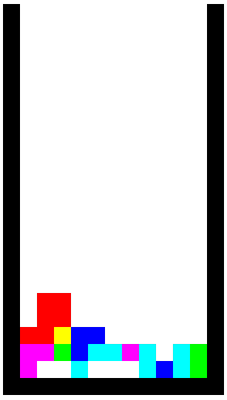
Final Writeup: Spectris



Intro:
For our project, we will be implementing Spectris, our own version of the quintessential video game Tetris. To move beyond the original version of Tetris, our game features a new twist: The blocks overlap depending on their colors. For example, a red block could be placed in the same location as a blue block, and those overlapped squares would turn magenta. Then, a green block could make those magenta blocks white, but blue and red blocks would collide with them. This makes for a new visual aspect and a new gameplay challenge. Besides the core Tetris gameplay, unique gameplay twist, and visuals, three other baseline features are sounds/music, high scores, and a color-blind mode.

The game will be controlled in one of two ways. Using the switches in the Nexys 4 DDR and having some gesture controls. The project was divided into 3 parts. The game logic, the controller, and the audio.

Initial Commit Goals:

- Functioning spectris, complete with generic tetris controls (Nexys 4 buttons), and block collision / line clear based on color.
- All basic controls using Nexys 4 controller.
- 2 background music tracks and 2 sound effects

Initial Goals:

- Glove gesture controller
- Keeping Score, displayed as a number on the side next to the tetromino queue (most likely to be # of lines cleared.
- List future blocks (tetromino queue).
- Main Menu with settings for Difficulty and variable fall speed.
- Hold function, where the player can hold a piece for later.
- Glove Control Scheme
- Accelerated music when the player is close to losing at the top

Initial Stretch Goals (try to implement some, but probably not all):

- Colorblind Mode / alternate color schemes
- Options to choose movement speed (default will be reasonable)
- Options to choose number of colors (for example 2 colors could be grey+grey = white, or red + cyan = white, as opposed to the standard red + green + blue = white)
- Gradually increasing difficulty, along with score manipulation, possibly different scoring based on more lines cleared at a time as well as difficulty
- Fancy intro screen animations.
- All block edge logic for whether a piece is still falling or landed with the rest of the stationary blocks, e.g. grey border for moving blocks, and the entire ground has a white border around it.

- Highscores List

Project overview:

Our game Spectris can be divided into 3 major sections: The controller inputs, the game logic, and the audio/video outputs. Each section has core features as well as expanded features that we wish to implement for extra interest / entertainment value. The project will be constructed on a Nexys 4 board.

For the inputs to our game, the Nexys 4 switches and buttons will work as debug / simple controls, which will including moving left, right, down, and turning. But for a more interesting player experience, we will also create a motion controller that allows the player to control the game with a hand-worn apparatus. This control glove would be viewed by a camera, which would connect to a module that converts the viewed image into game inputs.

The glove/camera apparatus is going to be able to detect movements in the player's hand in the following way: If the module detects that the player has moved his hand to the left of to the right it will map it to the direction the block should move. If the module detects that the rotation gesture has happened then this controller module will send a signal to the game logic telling it to rotate the block in the specified direction (clockwise vs counterclockwise). If again the module detects that the send-block-down gesture has been established, then the module will send a signal to the game logic telling it to accelerate it downwards.

The Camera module will detect if any of the gestures have been detected. The gesture include: moved left, moved right, moved down, no movement and a rotation gesture.
If the rotation gesture has been detected, the piece will be rotated 90 degrees. If the down gesture has been detected, then the piece will move all the way down. If either left or right has been detected the controller module will start a timer and if the timer is greater than a parameter the block will move in that direction and the signal bit will be reset to 0 and wait until the player had moved in the same direction for some time (passed the time parameter). If the controller module detects no valid gesture then the block will continue to move downward as normal.

The game logic is based on a finite state machine that changes state depending on player inputs or whether the game has ended (in game). The game outputs to the video and audio are as stated above and below. The FSM randomly chooses which Tetris block to use, and takes in how fast the blocks are falling, and whether a rotation or translation is occurring. In return, the FSM will update the current falling block's position, performing any translations and rotations if needed.

The tetris game board will be 10 blocks long by 20 blocks high, as per a normal tetris board. Each block is fully filled by a single color, so 200 RGB values must be stored. Memory used will be on the order of 10kB for full RGB values. However, for the pure basic version in which only three colors are RGB, each block's color value can can be simplified to three bits, representing each of Red, Green, and Blue. These are drawn by mapping the playing field grid to the pixel representation.

Falling tetrominoes will collide based on their color. For example, two red blocks cannot go through each other, so if a falling red tetromino collides from falling anywhere on a red block, that tetromino will lock into place; the next tetromino will spawn and begin falling. A red block and green block will overlap to make yellow, which can then overlap with a blue block to make white (red + green + blue). Only a full line of white blocks (sum of all available colors) can clear a line. As per normal tetris, the game is over when any blocks reach the top of the screen.

All of the game logic and settings can be troubleshooted / debugged / tested using the Nexys 4 and VGA connected monitor.

The audio will be synthesized using bitcodes saved in memory, rather than played from a stored waveform. Every certain number of cycles, the audio module loads and executes a bitcode, which will turn a note on or off at a certain pitch for a particular channel. The game FSM determines which background music should be playing; it also determines when sound effects need to be played, at which point the bitcode for the sound effect will take precedence over the background music. Each sound effect only uses 1 channel, however, so most of the music will not cut out when a sound effect plays.

The audio module will have 4 channels: 2 square waves, 1 triangle wave, and 1 sawtooth wave. These channels, which are either off or generating a wave of a certain frequency, are summed together and outputted to the audio out of the Nexys 4.

The music and sound effects will be composed externally, then converted to bitcodes for the game. This will allow us to have higher quality audio without spending memory on audio samples. Each bitcode is 14 bits in size: 2 bits for which channel, 1 bit for on/off, 7-bit pitch value, 4-bit volume value. If the audio system reads bitcodes at a frequency of 32 Hz, then 1 minute of audio data for all 4 channels takes up 13.44 kB. For reference, 1 minute of 16 bit PCM data sampled at 44.1 kHz would require about 5.292 MB of space.

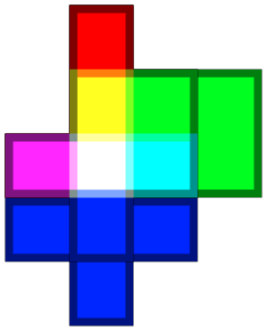
There will be at least two different choices for background music, and sound effects for cleared lines, locking tetrominoes, game over, and menu selections**.**

Game Logic (Benny):

The Spectris game logic features the standard movement patterns from generic Tetris combined with the additional factor that all blocks can be three colors: red, green, and blue. Each color only overlaps with itself, such that a green block falling into a red block will create a yellow block, and all three colors together creates a white block. Similar to regular tetris, having a full line, in this case a full line of white blocks, clears that line and moves the rest of the board downwards. Scoring in the game consists of getting 1 point for every block placed, and 100 points for each successfully cleared line.

Normal Tetris:

Spectris:



One of the most important aspects of implementing this game was choosing the proper representation. Our chosen game representation is setup is as follows. The game board itself consists of two arrays, one for the current moving block, and one for the background formed from previous blocks. Each array is 10 by 20 by 3 bits, as a standard Tetris board is 10 blocks wide and 20 blocks high, and each of the three bits represents a color (RGB) as a binary 1 or 0. The first of these two arrays corresponded with the placement of the current moving block, meaning that only 4 out of the 200 3-bit memories would be nonzero at a time. The second of these two arrays corresponded with the background, in which each x-y point could correspond with any of the 8 possible colors represented by the 3 bits. Each block is 32 pixels by 32 pixels, resulting in a total board length of 320 and height of 640 pixels, which fit quite well into the display size. Because the display height was 768 pixels = 32*24, the game field border has a width of one block (32 pixels), with a one block gap above and below. The advantage to using this type of representation is that first off, it doesn't take a lot of memory to save, nor to display. If the game representation instead uses something similar to copies of the blob module from previous labs, in order to display the board, the logic would have to look through all instances of the object so see which the current pixel would display, using VGA display logic. With this representation, with a simple check of whether the pixel is within the game board area, the block that a pixel corresponds to can be checked simply by taking the relative distance from a corner of the board and taking off the five least significant digits, since each block is 32 pixels by 32 pixels. In addition, the colors that blocks represent are very easy to change. When adding two colors, because colors are represented within that block as either a value of 1, 2, or 4, correlating to 001, 010, 100 in binary, additions of colors can be easily mapped, as the sum of colors yellow, cyan, violet, and white would be represented as 110, 101, 011, and 111. This allows for ease in mapping the game representation to the 24 bit RGB value of the pixel to display. In terms of block representation, instead of representing a moving tetronimo as four blocks, each tetromino was represented in these five variables:

1. Piece Shape ("T" shape, "L" shape, "O" shape, etc.)
2. Piece Color (In this case, R G B represented as a 0, 1, or 2)
3. Piece rotation (0-3 for the four possible orthogonal rotations)
4. Piece x-position within the game area (0-9)
5. Piece y-position within the game area (0-19)

This way, any possible movement within the game is easily changed. A movement left or right results in a change of 1 in the x-variable, a downward issued movement results in y=y+1, and rotating a piece is simply adding 1 to the piece rotation variable.

With regards to timing the visual aspect, although the audio runs on a 100MHz clock, for VGA to display the game, a 65MHz clock needed to be used, which was synthesized from the faster clock to run the 768 by 1024 VGA display.

The rest of the visuals, that is, the game border, as well as words and score shown on screen are created either as sprites, or as instances of the blob module. The four walls of the border are each their own blob module, and each line of words, such as “Spectris”, “Score”, or “Hold”, is its own sprite, whose color is determined by the color of the current falling block (or in the case of “hold”, the color of the held block”). Each digit of the score is its own instance of a different sprite as well. Each digit sprite takes in the BCD value of the number it needs to represent, which is transformed from binary to BCD via a separate module, similar to that of Pset8. (This is where I learned the very important lesson of never assigning a wire an initial value).

Now onto the meaty part: the actual game logic.

Because the game runs in 65MHz, but the game only needs to play in 60fps, this means the logic has plenty and plenty of cycles to determine the next state of the game. Which means that the game logic is very conducive to pipelining. THIS IS THE MOST IMPORTANT LESSON I LEARNED WHILE CODING THE LOGIC. In the early builds for the game logic, I tried to fit in all of the logic of selecting the next shape, figuring out where pieces could move, updating the two arrays and all variables representing the game, etc. all in one cycle with a gratuitous number of for-loops. This was a huge mistake, as synthesizing this logic was nearly impossible, as any machine I tried to do so on would freeze and crash more likely than not, and synthesizing times were over 30 minutes long, making debugging unfathomable. Later on, I would rewrite all of the game logic, in which the internal 65MHz clock that the game logic on was split between two variables, X and Y. On each clock cycle, X would always increment by 1, from 0 to 9, and each time X was reset, Y would increment by 1, from 0 to 23, effectively breaking down the clock into groups of 240 cycles. Within each of the first 200 cycles, that is when Y > 20, I would update everything relating to the two game boards of the corresponding pixel. For example, at x=5, y=11, I would update all logic in the “block” and “field” arrays at indices block[5][11] and field[5][11].

User controls were clocked such that movements could not be issued every cycle, or even every 240 clock cycles, as surely moving a current piece would be impossible if issuing a movement for a hundredth of a second would move it ten times. Thus, movement commands (left/right/down/turn) were only taken periodically based on the 65MHz clock, in addition to the natural fall speed of the game. By mapping the movement command of the game logic to 0 for most cycles, and only to either the user input, or downward movement every x cycles, the user could move a block on a reasonable time scale. This also allowed for the implementation of variable fall speed (and hence difficulty) by changing the delay between movement commands. In order to determine whether the user should be able to move a block, I issued a variable ‘movable’. Whenever the game registers that a movement command, such as moving left, right, down, or turning, movable is set to 1, and a separate instance of a piece, next_block was used. Next_block is created by updating the five variables representing a tetromino based on the movement issued. A rotation would indicate an increment of piece rotation, a movement right would indicate an increment of piece x-

value, and so on. During these aforementioned 200 cycles, the game logic would check each space on whether next_block was situated on that space, and if it did, whether it would collide with the background. At the end of these 200 cycles, if there was a collision, movement was of course not allowed, and if the movement was in the downwards direction, then it meant that that block should be placed, and a new block would need to spawn at the top of the screen. If there was no collision, the variables representing the current block would be updated to the variables for the next block, and the cycle would continue until the game ended.

Because each tetromino is represented as the 5 variables listed above, the game still needed to know what each piece “looked like”. To accomplish this, each block other than the long “I” piece was coded in a 4 by 3 by 3 binary format, representing the four rotations of the 3 by 3 grid that each piece could be represented in. Below is one such representation of the “T” block. The long “I” block had separate logic for much of the game, as although it didn't fit in the same 3 by 3 grid that all the other pieces did, its positions could be checked easily, as it was always either horizontal or vertical.

[0 0 0]
[1 1 1] = 9’b000_111_010
[0 1 0]

When trying to create these ‘image’ variables for each block, I found that the best way to create it was actually to make one large binary variable, similar to a sprite, that all the blocks mapped to. Thus I made a 7*4*3*3 = 252 bit variable that stored all of the images for the 7 possible blocks, and to check a particular tetromino of a particular rotation, I would extract the 9-bit sequence correlating to its 3x3 shape in in that rotation.

Of course all of this happened in the first 200 cycles of the total 240. Cycles 201-210 were where all of the actual updating logic occurred, such as setting the values of next_block based on user input (if user input was taken, which was again rare with regards to how often the game checked for updates based on the internal clock), determining whether a new block should be spawned since the previous block was placed, and updating the values of the current block based on whether movement was allowed or not.

The final clock cycles were used only if a line should be cleared or not. First, the game would check if a line needed to be cleared at all by running through each horizontal line and checking whether any of them had full white blocks (basically if all values of field[x][y] = ‘111’ for all values of x for that y). If so, it would save the line number that needed to be cleared. In the next 20 cycles, the game would then update each line from bottom to top. If the line was above or equal to the line needed to be cleared, the values of that line were set to the values of the line above. The exception is that for the topmost line, because there are no lines to fall down from, will always have a value of ‘000’ for each space within the line.

Add-on features, such as pausing the game, detecting game over, telling the audio to speed up once the block tower had built higher hough, were pretty trivial to add in. The notable two that affected normal game: holding a block and showing a next block, seemed daunting at first, but I found later on were easily implementable due to our game abstraction. Showing the next falling block simply meant inserting an intermediate between the currently moving

piece and the quasi-random algorithm used to select the next block (since “random” isn’t easily implemented in hardware). Implementing a hold feature simply required having another set of tetromino variables that would trade places with the current one whenever the use inputted a ‘hold’ command. This required setting a lock for the user, so that the user couldn’t continuously switch between two pieces indefinitely. This lock unlocks after any piece is placed, similar to modern Tetris, thus with each new block, the player may use the hold feature once.

By far, the most complicated sections of programming the game logic came from trying to tie in clearing a line in with the rest of the game, and figuring out a way to effectively turn a piece. Clearing a line effectively (that is, without nearly un-synthesizable nested for-loops) required expanding on the pipelining based on clock cycles that I implemented for updating the playing field, and effectively finding a way to turn a piece involved choosing a smart representation of the playing field as well as the current moving piece, in which checking movement could happen one space at a time instead of trying to check everything at once, by using the 252 bit variable representation for all of the 3x3 block display. Embarrassingly, when implementing the score feature, I spent several unneeded hours debugging because I had assigned a wire that carried the BCD score representation to the digit sprite module to 0, meaning no matter how the binary score changed, the sprite would always display 0. Also, I also spent many hours struggling with displaying any simple graphics before I had switched to using a 65MHz clock for VGA. All in all, choosing the proper abstraction and representation, as well as planning out clock cycles was the biggest lesson I learned in programming the game logic. If I could do it again, I would have liked to perhaps tried to implement the stretch goal of having different textures for blocks rather than colors, such as the diagonal texture I used to differentiate the game border with white block spaces. I think those looked really cool.

Controls (Alfredo):

Our game was to originally going to have two input methods.
-2 switch buttons
camera tracking
The first one was to be used for debugging and testing while the camera gesture control would be under development.
Because we were using a Nexys 4 DDR our camera of choice would be OV7660. Some of the benefits of using this camera include cheap cost a customizability. Some of the draw backs were that because of its cheap prize the camera would not reliable at times and would sometimes break without notice. Moreover, at camera would not accurately be able to depict certain colors and because it was so customizable it would at times be nearly impossible to pinpoint was exactly the problem would be and what registered would need to change during initial programming to ensure reliability within acceptable parameters.

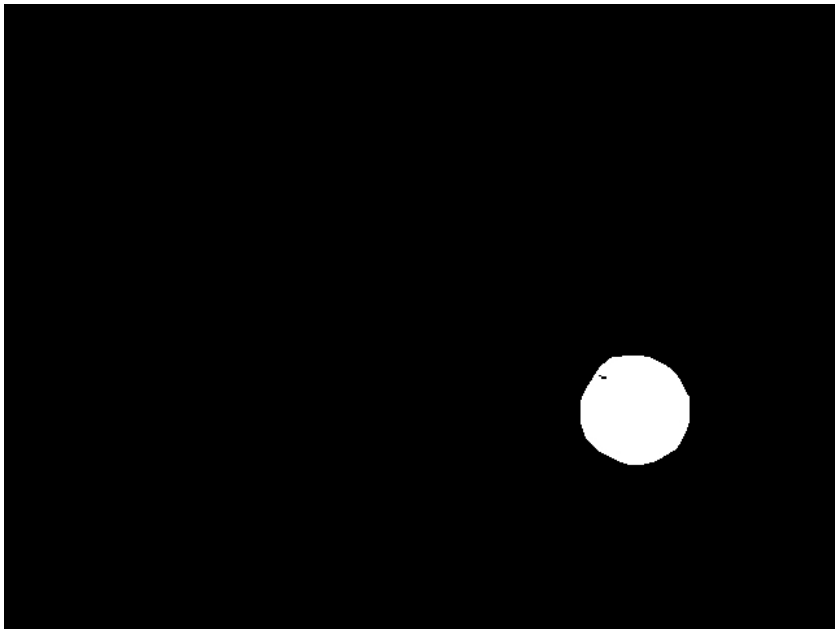
Camera overview:

Initial Camera Specs:
640 x 480 resolution
RGB color was outputed as a 565 16 bit color

The camera works as follows every clock cycle, the camera gets half of the pixel color, within two clock cycles the pixel color which should be complete. The pixel color is stored in in a16 bit number. That 16 bit number has to be transformed to a 12 bit number. Because the 16 bit number is RGB 565 meaning that the red is stored in a 5 bit number, the green in a 6, and the blue in a 5, we are able to take the Top 4 bits of each color and transform the 16 bit number to a 12 bit number that the Nexys uses to display to VGA.

Before that happens some processing must be done. Because we are doing tracking, we must first set a framework so that these pixel calculations and processing can be done.The first of these processes is selecting for a color we are going to track. With initial testing we found out that skin color is very hard to track because of how the color changes with different lighting. We then tried to segment out neon green and bright red and we encountered similar issues here, the main reason why this was unreliable was because the camera could not accurately display these pixel colors so segmenting for them was unreliable and not consistent with the reliability that we need. The best color we found that would be easily segmentable was white. We exuded to use a LED to produce this color. After we tested for the best color to process with we then began to do color segmentation.

Within our camera generator block, for every pixel i check if its above a threshold of white. In 12 bit hex, white is showcased as 12h'FFF. In order to have a robust white detection scheme I lowered the threshold to 12'hFF0. I found that this process work and showered great promise. A sample output of an LED look like:



Now, that this initial processing was done we ended to know where the middle of this blob was in order to do some sort of analysis about the light we are tracking.
I decided to tackle this finding the average x and y coordinate. To do this I needed to know the number of pixels that were white. I had a variable that would do this. The variable was to be incremented at every clock cycle until we reached an hcount of 638 and account of 478. After this the variable that holds the the count for number of white pixels was reset to 0 so that this calculation could occur every frame.Meanwhile, towel other calculations were going to be done at this time, I had two more variables that would increment the total x and total y

for the pixels that were white. What this means is that if a pixel was white its hcount would be added to the total x and its vcount would be added to its total y. What this does is that at the end of a frame we would have 3 variable pieces of information, the total x value (the sum of every hcount for every white pixel in a frame), the total y value (the sum of every vcount for every white pixel within a frame), and the total number of white pixels. What this allows us to do is to now calculate the average x and average y of white pixels for each frame.

Now that we have variables explained in the last section we had to decide how to calculate the average x and average y. Two methods explored are:
Divide the total x and y by the total number of pixels. This would effectively need to be a 32 dividend being divided by a 19 bit divisor. I achieved this by using the divisor module in the IP section.
The second was to just multiple. Let me explain how this works. In the next module we would need to check if this average x or average y was within a hcount and vcount range if so the we would finally output the 3-bits needed to control the spectrum game. Was we can do is the follow with some math manipulation:
Ex. Instead of using this:

```
if ( average_x < 214) begin
        BLAH;
```

We can use:

```
if( total_x < 214*total_pixel)
```

Because average_ = total_x/total_pixel we can move the Total_pixel to the other side by multiplying thus having the same effect as dividing without having to deal with the dividing IP.

The last part we would do is then translate where the center of the blob is to a corresponding output. To better understanding it would be best to understand what the controls all mean.
We have 5 possible scenarios.

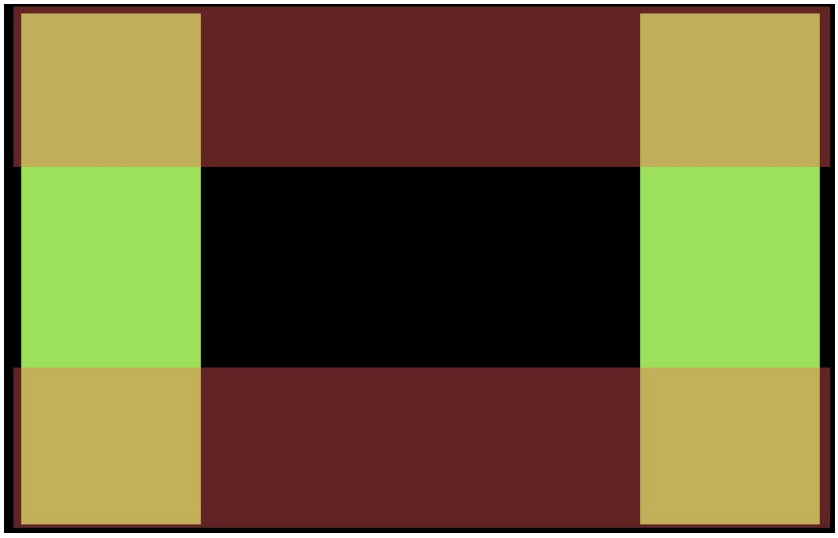
```
move_left
move_right
move_down
move_rotate
nothing
```

For each of the scenarios we then have a corresponding output:

```
 move_left = 3'b110
move_right = 3'b011
move_down = 3'b111
move_rotate = 3'b101
nothing= 3'b000
```

These would then be outputed to the general game control logic and would be used to control General gameplay.

This is how it is determined if what to output:



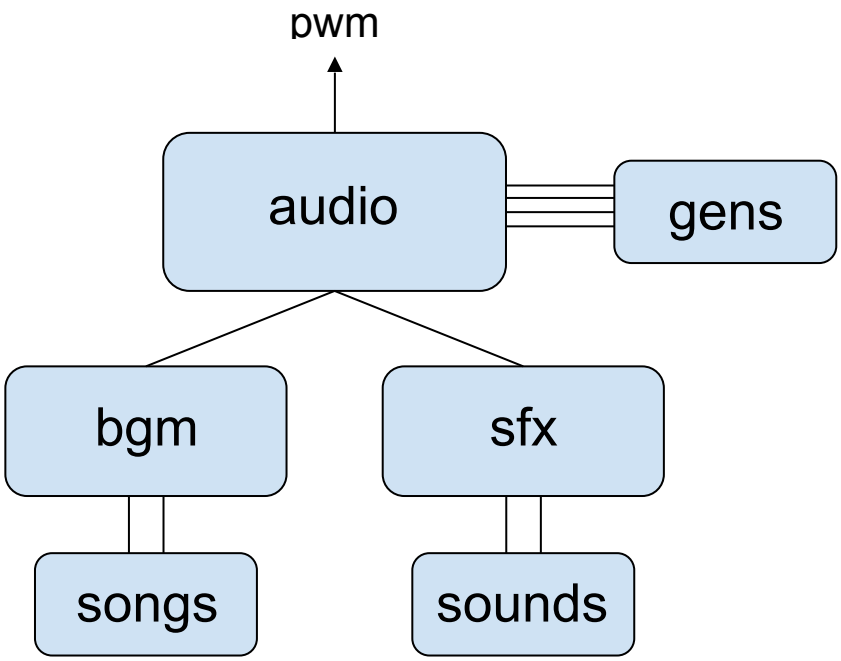
If the blob would be in the upper red box the output would be move_rotate
If the blob would be in the lower red box the output would be move_down
If the blob would be in the left green box the output would be move_left
If the blob would be in the right green box the output would be move_right
Else:
Nothing;
I got the general code working and tested individually and every individual module seemed to be working. When I put everything together there was a timing issue that would not allow the total_x, total_y and total_pixels to be calculated and because of this we have to revert to our backup and use switch as controls.

Audio(Jose)

The Spectris game has an audio system capable of playing 2 background music (BGM) tracks and 2 sound effects (SFX). In the interest of recreating retro game music, and also of saving memory, the audio is generated using instruction codes rather than played sample by sample.

The two BGM tracks are the classic Tetris theme and a mellower original theme. The two sound effects are a high-pitched sweep (used when blocks lock into place) and a shrill fanfare (used when a line is cleared).

Every 100Mhz clock cycle, the audio system counts up to a certain number to create its own slower clock. These slower audio cycles are useful because they reduce the number of cycles in any length of time; no sound effect or music track requires the excessive precision of 100 million actions per second.



The gens, which are the audio module's 4 square wave generators, are the only parts clocked by the fast 100mhz clock. These gens receive the number to count to (like a period) and a 4-bit volume, and return a 4-bit number representing the value of the wave - volume if the wave is up at the moment, or 0 otherwise. At all times, the audio module is summing up these generators into a larger-size number and encoding it into the pwm format used by the mono audio output of the Nexys 4.

To determine what volumes and frequencies to set the gens to, the audio module uses two similar modules, one for bgm and one for sfx. Every audio cycle these modules read a command from the selected song or sound effect. Each song or sound effect has its own module which simply returns a command depending on the value of the input address. They also indicate whether the address given is the last one for that sound, so that the parent module can loop the address (bgm) or stop properly (sfx).

The 45-bit instruction codes stored for these sounds work as follows: If the most significant bit is 1, the following 44 bits are instructions for the 4 square wave generators (for each, a 7-bit pitch number and 4-bit volume). If the MSB is 0, the following 44 bits determine how many audio cycles to wait until reading the next instruction. Because of this, stretches of time in which nothing changes can be represented with one instruction (or, if there are extremely long breaks, 1 instruction every 2^{44} audio cycles).

The bgm and sfx modules also contain a helper module that maps the 7-bit pitch value to the counter number used by the wave generators.

Since bgm and sfx both provide commands for the 4 wave generators, the audio module will use sfx commands first, if they are present at the current moment, and then otherwise apply commands from bgm if those are present. The sfx instructions have priority because sfx play for a short time only; the bgm will just resume afterward.

The disadvantages of this system include that it is relatively difficult to change songs. The sound effects were written manually, but the bgm songs were converted from MIDI files to

the Verilog modules using an external python script. The script extracts 4 MIDI tracks and copies the notes in them to produce a list of commands for all 4 tracks at once, which is the format in the Verilog song memories.

If the songs do not need to be tweaked, however, the system is efficient memory-wise. The largest possible song supported by the 10-bit addresses would have 1024 instructions, for a total size of 5.76 kB. By comparison, a comparable WAV file would be in the range of megabytes, or thousands of times larger.

The duration of the largest-memory song is effectively arbitrary since it depends on the amount of sleep instructions. Making the audio cycles slower would not reduce the amount of instructions, but would instead slow down the audio. Or put another way, a slower audio clock can play the same song using smaller values in its sleep instructions, but cannot eliminate any sleep instructions.

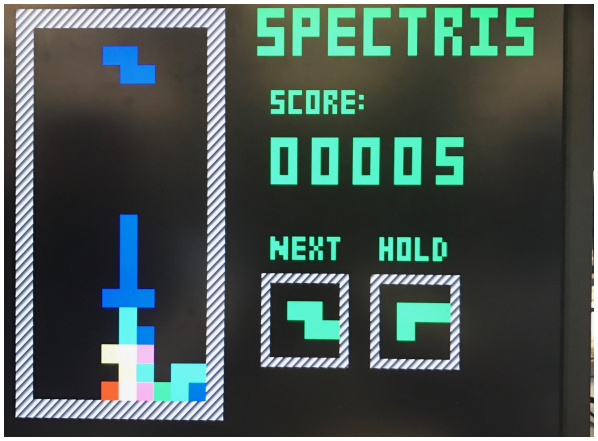
The game controls the audio module by inputting whether to play, pause, or stop the bgm, and which bgm to play. These inputs are the state, rather than flags to change state. The game also selects which sfx to play, but since sfx are short-lived there is no pause or stop control for them. Instead, the sfx plays as soon as it is selected and continues until it is finished; the 0 sound effect is reserved to play no sound. This is necessary for the game to play the same sound effect twice in a row, since it will need to switch the sfx to 0 before selecting the sound effect again.

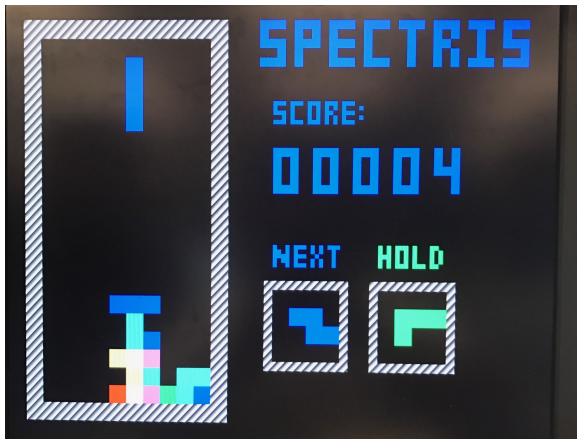
In addition to selecting the sounds to play and controlling the bgm, the game can flag “danger” to the audio system, which makes the audio cycles count on a shorter period—effectively increasing the tempo of all sounds. This is used when the player has stacked blocks very high and is in danger of losing.

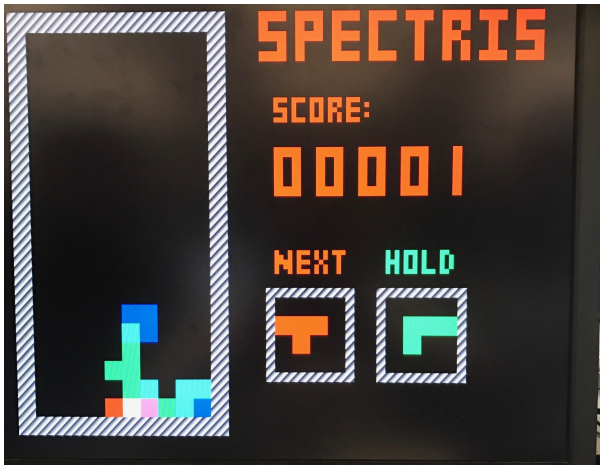
Some other limitations include that the 4 wave generators are not really independent; if only one note is supposed to change, the relevant instruction will nevertheless reassert the other 3 notes. Those notes can be reasserted as the exact same note as before, but it is still a waste of information. If the song memories were divided into one for each wave generator, the whole thing would take fewer bits overall since each wave generator would find moments to sleep while other notes were playing. The tradeoff, however, is a potentially more cluttered set of modules, which is why the song memories instruct all 4 waves at once.

Also, the tempo is handled at the level of the audio module, not the song instructions. Therefore, creating a song that changes tempo would require a workaround or overhaul in the python script, or a new subsystem that can dynamically and freely control the tempo. However, the current 2 songs are suitable at the fixed tempo (144 bpm) that is currently implemented.

Game play Images:







Conclusion

Overall the project was a success. We were able to implement a playable game with all its core functionalities. The most challenging part of the project was iterating to fix the game logic, which was too complicated to simulate. Our next steps for the project would be to add more game features, like menus and game mechanics, and get play testers' feedback on the game concept, though initial reception seemed very positive.

Top.v (camera code)

```
`timescale 1ns / 1ps
`default_nettype none

module top(
    input wire CLK_100M,
        input wire [15:0] SW,
        output wire [15:0] LED,
        output wire RGB1_Blue, RGB1_Green, RGB1_Red,
        output wire RGB2_Blue, RGB2_Green, RGB2_Red,
        //output wire [7:0] SSEG_CA, [7:0] SSEG_AN, //7 segment LED display
        input wire CPU_RESETN, BTNC, BTNU, BTNL, BTNR, BTND, //buttons
        inout wire [7:0] JA, JB, JC, JD, //PMOD headers
        //input wire [3:0] XA_N, XA_P, //analog inputs
        output wire [3:0] VGA_R, VGA_G, VGA_B, //VGA outputs
        output wire VGA_HS, VGA_VS

    );

    wire reset;
    assign reset = 0;

    wire video_clk;

    //camera signals
    wire camera_pwdn;
    wire camera_clk_in;
    wire camera_clk_out;
```

```
    wire [7:0] camera_dout;
    wire camera_scl, camera_sda;
    wire camera_vsync, camera_hsync;
    wire [15:0] camera_pixel;
    wire camera_pixel_valid;
    wire camera_reset;
    wire camera_frame_done;
    wire [11:0] hcount;
    wire [11:0] vcount;
    wire [31:0] tot_x;
    wire [31:0] tot_y;
    wire [18:0] tot_pixels;
    wire [2:0] command ;
    wire [55:0] x_average;
    wire [55:0] y_average;




    assign camera_clk_in = video_clk;
    assign camera_pwdn = 0;
    assign camera_reset = ~reset;

//assign camera outputs
   assign JA[0] = camera_pwdn;
   assign camera_dout[0] = JA[1];
   assign camera_dout[2] = JA[2];
   assign camera_dout[4] = JA[3];
   assign JA[4] = camera_reset;
   assign camera_dout[1] = JA[5];
   assign camera_dout[3] = JA[6];
   assign camera_dout[5] = JA[7];

   assign camera_dout[6] = JB[0];
   assign JB[1] = camera_clk_in;
   assign camera_hsync = JB[2];
   //assign JB[3]= camera_sda;
   assign camera_dout[7] = JB[4];
   assign camera_clk_out = JB[7];
   assign camera_vsync = JB[5];

   //assign JB[7] = camera_scl;

   wire [11:0] memory_read_data;
   wire [11:0] memory_write_data;
   wire [18:0] memory_read_addr;
   wire [18:0] memory_write_addr;
   wire memory_write_enable;

    //clock generation
     video_clk video_clk_1 (
       .clk_in1(CLK_100M),
       .clk_out1(video_clk)
       );
```

```
    //camera configuration module
    camera_configure camera_configure_1 (
        .clk(video_clk),
        .start(BTNU),
        .sioc(JB[6]),
        .siod(JB[3]),
        .done(LED[15])
        );

    //camera interface
    camera_read camera_read_1 (
        .p_clock(camera_clk_out),
        .vsync(camera_vsync),
        .href(camera_hsync),
        .p_data(camera_dout),
        .pixel_data(camera_pixel),
        .pixel_valid(camera_pixel_valid),
        .frame_done(camera_frame_done)
        );

    //write camera data to frame buffer
    camera_address_gen camera_address_gen_1 (
        .camera_clk(camera_clk_out),
        .camera_pixel_valid(camera_pixel_valid),
        .camera_frame_done(camera_frame_done),
        .capture_frame(1),
//        .capture_frame(BTNU),
        .camera_pixel(camera_pixel),
        .memory_data(memory_write_data),
        .memory_addr(memory_write_addr),
        .memory_we(memory_write_enable),
        .vcount(vcount),
        .hcount(hcount)
        );

    find_center find_center(
        .video_clk(camera_clk_out),
        .pixel_data(memory_write_data),
        .hcount(hcount),
        .vcount(vcount),
        .tot_x(tot_x),
        .tot_y(tot_y),
        .tot_pixels(tot_pixels)

    );

//    div_gen_1 x_ave(
//        .s_axis_divisor_tdata(tot_x),
//        .s_axis_dividend_tdata(tot_pixels),
//        .m_axis_dout_tdata(x_average)


//    );
```

```
//        div_gen_1 y_ave(
//        .s_axis_divisor_tdata(tot_y),
//        .s_axis_dividend_tdata(tot_pixels),
//        .m_axis_dout_tdata(y_average)


//    );



    send_input(
        .tot_x(tot_x),
        .tot_y(tot_y),
        .tot_pixels(tot_pixels),
        .video_clk(video_clk),
        .command(command)


    );

    led_test(
        .video_clk(video_clk),
        .command(command),
        .BTNU(BTNU),
        .LED(LED)

    );




    video_playback video_playback_1 (
        .pixel_data(memory_read_data),
        .video_clk(video_clk),
        .memory_addr(memory_read_addr),
        .vsync(VGA_VS),
        .hsync(VGA_HS),
        .video_out({VGA_R, VGA_G, VGA_B})
        );


    //frame buffer memory
    frame_buffer frame_buffer_1 (
        .clka(camera_clk_out),
        .wea(memory_write_enable),
        .addra(memory_write_addr),
        .dina(memory_write_data),
        .clkb(video_clk),
        .enb(1'b1),
        .addrb(memory_read_addr),
        .doutb(memory_read_data)
        );

endmodule
```

```
module camera_address_gen(
    input wire camera_clk,
    input wire camera_pixel_valid,
    input wire camera_frame_done,
    input wire capture_frame,
    input wire [15:0] camera_pixel,
    output reg [11:0] memory_data,
    output wire [18:0] memory_addr,
    output reg memory_we,
    output reg [11:0] vcount,
    output reg [11:0] hcount
    );


    parameter VCOUNT_MAX = 479;
    parameter HCOUNT_MAX = 639;
    parameter WHITE_VAL = 16'hFFF0;

//    reg [11:0] vcount = 0;
//    reg [11:0] hcount = 0;
    reg capture_frame_latched = 0;

    assign memory_addr = hcount + vcount * (HCOUNT_MAX+1);

    always@(posedge camera_clk) begin

        capture_frame_latched <= capture_frame ? 1 : camera_frame_done ? 0 :
capture_frame_latched;
        if(camera_frame_done) begin //set frame done
            vcount <= 0;
            hcount <= 0;
            memory_we <= 0;
        end

        else begin
            hcount <= camera_pixel_valid ? (hcount >= HCOUNT_MAX) ? 0 : hcount + 1 :
hcount;
            vcount <= camera_pixel_valid & (hcount >= HCOUNT_MAX) ? vcount + 1 : vcount;
            memory_we <= capture_frame_latched ? camera_pixel_valid : 0;

            if( camera_pixel > WHITE_VAL) begin
                memory_data <=12'hFFF;
            end
            else begin
                memory_data <=12'h000;


            end
          // memory_data <= {camera_pixel[15:12], camera_pixel[10:7], camera_pixel[4:1]}; //
convert camera RGB:565 to RGB:444
        end
    end
```

```
endmodule



module find_center (
    input wire [11:0] pixel_data,
    input wire video_clk,
    input wire [11:0] hcount,
    input wire [11:0] vcount,
    output reg [31:0] tot_x,
    output reg [31:0] tot_y,
    output reg [18:0] tot_pixels

);
    reg [31:0] x_total=0;
    reg [31:0] y_total=0;
    reg [18:0] total_pixels=0;

    always@( posedge video_clk) begin
//        if( hcount<2   && vcount <2 ) begin
//            total_pixels <=0;
//            x_total<=0;
//            y_total<=0;

//        end
        if( pixel_data> 12'hFF0  && hcount<638   && vcount <477 ) begin
            total_pixels <= total_pixels +1;
            x_total <= x_total + hcount;
            y_total <= y_total+ vcount;


        end

        if(hcount==639 && vcount ==478)begin

            tot_x <= x_total;
            tot_y <= y_total;
            tot_pixels <= total_pixels;

            x_total<=0;
            y_total<=0;
            total_pixels <=0;
        end

//        if( hcount>650 && vcount >490) begin
//            x_total<=0;
//            y_total<=0;
//            total_pixels <=0;



//        end

    end
```

```
endmodule


module send_input(
    input wire [31:0] tot_x,
    input wire [31:0] tot_y,
    input wire [18:0] tot_pixels,
    input wire video_clk,
    output reg[2:0] command



);


    always@ ( posedge video_clk) begin
        if(tot_y< 162*tot_pixels) begin
            command <= 3'b101;

        end


        if(tot_y > 324*tot_pixels) begin
            command <= 3'b111;

        end


        if(tot_x < 214*tot_pixels) begin
            command <= 3'b110;


        end

        if(tot_x > 428*tot_pixels) begin
            command <= 3'b011;


        end

        else begin
            command <= 3'b000;


        end




//        if(tot_y[55:24]< 162) begin
//            command <= 3'b101;

//        end
```

```
//        if(tot_y[55:24] > 324) begin
//            command <= 3'b111;

//        end


//        if(tot_x[55:24] < 214) begin
//            command <= 3'b110;


//        end

//        if(tot_x[55:24] > 428) begin
//            command <= 3'b011;


//        end

//        else begin
//            command <= 3'b000;


//        end



    end




endmodule

module led_test(
    input wire video_clk,
    input wire BTNU,
    input wire [2:0] command,
    output wire [15:0]  LED


);

        assign LED[0] = command[0];
        assign LED[1] = command[1];
        assign LED[2] = command[2];



//        always@(posedge video_clk) begin

//            assign LED[0]
//            if(command[0]) begin
//                assign LED[0] = command[0];
```

```
//          end

//          else begin

//          end


//       end


endmodule


module video_playback(
    input wire [11:0] pixel_data,
    input wire video_clk,
    output wire [18:0] memory_addr,
    output reg vsync,
    output reg hsync,
    output wire [11:0] video_out
    );


    // horizontal: 800 pixels total
    // display 640 pixels per line
    reg hblank,vblank;
    wire hsyncon,hsyncoff,hreset,hblankon;
    reg [11:0] hcount = 0;
    reg [11:0] vcount = 0;
    reg blank;
    //kludges to fix frame alignment due to memory access time
    reg blank_delay;
    reg blank_delay_2;
    reg hsync_pre_delay;
    reg hsync_pre_delay_2;
    reg vsync_pre_delay;
    reg vsync_pre_delay_2;

    assign video_out = blank_delay_2 ? 12'b0 : pixel_data;

    assign hblankon = (hcount == 639);   //blank after display width
    assign hsyncon = (hcount == 655);  // active video + front porch
    assign hsyncoff = (hcount == 751); //active video + front portch + sync
    assign hreset = (hcount == 799); //plus back porch

    // vertical:  525 lines total
    // display 480 lines
    wire vsyncon,vsyncoff,vreset,vblankon;
    assign vblankon = hreset & (vcount == 479);
    assign vsyncon = hreset & (vcount == 489);
```

```
    assign vsyncoff = hreset & (vcount == 491);
    assign vreset = (hreset & (vcount == 524));

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

    assign memory_addr = hcount+vcount*640;

    always @(posedge video_clk) begin
        blank_delay <= blank;
        blank_delay_2 <= blank_delay;
        hsync_pre_delay_2 <= hsync_pre_delay;
        vsync_pre_delay_2 <= vsync_pre_delay;
        vsync <= vsync_pre_delay_2;
        hsync <= hsync_pre_delay_2;
         //hcount
         hcount <= hreset ? 0 : hcount + 1;
         hblank <= next_hblank;
         hsync_pre_delay <= hsyncon ? 0 : hsyncoff ? 1 : hsync_pre_delay;  // active low

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

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