0] left_bar_sum;
	wire [31:0] right_bar_sum;
	healthbarChanger lbar(.clock(clk),.reset(reset),.hit(l_hit),.sum(right_bar_sum));
	healthbarChanger rbar(.clock(clk),.reset(reset),.hit(r_hit),.sum(left_bar_sum));


	// Create the Health Bars
	wire [23:0] bar_pixel;
	healthBars
bar(.clock(clk),.hcount(hcount),.vcount(vcount),.color(GREEN),.left(left_bar_sum),.right(right
_bar_sum),.pixel(bar_pixel));


	// Use LEDs to display the winner
	wire left_score;
	wire right_score;
	winner
win(.clock(clk),.reset(reset),.left(left_bar_sum),.right(right_bar_sum),.led_left(right_score)
,.led_right(left_score));

	always @(posedge clk)
		dispdata <= {31'b0,left_score,31'b0,right_score};


	// Display all pixels
```

```
wire [23:0] two_pixel = l_wand_pixel | r_wand_pixel | setup_pixel | bar_pixel;
```