6.111 Final Report: Dance Dance Revolution

Andrea Bolivar, Grace Quaratiello

Table of Contents

Introduction **1**

Hardware **1**

System Overview **4**

- Sensor Module 5
- Audio Module 6
- Game Module 8
- Visual Module 9
- Selector Module 12

Timeline and Organization 13

Challenges **14**

Appendix **15**

Introduction

We implemented a version of the popular arcade game Dance Dance Revolution (DDR). The goal of the game is to score points by stepping in time with choreography that is displayed on a screen. Conventional DDR boards detect where a player is stepping by using a pressure-based sensor, but our hardware detects the position of the player's feet based on the interruption of laser beams.

We went into the brainstorming process wanting our final project to be a fun, interactive game, and the floor piano project shown in class inspired us to create a game based on sensing steps. DDR, the most prominent game of this type, was the first idea that came to mind. We had both played DDR when we were younger, and who doesn't love to dance?

Hardware

Andrea and Grace

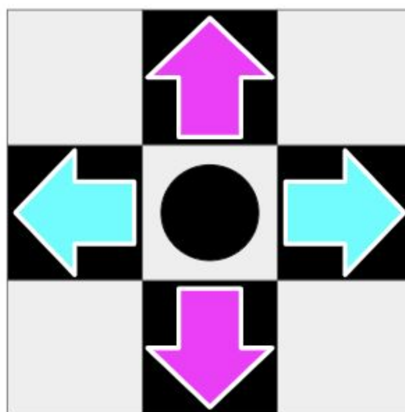

Figure 1: Dance Dance Revolution Example Pad

A Dance Dance Revolution pad can be represented as a nine-square grid where four squares hold directional arrows and one square is the center "default" square (Figure 1). In our initial proposal, we stated that we would have 8 arrows on the pad, but the VGA monitor was not big enough for all eight arrows to be displayed along with the score and other images. We ruled out downsizing the arrows because the player needs to be able to see them clearly. We ultimately determined that the diagonal arrows were not critical to the game and went with the classic four-arrow pad.

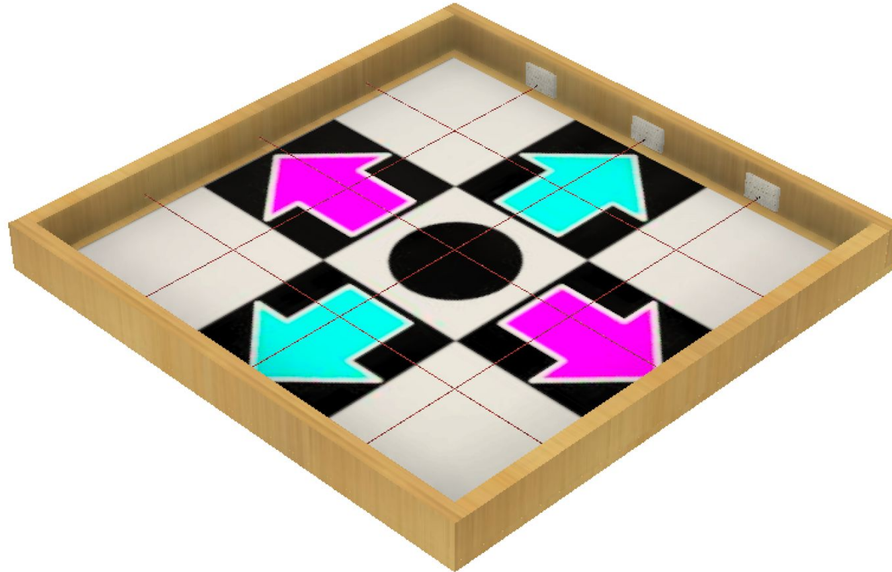

Figure 2: DDR Board Model

We designed a 4-square-foot board (Figure 2), where three square feet are used for gameplay and one square foot creates a six-inch buffer around the perimeter so the user doesn't feel cramped or restricted while playing. Four two-inch high rails line the board. Two of the rails each hold three lasers placed one foot apart from each other, creating nine laser intersections on the board in the center of each square. The other two rails each hold three phototransistor circuits. Each circuit is placed opposite a laser, as the phototransistor acts based on the light it receives. The player's position can then be sensed based on which laser intersections are being interrupted at a given time.

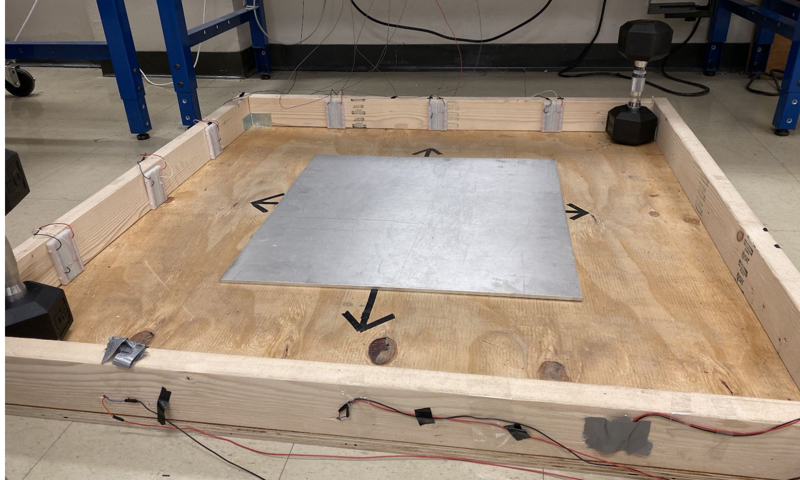

Figure 3: Wooden DDR pad

We contemplated letting the rails sit directly on the ground, but decided to secure the rails to a plywood base for stability. We constructed our board (Figure 3) from a 4 foot by 4 foot sheet of plywood and four wooden 2 inch by 4 inch rails cut to form the correct perimeter. We first attached the four rails together with corner L-brackets. Then, we drilled 1.5-inch-long wood screws through the plywood and into the rails.

In order to place the six lasers, we drilled three holes into each of the two rails designated to hold them. The holes are approximately 5 millimeters larger in diameter than the lasers, effectively securing the lasers inside the holes so they could point in the correct direction. We initially discussed using hot glue to hold the lasers in place inside their holes, but we decided against it once we recognized that the lasers became dim and needed to be replaced relatively often. The lasers are secure in the holes, but their position isn't fixed, so they have to be aligned with their corresponding phototransistors in between some games. However, we believe that replacing the lasers would have taken much more time had they been hot-glued in place. Each laser was placed in series with a 100 ohm resistor and powered in parallel with the other lasers with 3.3 volts from an external power supply. We placed this breadboard on the outside of one of the rails.

We attached six small breadboards with adhesive backs to two of the rails. The breadboards give us the flexibility to adjust the position of a phototransistor depending on where the laser hits the board. This came in handy, as the lasers aren't fixed and sometimes shifted in between (or during) testing and gameplay.

We built a voltage divider on each of the six breadboards to measure the voltage drop across the phototransistor. On a breadboard, a phototransistor was placed in series with a 4.7 kiloohm resistor. We probed the circuit at the output of the phototransistor and wired it to a JB port on the FPGA. The six breadboards are powered in parallel using the 3.3V from the FPGA. This wiring for power supply was done on a small breadboard placed on the outside of one of the rails, like that for the powering of the lasers.

System Overview

Our system is comprised of five main modules: sensor, visual, selector, audio, and game logic. The sensor module processes the input signals coming from the JB ports based on the outputs from the six phototransistors, determining which squares a player is stepping in at a given time. This information tells the game module how to update the score for a given step. The visual module generates arrows based on data it reads from a ROM that stores a COE file containing hard-coded five-bit numbers. Once the visual module determines that an arrow has reached the top of the screen, then it sends that `correct_data` to the game module to compare it with the `intersection_data` from the JB ports at that time. It also outputs a `ready_in` signal to notify the game module to adjust the score according to the comparison. The selector module generates the menu image and receives user input to determine the `speed` at which the arrows move up the screen and the song to be played. The user selects the `level` by adjusting two of the switches on the FPGA. The `level` selected by the user is sent to the audio module, which plays stored audio for the chosen level. Our modules required a few different clock domains, so we used Vivado's Clock Wizard to generate a 25MHz clock (to interface with the SD card) and a 65MHz clock (to interface with the XVGA display) from the original 100MHz clock.

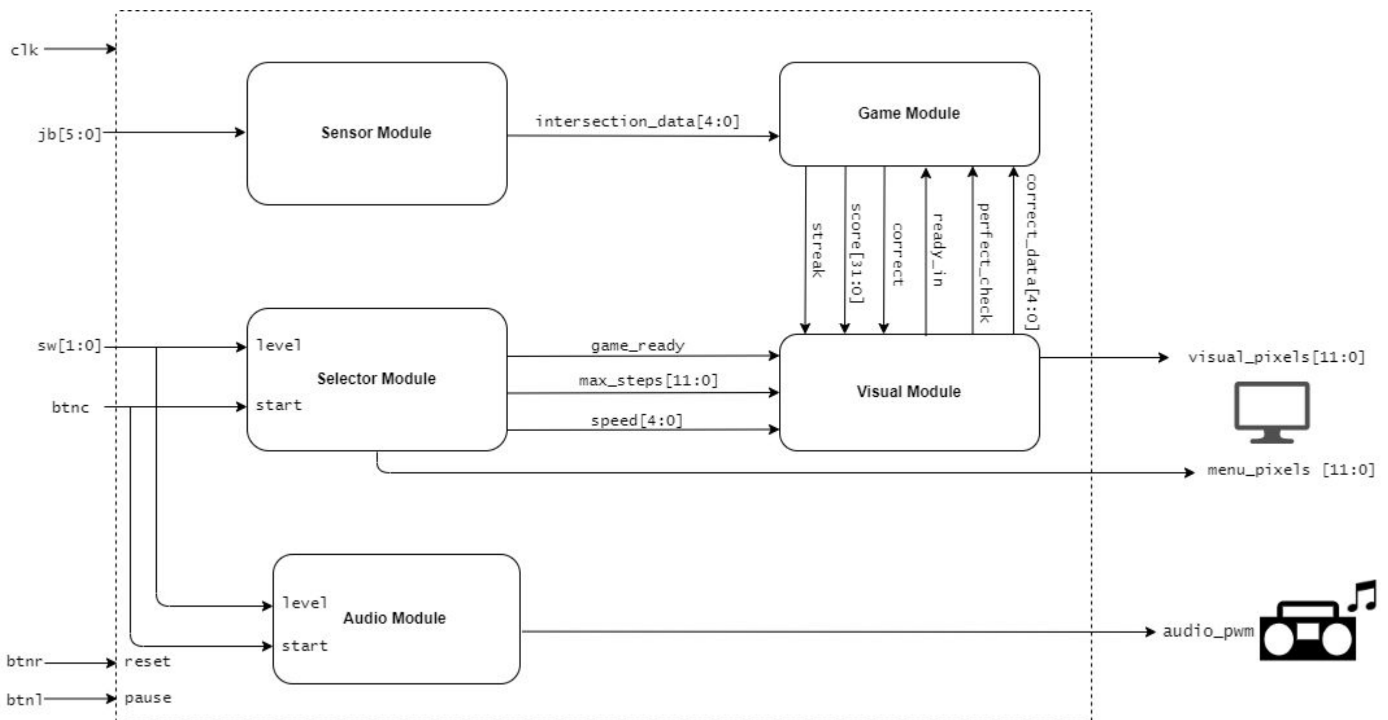

Figure 4: Overall System Diagram

Sensor Module

Andrea

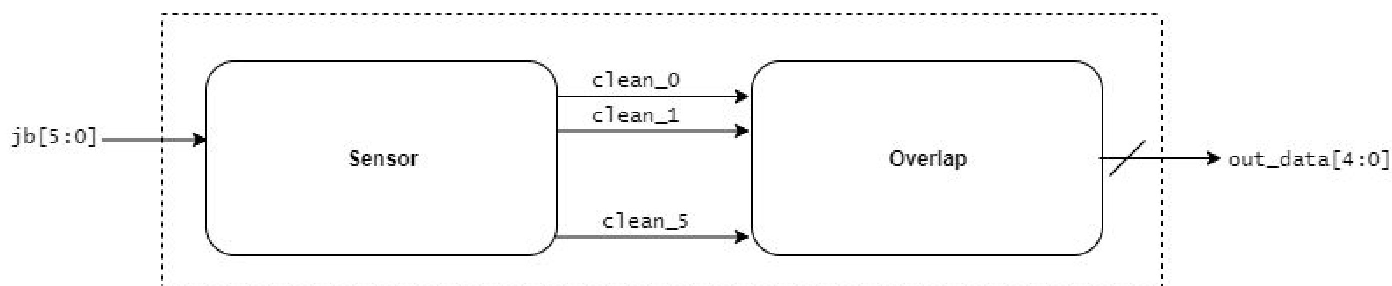

Figure 5: Sensor Module Block Diagram

The sensor module processes the six inputs from each phototransistor (`jb_sensors`). These inputs are debounced to create clean outputs `clean_0` through `clean_5` to send to the overlap module, where the locations of the player's steps are determined. The bit's value is a 0 if light has reached the phototransistor and 1 if the light has been blocked. This directly translates to a 1 if a player is stepping in the corresponding area since a foot is blocking the light. The overlap module will take in six one-bit inputs, and determine which of the four possible arrows are being stepped on. The output will be a five-bit number of the form ABCDE. At each index, a 1 means both of the lasers that form the corresponding intersection are completely blocked, and 0 means either one or neither of the lasers are blocked. This converts the player's position to a number that can be compared to the correct data stored in memory. C, the index that corresponds with the center, is is not used, however, it is included because the game could increase in complexity by incorporating the center as another scored position[1].

This module was tested by displaying each of the clean outputs on the board's LED display (above the switches). This helped us to debug the hardware since we could quickly detect during the game which lasers and transistors were not lined up correctly. When nobody is standing on the platform we would expect all of the LEDs to be off; because all of the light is passing, all of the clean outputs should be 0's. If there are any LEDs on, we would check to see if the problem is the phototransistor or laser. If the laser is not shining onto the breadboard, we would readjust the laser. If the laser is in fact hitting the breadboard but the phototransistor isn't receiving the light, then we would move the phototransistor to the correct place on the breadboard.

[1] C corresponds to the center space. We set the value at this index to always be zero in both the overlap module and the choreography because the player is never scored based on stepping in the center. Since the value at this index always matches, the player is neither rewarded nor penalized for stepping in the center, default square.

Audio Module

Grace

Our implementation of Dance Dance Revolution contains three different songs, depending on the level that the user chooses to play. We initially planned on storing audio in BRAM, but quickly learned that the total block memory available on the FPGA would not have allowed us to store three full-length songs. We ultimately decided to use a 2 GB microSD card.

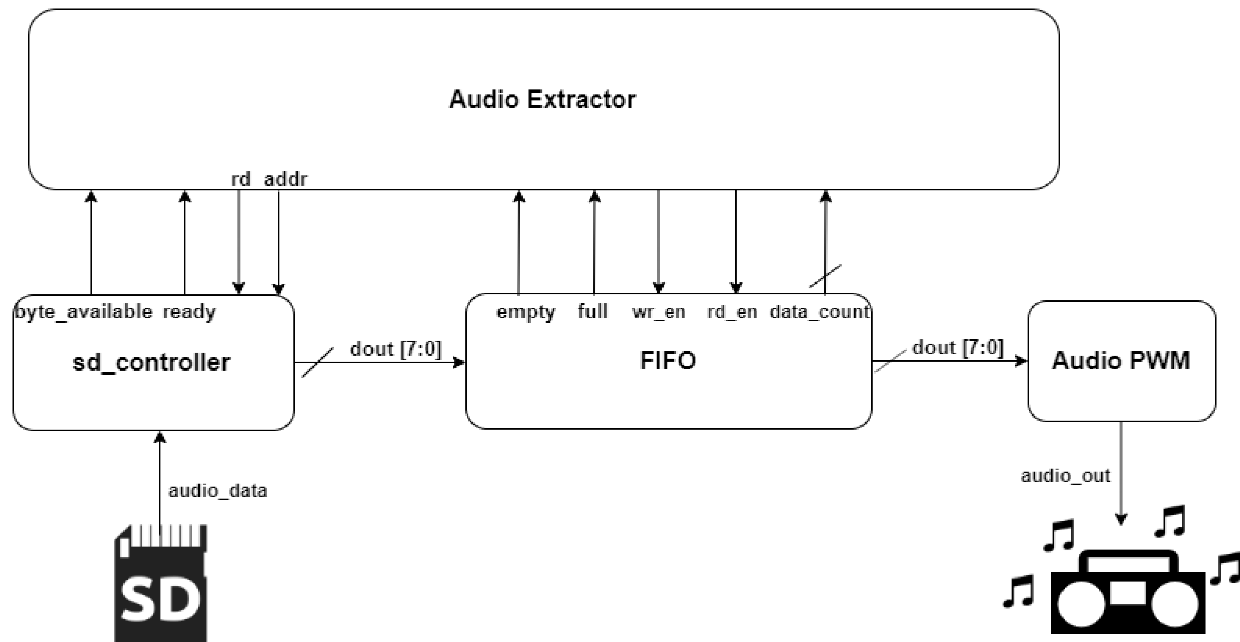

Figure 6: Audio module block diagram

We were given a module, `sd_controller`[2], that interfaces with a microSD card and performs 512-byte reads and writes, running on a 25MHz clock. Using a hex editor, we pre-wrote our songs as 32 kHz unsigned 8-bit .WAV files onto the SD card, so we only performed read operations. In order for a read operation to take place, `sd_controller` must be outputting a high `ready` signal at the same time that it receives a high read-enable signal, `rd`. This `ready` signal occurs whenever the SD card has the ability to perform an operation, so `rd` is kept high whenever the system should be playing audio, and it is low when it should not. Additionally, an SD card is split up into 512-byte sectors, so the input address, `addr`, must be a multiple of 512 in order for a read operation to occur.

During a read operation, the 512 bytes are presented on the module's output pin, `dout`, one byte at a time. Every time a new byte is available, `sd_controller` outputs a high `byte_available` signal. The rising edge of this signal is used to keep track of when a byte

[2] The sd_controller module "allows reading from and writing to a microSD card through SPI mode." This module was provided to us. Consulted "SD Card Read/Write" tutorial by Jono Matthews and the 6.111 HeartAware project from Fall 2015 in developing this aspect of our project.

should be read to the FIFO (first in, first out) and to count the total number of bytes read from the SD card.

Because bytes are being read from the SD card at a much faster rate than the song is sampled at, each byte is sent to a FIFO once it is presented on `dout`. The FIFO stores the bytes from the SD card and sends them to the PWM (pulse-width modulation) module[3] at the sampling rate of the song, 32 kHz, in the order they were received. This difference in clock frequency also caused the FIFO to fill extremely quickly and stop allowing reads. To combat this, `rd` (the read-enable for the SD card) should be set low whenever the `data_count` in the FIFO reached a certain threshold at least 512 bytes lower than its total capacity. This allows the current read operation to finish without initiating a new one. Once enough space becomes available in the FIFO, `rd` can be set high again.

In order to accomplish the necessary behavior, the FIFO's read-enable signal, `rd_en`, must be set high at the same frequency as the sampling rate of the song, and its write-enable signal, `wr_en`, must be high at the rising edge of the `byte_available` signal from the SD card. Both the read-enable and the write-enable should be pulses—leaving the write-enable high for too long allows the same byte to be written to the FIFO multiple times, and leaving the read-enable high for longer than one clock cycle outputs the bytes from the FIFO at the incorrect sampling rate.

Once the audio data is processed by the PWM module, it is sent to the FPGA's audio jack to be played through a speaker.

The audio output would sometimes be delayed by a few seconds after the game began. We noticed that the audio would start playing almost right away when we didn't start the game immediately after resetting it. We added a "loading" image to our menu to indicate to the user when the game was and wasn't ready to begin.

This module was primarily tested with an Integrated Logic Analyzer (ILA), which allowed us to analyze various signals as our system operated on the FPGA. Debugging by listening to the audio would show when something was wrong, as the audio wouldn't sound correct, but it was not effective at pinpointing any problems. The ILA helped determine that the `full` flag from the FIFO was almost always high, resulting in many bytes from the SD card being thrown away without being sent forward. This prompted us to introduce an upper `data_count` threshold for allowing SD card reads. This quick fix prompted the entire audio system to work correctly for the first time.

[3] PWM module provided by Gim Hom.

Game Module

Andrea and Grace

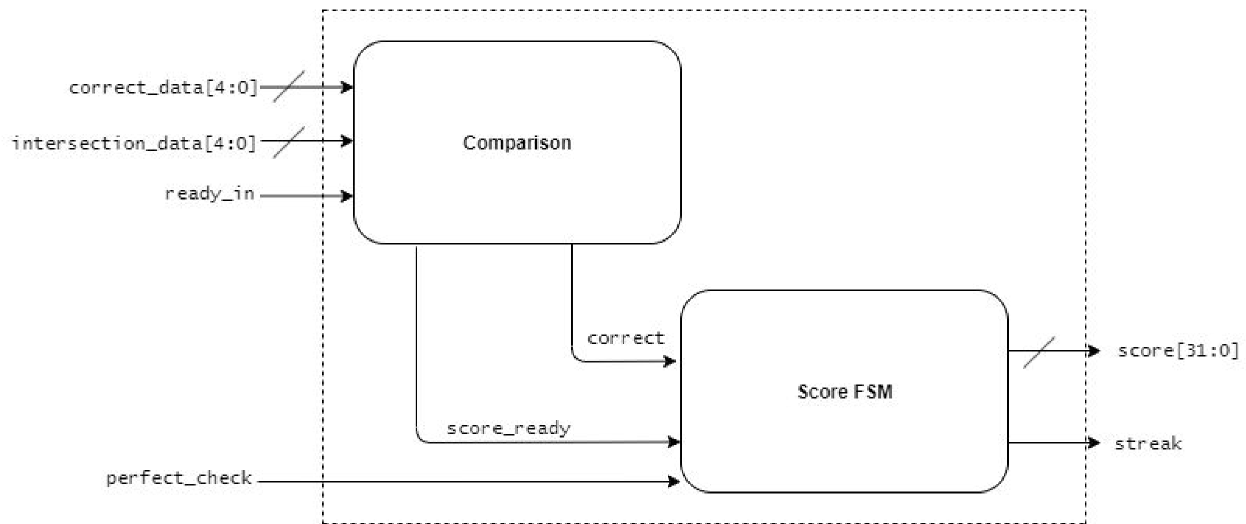

Figure 7: Block Diagram for the Game Module

This module takes in the processed data from the sensors and compares it to the correct data to determine if the player is stepping on the correct arrows at the right times. If `intersection_data` and `correct_data` are equal, then `score_ready` is set to 1, indicating to the score FSM that the player is correct and should be awarded points. The score increments when the player is correct and stays constant otherwise. This game module also takes in a `perfect_check` input to know to award a player more points for better performance. If a player's step is marked as perfect, an additional two points are awarded. Steps are judged as perfect or imperfect in the visual module. Additionally, a player enters a streak if the five previous steps have been both correct and perfect. The streak is reset after any incorrect or imperfect move. Two additional points are awarded for each correct and perfect step while a player is in a streak.

To test this module we created a testbench that checked how `updated_score` changed as the comparison module compared `intersection_data` and `correct_data`. At this point we also had a baseline game working with switches, so we also did more advanced testing with the perfect signal by looking at the score increasing on the screen as we played with the switches. [4] We also tested the `rst_in` button to make sure it reset the `updated_score` to 0.

[4] An ILA was used to see the signals while the game was played.

Visual Module

Andrea/Grace

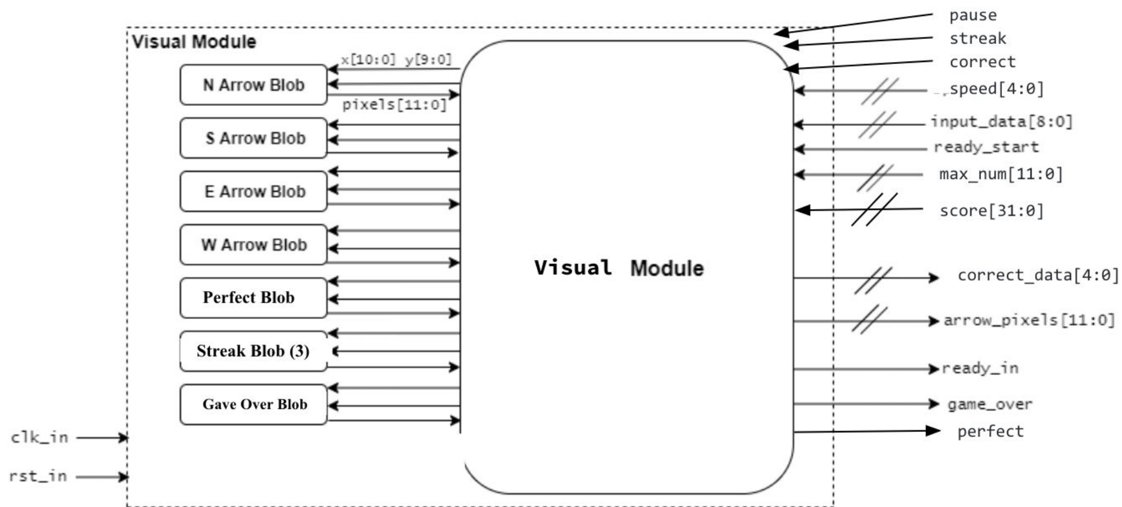

Figure 8: Block Diagram for the Visual Module

This module handles the timing of checking for correct and perfect steps and determines which pixels should be displayed on the XVGA display during gameplay. There are four different arrow blobs corresponding to the four different arrows that a player can step on to earn points. During the game, these arrows will move up the screen according to steps stored in memory.

The visual module reads predetermined choreography from a ROM that holds a COE file consisting of five bit numbers in the same form as the `intersection_data` in the sensor module. The numbers read from the ROM determine which arrow blobs to display—a 1 at a given index indicates that arrow should be displayed. A new value is read from the ROM once the previous row of arrows reaches the top of the screen and the next row needs to be generated.

This visual module also takes in a `ready_start` signal to know when the pixels displayed on the screen must switch from the pixels outputted by the selector module to the pixels provided by the visual module. While the menu should be displayed, the visual module outputs a 12-bit register of zeros. Once `ready_start` goes high, the visual module becomes responsible for the pixels on the XVGA display and assumes a nonzero value based on the images being displayed. Arrows appear at the bottom of the monitor with fixed x coordinates, and their y coordinates decrease (so the arrow moves upward) based on the `speed` given as an input.

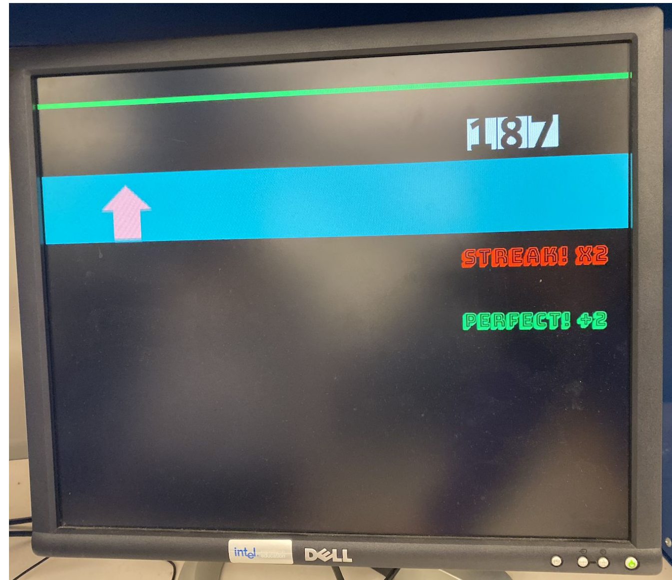

Figure 9: Game Display

The perfect region is the blue line on the game screen that has roughly the same thickness as an arrow. When the arrow is completely in this region (Figure 9), the perfect signal goes high when the `correct_data` for the step is sent to the game module. If a player gets the step correct in this region, the top bar turns green, the score increments appropriately, and a new line of arrows appears at the bottom of the screen.

If a player fails to step on the correct arrow(s) while it is within the perfect region, there is still time to get it correct. A step is considered correct if made before the tip of the arrow reaches the bar at the very top of the screen. In this case, the `perfect` signal sent to the game module is set low, but the player still receives one point for just being correct. When visual is ready for the game module to compare the user input with the correct data, it outputs a high `ready_in` signal.

While expanding on the visual module, we ran into a pipelining issue, which was discovered when the right side of an image appeared on its left instead. We discovered that we had four cycles of latency due to our modules taking clock cycles to read image data from ROMs. To fix this problem, we created the parameter HCOUNT_LATENCY, which is set equal to four, and reads ahead in the ROM.

While the game is in progress, the score is displayed on the screen. In order to display the numbers, there are three score blobs: one for the hundreds place, one for the tens place, and one for the ones place. The score received from the game module is sent into a combinatorial binary to BCD converter that uses the double dabble algorithm[5] to calculate the digit that each blob should display. The COE file given contains 48 pixel by 48 pixel images of the digits 0

[5] More information on the double dabble algorithm (binary to BCD conversion): https://my.eng.utah.edu/~nmcdonal/Tutorials/BCDTutorial/BCDConversion.html

through 9. Each blob takes in the desired digit, `num`, and the ROM address it should read from is calculated by:

```
image_addr = ((hcount_in + HCOUNT_LATENCY) -x_in) + (vcount_in-y_in) * WIDTH + 2304 * num
```

The game notifies the player of either executing a perfect step or being in a streak by displaying those words when necessary. When perfect is high and the player is correct, the perfect image displays, and when the player is in the streak state, the streak image displays. The images display together if both conditions apply to a player at a given time (Figure 9).

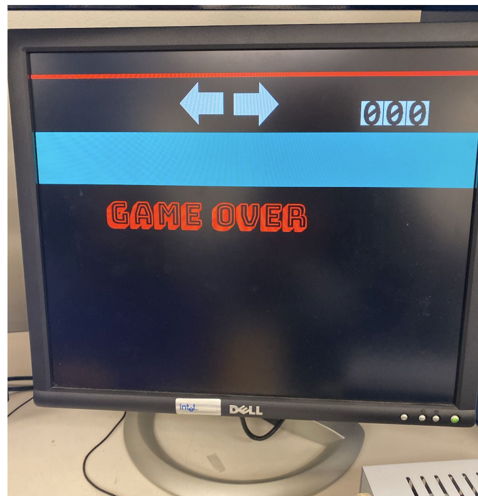

Figure 10: Game over

Once the game is over, and the `max_num` of arrows have been displayed, the visual module will then output game_over pixels (Figure 10) to indicate that the game is over. To reset the game after this happens, then you will need to press the reset button to bring you back to the menu screen operated by selector.

We used an ILA to test that the `ready_in` signals are sent out correctly, and we also used it to see the score at a given point. It also helped to make sure that the game module was in fact sending the appropriate signals. To test the location of the arrows and their vertical height, we displayed them on the screen and adjusted their x and y values accordingly.

Selector Module

Andrea

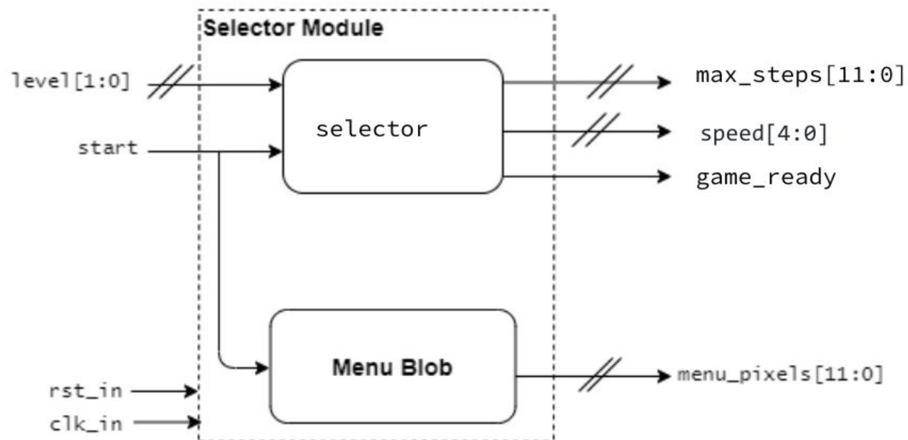

Figure 11: Block Diagram for Selector Module

The selector module determines the level of the game based on user input, and displays the menu. The `speed` at which the arrows move on the screen during gameplay is based on the `level` the user selects This module takes the `start` input (btnc) and a 2 bit number (from `sw[1:0]`) representing the level of the game (easy, medium, or hard). The selector module outputs a `ready_start` signal and the appropriate `speed` for the arrows to move up the screen. Selector also determines the length of the choreography based on the level chosen to make sure that all games have roughly the same duration. Faster levels have more choreography steps than slower ones.

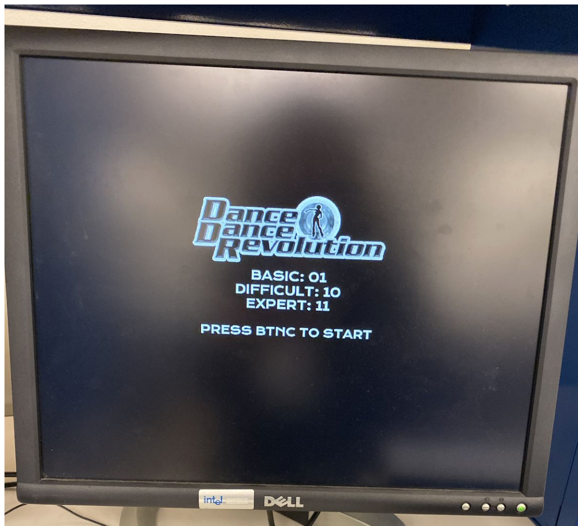

Figure 12: Game Menu

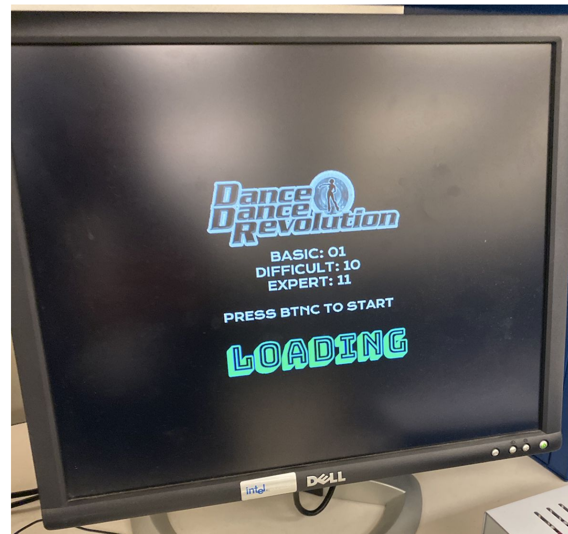

Figure 13: Game Menu with Loading Visible

The start menu (Figure 12) displayed before the game begins is generated in the menu blob. The word “loading” appears when the game has been reset (Figure 13), as it takes a few seconds for the SD card to become able to carry out read or write operations. The timer module from Lab 4 is used to display “loading” for a set amount of time. The selector module ultimately determines the pixels to output based on the `start` signal. If the game hasn't started then it outputs the menu pixels, otherwise it outputs 0’s.

The selector module was tested by visually checking that the menu and loading images displayed properly and using the hex display to view the level selected along with the determined speed.

Timeline and Organization

We started working on this project early since we knew that the hardware would take some time to fully set up and test. We stuck to our timeline well, so we were able to achieve all of our goals.

We designed and constructed the hardware together because it was one of the very first steps of the project and it was important for both of us to know how to troubleshoot and adjust it during the testing stages of the project.

We deviated from our original distribution of tasks it once we integrated the project and started working on the stretch goals due availability when we were able to start new tasks. This allowed us both to each contribute to almost all of the project areas and get a good understanding of the entire system.

Commitment	Baseline Goals	Stretch Goals
Andrea ● Sensor Module ● Visual Module ● Selector Module Grace ● Game Module ● Audio Module ● Selector Module (as pertains to Audio)	System Integration: A functioning game where: Andrea ● Three game levels where the arrows move at different speeds Grace ● Score counts up normally at the correct times	Andrea ● Pause the game (as pertains to Visual) ● Game over screen ● Complex scoring FSM (goals, streaks, perfect) and visual signals too Grace ● Pause the game (as pertains to Audio) ● More songs available on SD card. ● Display the score on the monitor. ● Updated menu screen and added loading screen and images based on scoring state (perfect steps and streaks)

Challenges

Andrea and Grace

The largest challenge we collectively faced during the course of this project was working with our hardware, especially the plywood base. The plywood base would warp overnight in unpredictable ways due to the humidity in the lab. This caused the phototransistors and the lasers to not line up properly each time we played or tested. Right before our final video presentation we struggled with this problem, since the wood was specifically warped in the center overnight, but we placed a large metal sheet in the center of the box to resist changes due to the weight of the player.

If this project were to be recreated, then I would advise against using the base plywood and only using the 4 wooden rails. The initial thought process for using the base plywood was to reassure us that the corners of the box would be secure and not shift. Although, since we did use L-brackets to secure the corners, the base plywood only complicated our design.

Grace

Figuring out how to interface with the SD card in order to get audio to play was the most prominent challenge I faced while working on this project. While there were materials available so I could familiarize myself with the SD card structure and operations, they lacked some crucial details. I was not familiar with the concept of using a hex editor, so I didn't even have the files written to the SD card correctly for a significant amount of time. Even if my code was correct, I would have attributed any of my problems to incorrect logic or a bug.

Luckily, I figured out how to correctly write the files to the SD card using HxD, a hex editor, and I was able to rule that out when I had problems moving forward. It took me a little while to get my logic for interfacing with the SD card correct, but using an ILA helped immensely by allowing me to see how all of the signals were acting during operation.

Once I got the audio working, I noticed that a low-frequency noise played before any of the songs began. Because I believed that I had written to the SD card correctly, I attributed this problem to Verilog. Turns out, there was irrelevant data written at the beginning of each of my .WAV files, and raising the start address for each song solved the problem.

It was extremely rewarding to finally get the audio to play, because it did take a lot of effort for it to get working. However, I do believe that I struggled for much longer than I needed to. This experience taught me to get familiar with items I will be working with before I try to integrate them into a system and to look beyond Verilog for the root causes of issues. I could have made progress much sooner if I had realized that I was writing data to the SD card incorrectly in the first place.

Andrea

The initial integration step between the visual and game modules was challenging for me. The difficult part was making sure all of the score_ready signal would only get sent once. The visual module requires the 65mHz which is faster than the clock at which the pixel locations were being updated. Since the signals are sent based on the slower clock, then I had to make sure that it was only being sent once and read once. At first this clock problem was unclear, and I was able to locate the problem using an ILA. It was resolved by checking for the rising edge of each signal sent with a different clock.

Appendix

Verilog Code: https://github.com/andreab98/6.111_ddr