6.111 Project Report

Mario Bros Classic in Real Life

Jose Guajardo, Nancy Hidalgo, Isabelle Chong

System Overview and Block Diagram:

For our 6.111 final project, we recreated the side-scrolling Mario Bros Classic game for Nintendo with several added functionalities. We programmed this project on the Nexys4-DDR FPGA using the XVGA display to host the game graphics. Throughout the project, we made use of both internal and external memory in our game functionality, using the Nexys4-DDR's internal BRAM to store game graphics and the overall map of our level, and an SD card to store audio for the Super Mario Bros theme. We also programmed the game so it could be controlled using both the built-in buttons on the FPGA and a Teensy-interfaced IMU.

Overview of Sprite Generation Approach and Terminology - Jose

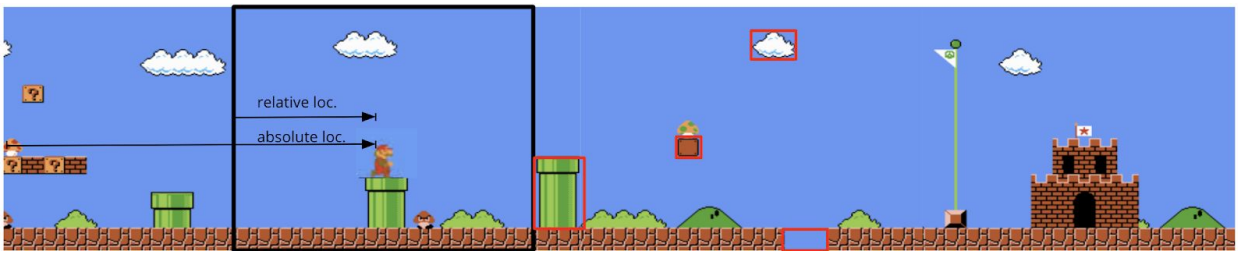

Figure 1: Early diagram of overall level layout

The level that we created for our project was based loosely on the level that can be seen above. Since the level is wider than a single camera width, one of the key design aspects and challenges for our project was implementing side scrolling. In order to implement side scrolling, we kept track of several attributes of each of the objects in the level. Firstly, we saved the absolute positions of each object. The absolute positions (both x and y) are referenced from the very beginning of the level, x = 0. In addition, the game mechanics module keeps track of the location of the left edge of the camera view. By using the position of the camera's left edge, we can compute a relative position for each object, which is measured from the camera's left edge. An object's relative position can be calculated simply by subtracting the camera position from the object's absolute position. The objects in the game that side scroll are those whose relative positions changes as the game progresses. In our implementation, Mario is not considered a side scrolling object. Instead, he remains in the same relative position while the objects in the level scroll past him. Relative positions were only implemented in the x direction, since our side scrolling only operates horizontally. However, a similar approach could have been implemented to smoothly integrate side-scrolling in two dimensions.

Sprite Pipeline - Jose

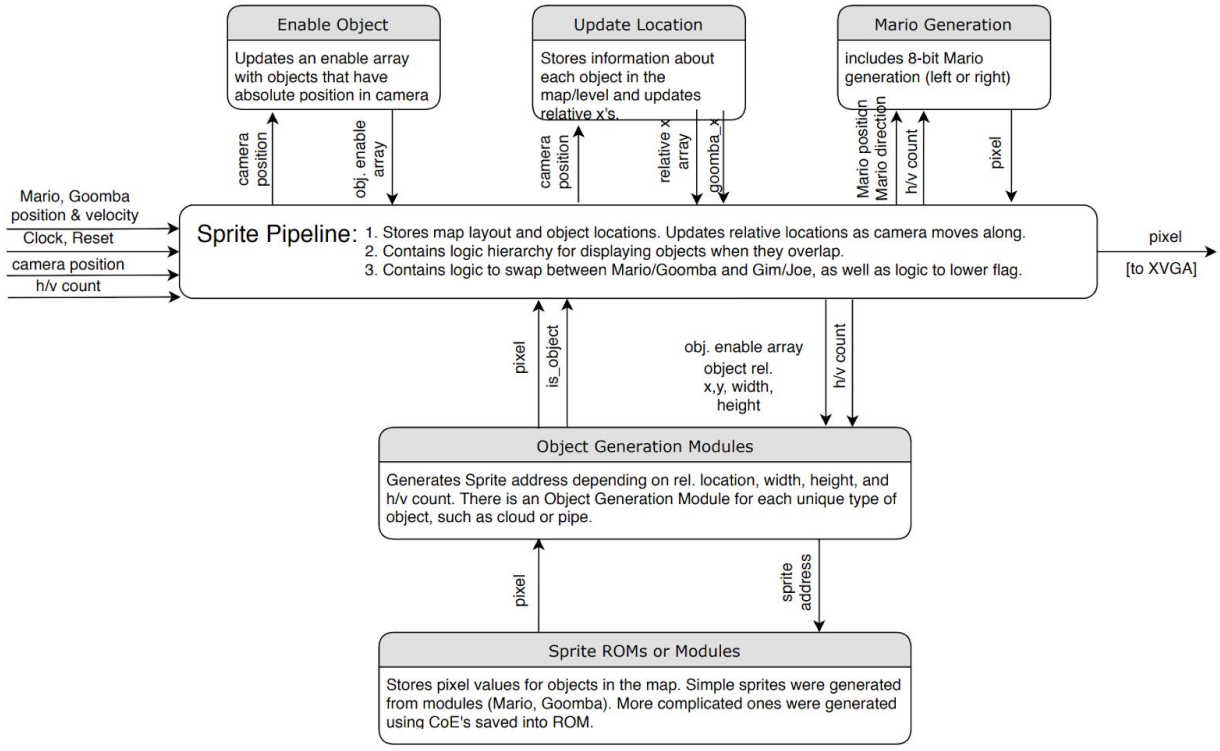

Figure 2: Sprite Pipeline block diagram

The Sprite Pipeline one of the three major modules of our project. The Sprite Pipeline is implemented in our Verilog code as an instance of sprite_wrapper.sv, which contains an overarching sprite module that integrates sub-modules of the Sprite Pipeline. The Sprite Pipeline is responsible for displaying all of the characters and objects that are a part of the game level. Internally, it keeps track of object positions as the location of the camera as it moves with Mario, implementing side-scrolling. Lastly, it communicates an output pixel value to top level, which outputs it to XVGA. One of the key properties of our sprite generation system design is that all sprites are generated in parallel. As part of developing the Sprite Pipeline, I designed the entirety of our level, including locations of colliding objects, such as pipes, platforms and the flag, as well as cosmetic objects like the bushes, clouds, the castle and fireworks.

The Sprite Pipeline creates instances of the sprites modules that are described in Sprite Generation. One instance is created for each object in the level. Each of the instances outputs a flag, such as “is_mario,” which is high when an object pixel exists at that h/v count location. Those flags are used at the end of the Sprite Pipeline to create a logic hierarchy for displaying objects when they would overlap. The Sprite Pipeline contains logic to prevent objects from being rendered outside of the gameboy UI’s screen. Additionally, it contains logic to switch between the Mario and Goomba Sprites and the Gim and Joe sprites. Lastly, the logic that

lowers the flag after the game is won is within the Sprite Pipeline, and updates the position of the flag over a period of time.

The Sprite Pipeline integrates several main modules:

- Enable Object
- Update Location
- Mario Generation
- Sprite Generation

Enable Object Module - Jose

The Enable Object module communicates to the Sprite Pipeline an array of enable flags, each corresponding to an object in the game level. The module combinationally determines if each object exists entirely between the edges of the camera view by taking into account the object's x position and width. However, this would mean that if an object overlapped with either camera edge, it would not be rendered at all. As a result, we added a dead zone with size equal to the width of the widest object that side scrolls, in this case the castle.
Block Diagram:

Sample Implementation for Goomba's enable flag:

```
goomba_enable = ((left_edge - dead_zone_width) <= goomba_x && (goomba_width
+goomba_x) <= (right_edge + dead_zone_width));
```

Update Location/Side Scrolling Module - Jose

The Update Location module is responsible for updating the relative location of each object that is displayed and communicating those locations to the Sprite Pipeline. If an object's flag is enabled by the Enable Object module, the Sprite Pipeline needs to know the location of the object relative to the camera's left edge in order to render it in the correct location. As the game mechanics module sends an updated camera position location, each object on the screen scrolls in the direction opposite to Mario's movement. That is, if the player chooses to move right, all objects scroll left until they are no longer on the screen and their enable flags are low. All objects in the game have their locations updated by the Update Location module, except for Mario, whose relative x location never changes as the game scrolls.
Block Diagram

Sample Implementation:

```
rel_goomba_x <= goomba_x - camera_position;
```

Mario Generation - Jose

The Mario Generation module is responsible for generating one of two mario sprites - the left or right facing Mario sprite. Depending on Mario's direction, which is determined by the game mechanics modules, Mario's sprite is updated. Mario's sprites are saved in independent modules, rather than in ROM, in order to save memory space on the FPGA.

Sprite Generation Modules - Izzy

Mario Bros Classic is a game with many graphical components, so memory was a concern for this portion of the project. In an effort to conserve memory, some game sprites such as Mario, Goomba, and brick blocks were generated as modules, using a case statement in order to individually color select pixels within a range of the sprite's height and width.

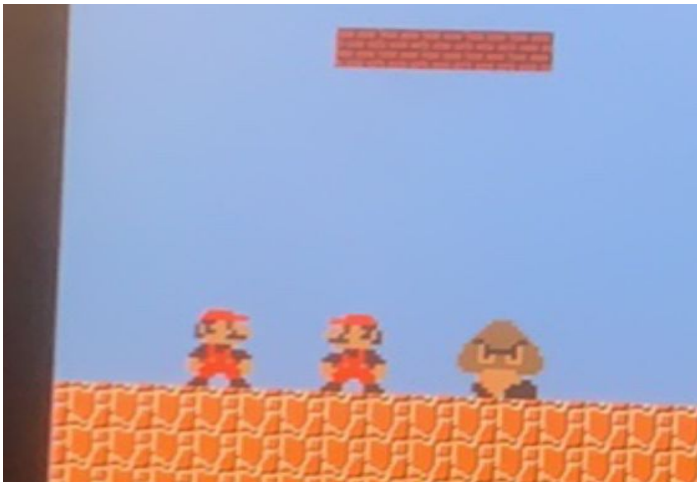

Figure 3: Two Marios, a Goomba, and some bricks, as generated by System Verilog Modules

However, the majority of the sprites were more complicated and thus needed to be generated as COEs. To do this, we used the MATLAB script provided to us in Lab 3 (Pong Game). In the case of black and white sprites (game over, score numbers, and interface logo), sprites could be generated using only an image ROM. However, because most sprites needed four ROMs (image, red, green, and blue) to be displayed properly, sprite storage was a concern. We attempted to mitigate this problem through a series of tricks. The first was that, since Super Mario Bros art is very pixelated and the XVGA screen is only 1024x768 pixels, we were able to scale down our sprite images in the PNG application using the "adjust size" functionality to much smaller sizes without any appreciable loss in quality of image. We would then convert these PNG files to BMPs using an online converter (https://online-converting.com/image/convert2bmp/) with settings on 4 bit indexed color and row direction from top to bottom. We could then pass the BMP file through the MATLAB script to generate COE files that would require less depth (width x height) in our ROMs and therefore less BRAM. The second trick was scaling up images. Using the line of System Verilog below:

```
assign image_addr = (hcount_in/4-x_in) + (vcount_in/4-y_in) * ADDR_WIDTH;
```

By dividing hcount_in and vcount_in by factors of four, we could get the XVGA to output the same pixel 4 times in a row, therefore scaling up the image and increasing its size. This particular method was used to generate the Gameboy background interface of our project. We can also use logic to extend images so we can generate different sized sprites using a single set of ROMs. This was used for the pipe sprite using the following line of System Verilog

```
assign image_addr = (vcount_in - y_in < ADDR_HEIGHT)? ((hcount_in-x_in + 3) +
(vcount_in-y_in) * ADDR_WIDTH) : ((hcount_in-x_in + 3) + (ADDR_HEIGHT - 10) *
ADDR_WIDTH);
```

This Verilog code tells the image_addr to repeatedly read one of the bottom lines of the image in the case that we want to make a pipe greater than the height of the pipe COE (50 pixels).

In order to further save memory needed for map generation, we attempted to make a “gameboy” background module that would allow us to decrease playable “screen” size and therefore decrease map size overall. This pixel gameboy image was displayed in the background of our game at all times to enhance overall user experience.

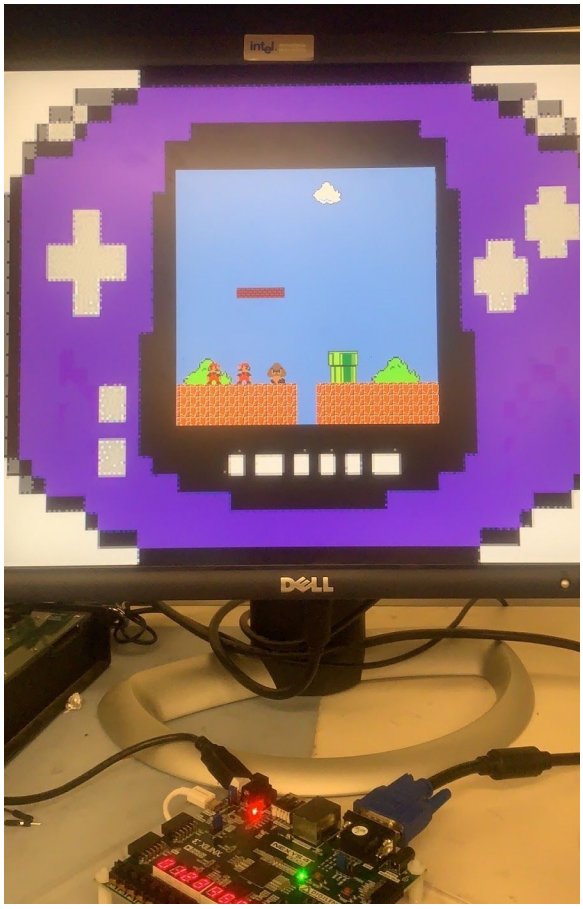

Figure 4: Display of stationary Mario sprites on Gameboy background

A different problem encountered with the generation of sprites was the problem of aesthetics. Because COEs are generated from images and images are rectangular, it was difficult to separate irregularly shaped objects, such as clouds and bushes, from the rectangular backgrounds they were on. This was solved by first erasing the backgrounds of the images (either by searching for images on Google that themselves had transparent backgrounds, or by cropping using the magic lasso tool in PNG and then cleaning up excess background with the irregular lasso by hand. Note that the cropping method can probably be more easily done in Adobe Photoshop, if one has the money for licensing fees). The transparent background images

would then be placed onto pure-colored backgrounds, either green or red (if the image contained green or green-ish tones).

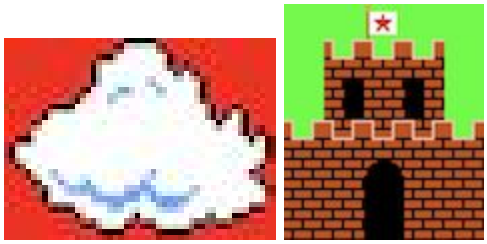

Figure 5: A cloud and castle on respective red and green backgrounds

In the COE generating module, these backgrounds could be filtered out using the following code structure

```
if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) && (vcount_in >= y_in && vcount_in
< (y_in+HEIGHT))) begin
	if (!(green_mapped[7:4] > 4'h8 && red_mapped[7:4] < 4'ha && blue_mapped[7:4] <
	4'ha)) begin // checks if pixel value for green is high, but values for the
	other two colors //are low -> pixel is probably a shade of green
		is_castle <= 1; // raises flag that pixel is part of the sprite object
		pixel_out <= {red_mapped [7:4], green_mapped[7:4], blue_mapped[7:4]};
	End
	else begin
		pixel_out <= 0;
		is_castle <= 0;
	end
end
else begin
	pixel_out <= 0;
	is_castle <= 0;
end
```

Using the variable is_castle for this example, in this case part of the sprite generation for the castle sprite, allows us to flag the green background portion of the image as “not castle,” thus allowing us to not display that portion of the image by setting pixel_out to 0. The is_castle variable also has a use in the top level. When trying to display multiple sprites on a single screen, one way to make the images appear is to use the bitwise “or” operator. However, in the case that we have multiple sprites layered on top of each other (in our case the gameboy “background” and square “screen,” as well as the additional obstacles) using a bitwise “or” causes images to have a translucent effect, which looks unappealing. Therefore, rather than using the “or,” we mux the different inputs using a statement like such:

```
pixel_out <= (you_died)? ((is_game_over)? game_over_screen_out: (is_buttons)?
button_out: (is_logo)? logo_out: gameboy_out):
	(is_timer)? Timer_out:
	(is_lives)? Lives_out:
	(is_object)? Object_pixel:
```

```
(is_floor)? floor_out:
(is_screen)? screen_out:
(is_buttons)? button_out:
(is_logo)? logo_out: gameboy_out;
```

This allows images to remain opaque when overlaid with each other.

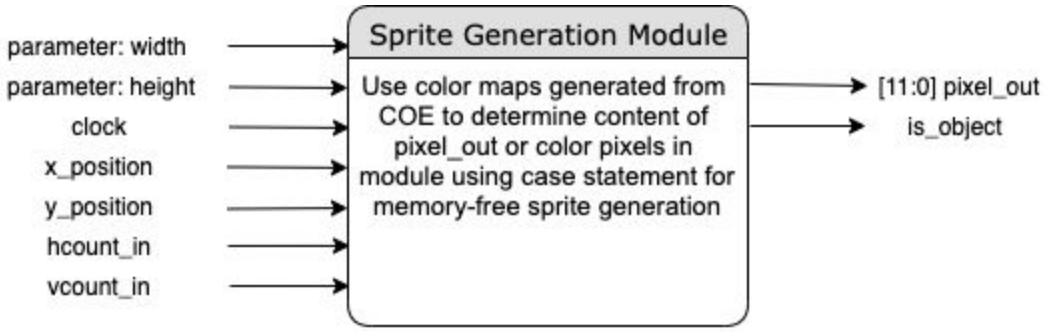

Figure 6: Block diagram for general sprite generation module

Additional support in the graphics generation modules was needed for sprites such as the gameboy buttons, flagpole, timer, and live hearts. For instance, we placed black square and cross sprites on top of the button pixels in the gameboy background, and activated the sprites in response to the left, right and up signals given by the FPGA buttons (or alternatively, IMU) in order to show sprite motion within the background interface. For the flagpole sprite, we created a module that combined the smaller flag and pole sprites into one callable unit, adding a module input that allowed us to change the position of the flag on the pole which we later leveraged with the game mechanics module to create a lowering flag animation once the game was won. For the timer sprite, we combined three instantiations of single number sprites into a three digit number sprite module that took in a decimal number typed as hex, then indexed through its bits to find hundreds, tens, and ones inputs for each number sprite which could then be displayed on the screen in three digits. Finally for the live hearts sprite, we combined three heart sprites in a module that accepted an input of the number of lives left in order to toggle which pixels to display. We added animations to these in order to make the overall game experience more immersive, and to allow players to see their progress and receive greater feedback on their actions as they played through the game.

Game Mechanics - Nancy

The game mechanics module was the module that controlled overarching game behaviors, such as pausing the game, resetting the level when Mario died, and keeping track of Mario's lives and the time left to complete the level. This module consisted of an FSM with 6 states:

1. Start game: State where the timer resets to 51, Mario starts at the beginning of the level
2. Game Play: State where Mario's position updates based on the user inputs, collision detection, and the timer
3. Two Player Game Play where Goomba and Mario's position are updated based on inputs and collision detection

4. Pause Game: State where Mario's and Goomba's positions are updated, timer stops counting down
5. You Won: State where either Mario has reached the end of the level in single-player mode or Mario has gotten past Goomba in two-player mode
6. Game Over: State where Mario has run out of lives and Game Over screen is displayed.

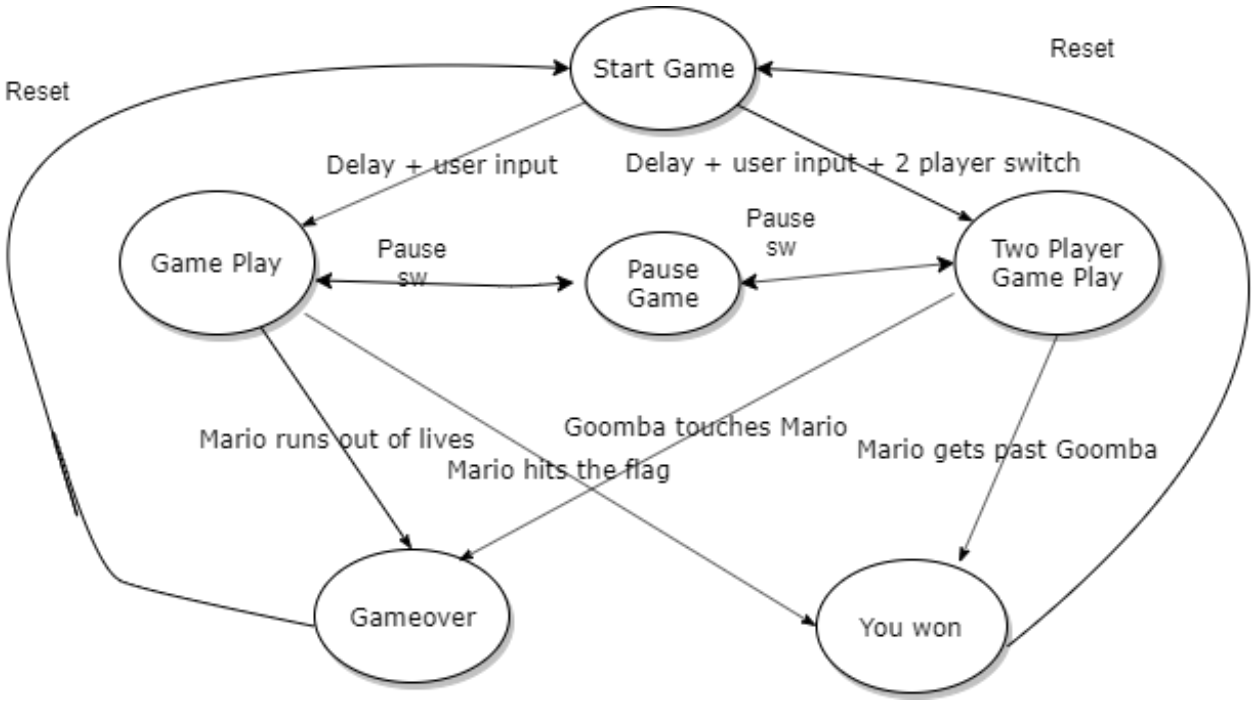

Fig. 7: Simplified FSM diagram of the game

Mario's lives would only be decremented when the timer ran out or when Mario collided with the bottom of the hole or when Mario collided with Goomba. The timer would work similar to how the counter worked in the pong lab, just decrementing rather than incrementing. Each bit of the timer would be updated when necessary.

This module connected the movement and collision detection modules to each other and to the rest of the system (sprite generation, IMU controller, etc). This was the first module I wrote and to debug it I wrote a test bench for it. Once everything had been integrated, there were a few kinks, namely how the game would reset. While the FSM was transitioning properly, Mario's position wasn't being reset. This was due to the different clock domains the movement module and the game FSM were running on. The movement module had to run on vsync, the game fsm was transitioning out of the start game state too quickly. To fix this, I added a delay to the start game state.

Movement - Nancy

The movement module updated Mario and the Goomba's positions, as well as the position of the left edge of the window and the direction of the 'camera' based on user inputs, the current state of the game, and the collision detection.

If at any point the game restarts (either when someone hits the ‘reset’ button or when Mario dies), Mario and Goomba’s positions are reset to starting locations. If the game is in the paused state, the you_won state, or in the gameover state, Mario/Goomba’s positions are not updated. The magic happens in the Game Play and Two Player Game state.

When the Game FSM is in the Game Play or Two Player Game Play state, Mario has 9 possible directions of movement: no movement, up, down, left, right, and the 4 combinations of up/down with left/right. In each state, if Mario isn’t restricted by a collision his absolute x and y positions are incremented/decremented by adding/subtracting the speed input to his current position, as well as the direction of the camera and the absolute location of the left edge of the camera. This information is fed into the aforementioned sprite pipeline which takes these absolute positions within the level and turns them into relative locations rendered on the XVGA. In this moduleFrom the IDLE state, Mario transitions to these different directions based on user inputs, then based on the type of collision, Mario’s position is updated. For example, if the user sends a jump signal and Mario is below a brick platform, Mario transitions to the down state. Or if Mario is on top of a pipe and goes to the right of the pipe, Mario’s direction becomes down and to the right. A simplified diagram of the finite state machine for Mario’s direction is depicted below:

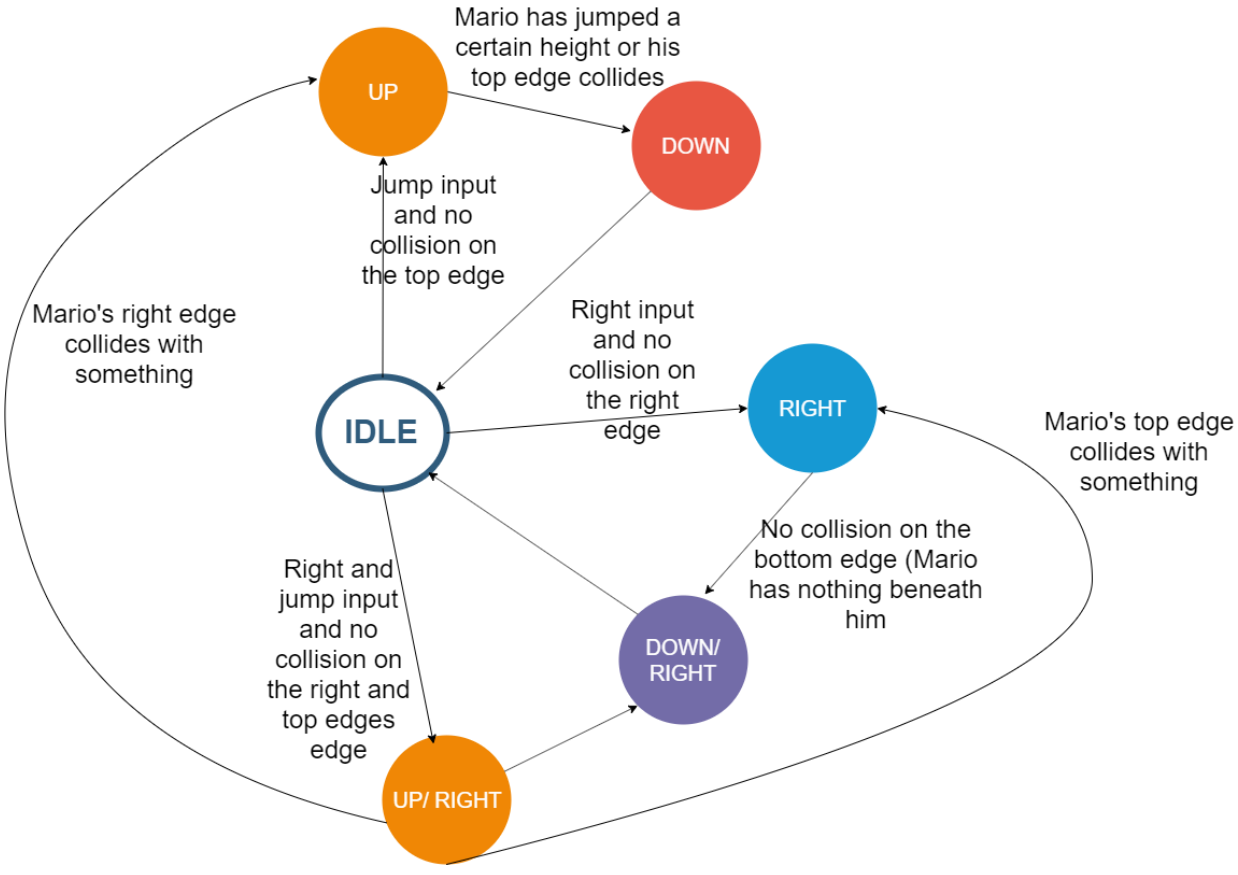

Fig 8: A simplified diagram of the FSM controlling Mario’s direction of movement

Mario's behavior is the same in the one and two player versions, however goomba's varies between the two. In the one-player version Goomba either moves left or right based on his direction. Once Goomba hits a pipe, the collision detection module reverses Goomba's direction, making the Goomba move the other way. Meanwhile in the two-player version, Goomba's movement depends on user inputs and on his collisions with the pipes. Users can move Goomba left until he runs into the leftmost pipe, and right until he hits the pipe to his right.

**Note: once we started integrating the modules, I realized that there was a simpler way to implement Mario's movement. Rather than have separate states for the combinations of left/right with up/down, I could have decoupled the horizontal and vertical directions, making the left and right movement solely dependent on inputs and the collisions, then making the vertical movement a state machine with only 3 states (UP, DOWN, and IDLE). This also would've made it simpler to implement parabolic movement, where the vertical speed was constantly decreasing over time.

This simpler implementation was written and can be found at the bottom of the game_mechanics.sv file, but unfortunately I prioritized other features and was unable to finish debugging it in time.

Collision Detection- Nancy

To implement collision detection in our game, we created two different collision detection modules: a collision detection helper that took Mario's position and the type of obstacle and output a collision type. Then there was a collision detection module that would determine which two obstacles were closest to Mario and send those two obstacles to two instances of the helper module. Then based on the outputs of those two instances, this module determines what collision type to send to the movement and game mechanics modules. This module also determines if there was a collision with a star coin and determines if the Goomba has collided with a pipe.

This was the trickiest module to debug. I couldn't test this module very well until it was integrated with everything else. If I were to do it again, I would've written a test bench that tested this module in combination with other modules.

Helper Module: Given the obstacle (it's absolute position in the level, it's dimensions, and the type of obstacle), this module determines if there's a collision with that object and, if so, what kind of collision. In our level, we had three categories of obstacles: a flag, a hole, and bricks/pipe that generally would restrict Mario's movement. Mario had to interact different with each category, falling into the hole, hitting the flag, and running into the restricting obstacles.

There are 11 types of collisions: collisions that restrict Mario's movement on one of his 4 edges (top, bottom, left, right) , the right angle combinations (bottom & right edges, top and left edges, etc), collision with the hole, collision with a Goomba, and no collision. Collision type is a four-bit output, with each bit representing one of Mario's four edges. This enabled simpler logic for the right-angle combinations. The hole collision and Goomba collisions were arbitrary numbers that would correspond to edge combinations that wouldn't occur in this level (i.e. 1110, 1111).

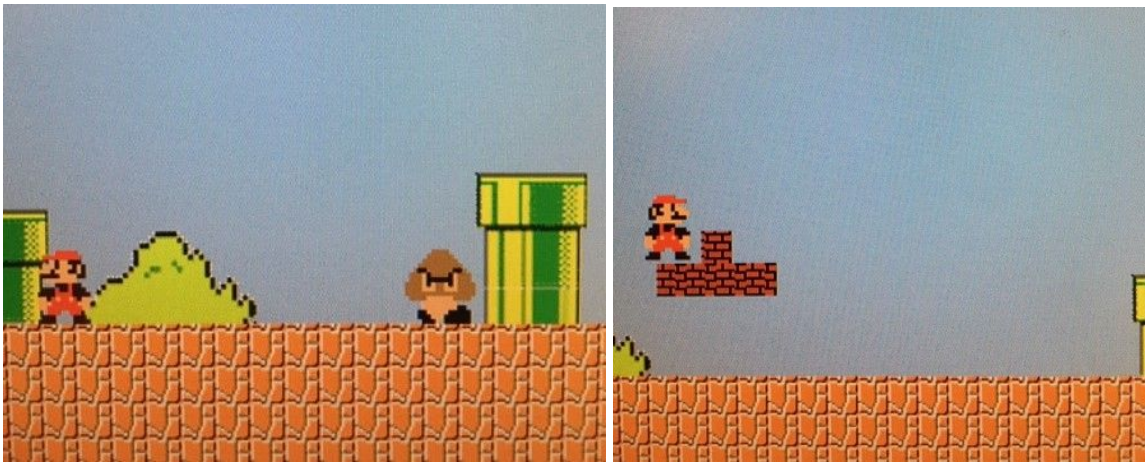

Fig 9: Images depicting different types of collisions (Left & bottom restrictions and right and bottom restrictions). Note that in the picture on the right, the brick obstacle actually consists of two separately rendered and separately generated objects, one longer platform with a smaller one on top. For this reason, we had two instances of the collision helper module instantiated, so that we could detect collisions with two different objects at one time. This then required additional logic to determine what collision type to send to the movement module. Simply put, if one instance detected an obstacle and the other didn't, that collision type would be sent, if they both detected different ones, they would be combined, or if one detected collision was more restrictive, that one would be sent to the movement module.

IMU Controller - Izzy

To make the game more engaging and add an extra level of difficulty to our project, we incorporated an IMU controller as an alternative way to move Mario. We interfaced the IMU to the project through the Teensy microcontroller the same way that was done in Lab 5B (FPGA IMU controlled level), using an IIR filter to make readings more steady and outputting the IMU readings on the seven segment display of the Nexys4.

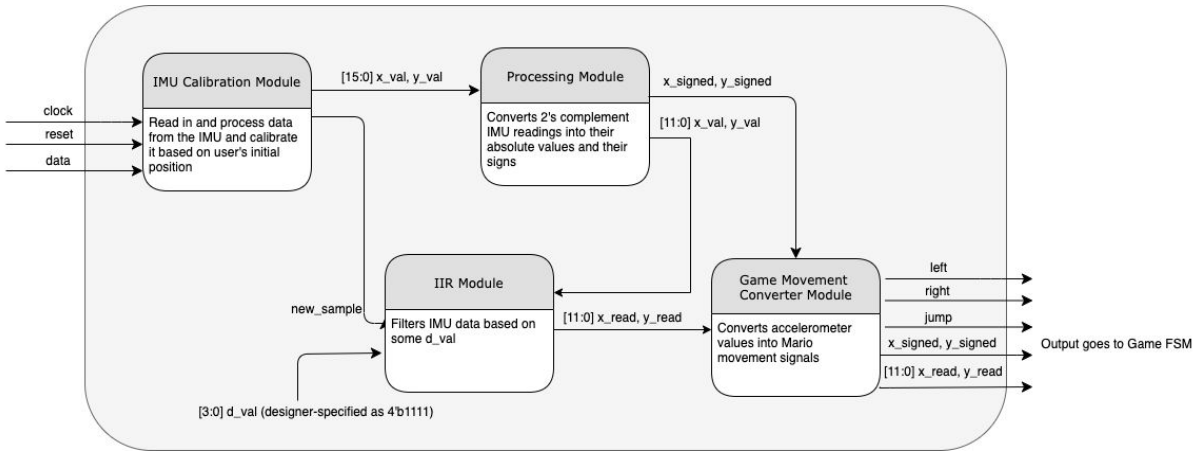

Figure 10: Block diagram of IMU controller and IIR filter

We automatically set the d_val of the IIR as 4'b1111 in order to make the motion as smooth as possible. However, because human hands still tend to be quite jittery, further filtering was needed. Therefore, we only considered signals above 12'hFF in determining whether or not Mario would move left or right. We also used the IMU output readings as a way to control Mario's speed by taking the top 4 bits of the x_read and y_read, which were typically values from 1 to 4. That way, we could have the number of pixels Mario moved per cycle vary with how much the player tilted the IMU. Because the IMU's positive and negative directions may be a bit unintuitive compared to the moves that a player would naturally tilt the controller to move Mario left or right, we tied "inverted controls" to switch 12 on the FPGA.

SD Card Audio - Izzy

The Super Mario Bros Classic Overworld Theme is a classic part of the game. Therefore, we wanted to be able to play the theme music while our game play was in motion. Because we were worried about using up too much storage space in the BRAM of the FPGA, we decided to store the data on an SD card and have the FPGA read that out of the audio port for sound data. The steps described here are also detailed in this tutorial (by Grace Quaratiello https://d1b10bmlvqabco.cloudfront.net/attach/jwaxfxqhaz64tj/j7ah50uwh5r3cp/k3q026gvbn3l/SD_Card_Audio__6.111.pdf). First, we downloaded the Overworld Theme online as an mp3, which we converted into a WAV file sampling at 32 kHz with 8 bit unsigned PCM using Audacity. We them wrote this WAV file to an SD card using HXD (only available on PC). Here, we made sure to recalculate the sampling rate, as detailed by Grace's tutorial, by dividing the number of samples by the length of the audio, where we found that the actual sampling rate of the audio (possibly because Audacity had converted a 16 bit audio at 32 kHz to an 8 bit audio) was 64 kHz. The SD Card reads out at a speed of 25 MHz, which is too fast for humans to hear. Thus, we had to pass the audio through a FIFO (of width 8 and depth 2048, which can be made using the IP Catalog) in order to get it to output at the correct 64 kHz rate. We used the sd_controller module provided by the 6.111 course staff to get output from the SD card to pass into the FIFO. A block diagram of the process with simplified inputs and outputs is shown below.

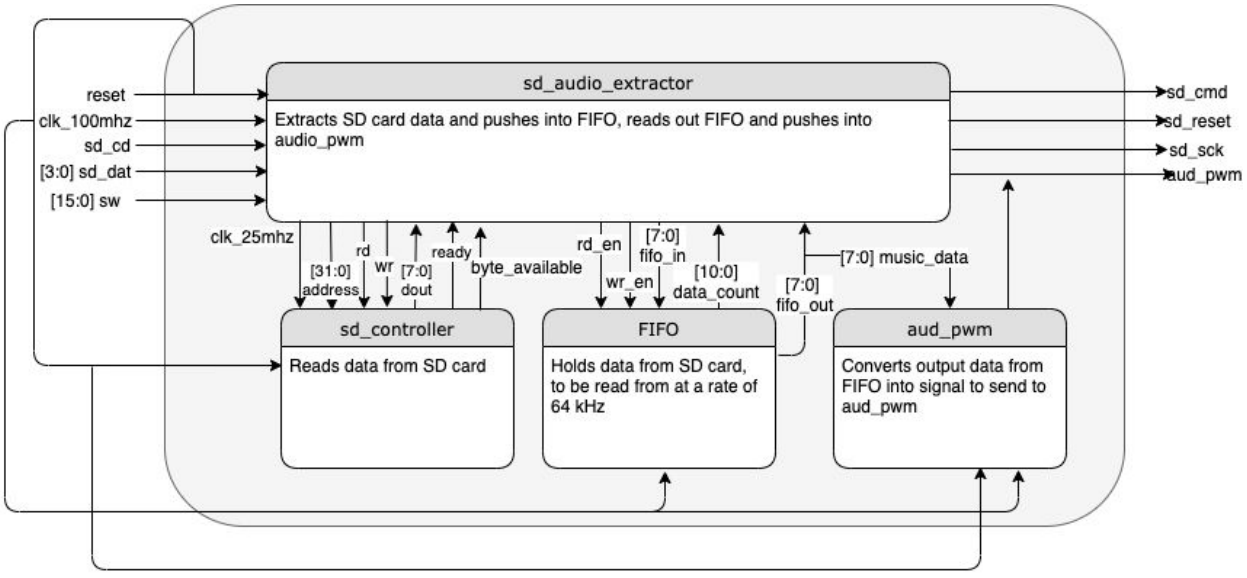

Figure 11: A simplified block diagram of the SD audio module and its inputs and outputs

Within the sd_audio_extractor module, we want to read data when we set the rd wire to high and the sd_controller has output a ready signal. The SD card is divided into sectors of 512 bytes. Therefore, after reading through 512 bytes at a given address we have to increment that address by 512 to read the next sector. We want to write from the sd_controller module to the FIFO when we have a new byte available; therefore, we will set wr_en on the FIFO to 1 on the rising edge of byte_available from the sd_controller and 0 otherwise. However, the FIFO has a limited amount of space, and if we let it fill up, we will lose audio data from the SD card because we will not be able to write to it until it clears out. Thus, when data_count in the FIFO is greater than 1024, we will change the rd signal to the sd_controller to 0 so we will stop reading data into the FIFO until we have enough space for it. We want to read data from the FIFO at 64 kHz and pass this data to the aud_pwm module. Therefore, we will set up a counter structure so that we set rd_en on the FIFO to 1 every 10,000,000/64,000 or approximately 1562 cycles, and make it 0 otherwise. A basic diagram of this state machine is shown below

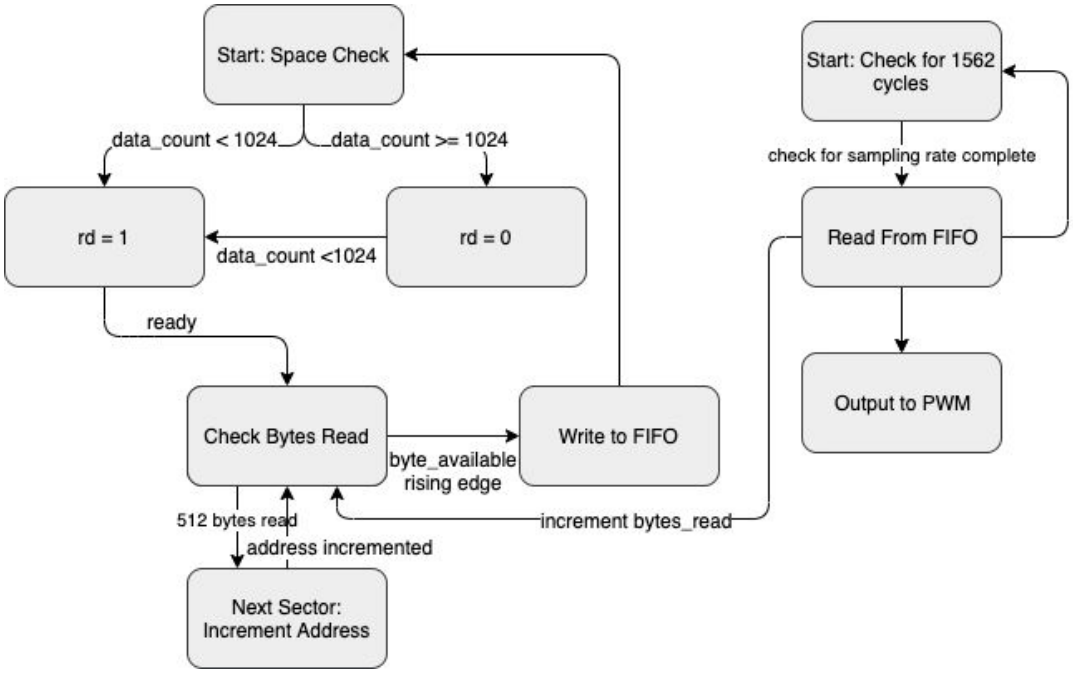

Figure 12: A basic state machine of the process of reading audio from the SD card and FIFO system

The aud_sd needs to be pulled high to amplify the sound, using something like this code

```
assign aud_pwm = (aud_pwm)? 1'bZ: 1'b0; // where aud_pwm is an inout
assign aud_sd = 1;
```

In order to get louder amplitude audio, we also used a triangle wave rather than a sawtooth wave, which is in the original audio_pwm module on the 6.111 website. Here is the audio_pwm that we used (as provided by Gim):

```
module audio_PWM(
       input wire clk,            // 100MHz clock.
       input wire reset,          // Reset assertion.
```

```
        input wire [7:0] music_data,      // 8-bit music sample
        output reg PWM_out          // PWM output. Connect this to ampPWM.
        );
        logic [7:0] count, ramp;
        logic flip;
        assign PWM_out = ramp < music_data;
        always_ff @(posedge clk) begin
                if (reset) begin
                        count <= 8'b0;
                        flip <=0;
                end else begin
                        count <= count+8'b1;
                        flip <= (count == 255) ? !flip : flip;
                end
        end
        // this creates a symmetrical ramp for better audio output.
        assign ramp = flip ? count : 255-count;
endmodule
```

SD card audio was quite difficult to debug, so a good process to go through when debugging SD audio would be to

1) Input a sine wave on the SD card and try to play that, checking with a tone generator to see if output is accurate
2) If another group in the lab happens to also be doing SD card audio, try swapping cards with them and seeing if their audio works with your code
3) Try generating different songs on your SD card and seeing if those work, as sometimes the problem may be with the mp3 or the SD card itself (a problem we specifically had, our mp3 when converted to WAV and placed on HXD tended to clip high notes)

Concluding Thoughts

Overall, the completion of this project was a fulfilling exercise. Our group worked well in parallel, since we scheduled our task completion such that each person could work on their individual modules independently, and we staggered our integration such that we first integrated graphics and side-scrolling, then added game mechanics. However, if we were to redo this project from the beginning, there are several things we would have done differently. One recurring problem we had was merge issues with our GitHub. Since we didn't create a GitHub repository until the second week of our project, when we had already completed a significant portion of our individual tasks, we had problems with version control for the rest of the duration of the project. Thus, if we were to redo the project, we would have started with creating our GitHub.

In addition, we could improve upon our design for Mario level creation. In our design, we made a single Mario level in which we hard-coded obstacles. However, ideally we would have had a single level-making process that we could instantiate within a module structure. This would allow us to create more varied levels more easily and would have added complexity to

our project. Despite these desired improvements, we were ultimately able to accomplish the majority of our goals on time.

Appendix: Verilog Source Code Link
https://github.mit.edu/joseguaj/MarioBrosClassic