Lab 4 Report: Dance Dance Revolution
CS M152A W19 — Lab 5 — Group 7

Duy Duong 505-183-737
Ryan Lauri 105-021-006

Introduction

For this lab, a game in the style of Dance Dance Revolution (DDR) was implemented on the Xilinx board as according the group's final revised proposal.

On startup, the game greeted players with a menu on the computer display (connected through the boards' VGA controller) where they could choose different levels to play in addition to the game's difficulty and color scheme settings using a Pmod numeric keypad as selection input. Once players set their settings and began a level, a predefined stream of "dance moves" (hereby called 'notes' within the context of this project) began to 'fall' down from the top of the computer display in four discrete columns on the display towards hitboxes near the bottom of the display.

During the game, players were tasked with pressing buttons corresponding to specific columns when they had a 'note' in the hitbox. For each note entered successfully, the game played a sound frequency corresponding to a certain column over a buzzer connected to a portion of a Pmod connector acting as a PWM output. Additionally, the players score (starting at zero) was incremented on the board's seven-segment display, with more points being awarded for longer streaks of correct note hits. The high score earned for each level were stored for one reset cycle and displayed on the seven-segment display while selecting levels.

The remaining portion of the report will talk the intricacies of the implementations of each component of this game's functionality in addition to addressing our testing methodology and possible improvements that could be made.

Design Documentation

1 Visual Gameplay

1.1 Display

The module responsible for the main VGA display (`vga640x480`) consisted of a pair of horizontal and vertical counters that went over every pixel once every pixelClk cycle (25MHz clock) on the screen from the top left corner, generating the hsync and vsync signals when they were in

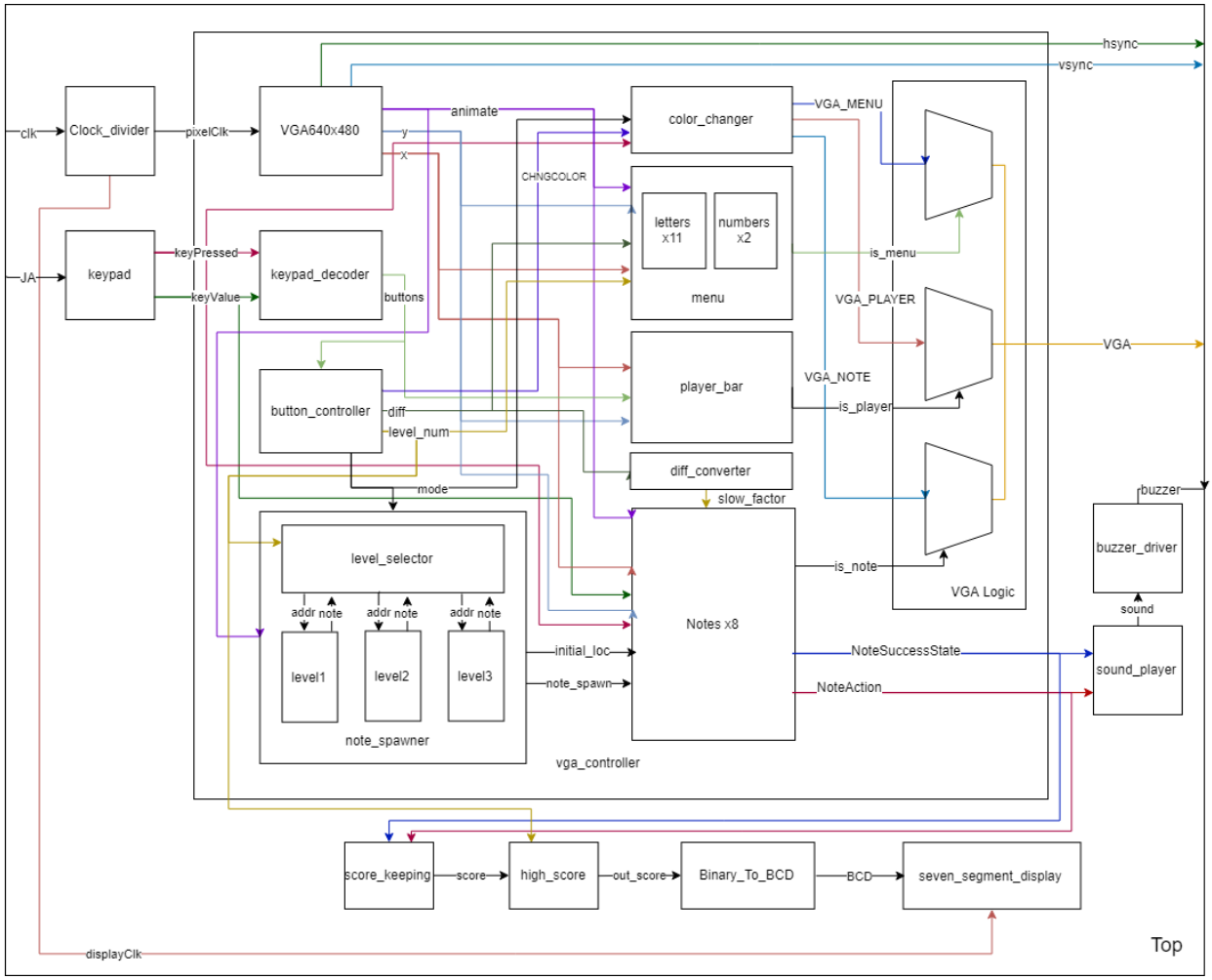

Figure 1: Top level block diagram showing the relationship between the different modules and submodules.

the correct position. When the counters were in the visible pixel range, the output wires x and y were set to the location of the current pixel being drawn, with (0,0) being the top left and (639,439) being the bottom right. Using these output wires, other modules drawing to the VGA display could correctly signal the VGA controller exactly when to draw their values on the screen.

The other important output of the VGA module was the *animate* signal, which goes active high for one clock cycle at the very end of the final drawn pixel. This signal was used by all other modules utilizing the VGA as it served as an enabler for them to update their pixel locations. As the horizontal and vertical counters of the VGA module continue to go into the vertical back porch after animate was set to high and the screen would not be redrawn until the counters were reset to 0, the other modules can safely update their position without interfering with the VGA display.

1.2 Player

The player was represented as a single line of 4 hollow blocks at the bottom of the screen, each of which had the same size as 1 note (100 pixels in width and 30 pixels height) but had a different color from the notes depending on which color scheme was currently selected. Upon pressing a button corresponding to one of the 4 boxes, the box chosen would fill up briefly for half a second, signaling that the input had been received. This effect was implemented using a delay timer of half a second that refreshed to its maximum value upon a key press. When the delay timer was not 0, the currently chosen box would fill up, ensuring that the box would continue to be filled when the key was held down and that it would return to hollow again briefly after the key was released. The four boxes from left to right were mapped to buttons 4, 8, 2, and 6 on the keypad respectively, as shown in the simple illustration below:

Figure 2: Visual representation of the player bar, with the mapped key value displayed inside of the boxes

1.3 Notes

Periodically during gameplay, notes would fall down from the top of the screen in a certain pattern dictated by the level currently selected. These notes were implemented as rectangles of 100 pixels in width and 30 pixels in height, with a default color of blue (changeable with the color scheme option). The individual notes, once spawned into the screen, fell at a constant speed (ranging from 20 pixels to 60 pixels per second depending on the difficulty chosen) in a straight line, eventually overlapping with the player's bar. Each note contained its own logic to handle collision with the player bar as well as accepting input from the player, as described below.

When a note was spawned, it was assigned one out of four possible location (corresponding to the 4 boxes in the player bar) as its initial location, which it remembered as its column value. When a player presses a button, that button value was mapped to one of the 4 columns. If the

player's button press was in the same column as the current note and the note was colliding with the player bar (colliding was defined as at least 50% of the note's area was inside the player bar, or that $|player\_bar\_y - note\_y| < HALF\_NOTE\_HEIGHT$, the note was registered as a hit. When a note was hit, the player's score, combo counter and multiplier would go up, and the note would disappear.

If the player could no longer hit the note (there was no more possibility of collision as the note continued to fall downwards past the player bar), the note would despawn and a failure would be registered. Upon receiving this failure notice, any multiplier/combo counter would be reset to 0. To interact with the other modules in the game, the note modules output a NoteSuccessState value that was set to 1 if the note was successfully hit and 0 otherwise, as well as a NoteAction value that went active high for one clock cycle every time a note despawned, whether from being hit by the player or from reaching the bottom of the screen.

1.4 Level loading

Level loading was accomplished using a module called `level_selector`). This module took in an input *addr* that corresponded to the zero-indexed note number desired by the note creation module in addition to an input *level_num* that corresponded to the level being played. Additionally, three separate modules, `level_rom1`, `level_rom2`, and `level_rom3` were instantiated within this module, each one taking *addr* as an input and outputting the note location for that address as *note*.

The individual 'level' modules functioned by initializing pre-generated two-dimensional arrays of locations (generated randomly with a Python script for each module) and simply setting *note* to the four-bit location at the address of the array fed into the module by *addr*. These note location arrays were comprised of 49 four-bit values, with the first 48 being grey-code (each one corresponding to a different game column for the note to be animated on) and the last being all ones to signify the end of the level. The *note* output of `level_selector` was simply set to the *note* value output from the level module corresponding to the level passed in as *level_num* through a switch case.

1.5 Note creation

Note creation was handled from within the `note_spawner` module, which further instantiated `level_selector` to receive note locations for the new notes being created. `level_selector` is instantiated with *note_count* (a register that kept a count of the number of notes that have been loaded starting at 0) as the *addr* input as a means to always have a *note* output from `level_selector` corresponding to a new note location to load into the eight notes' location string when needed until the end-of-level is reached. Specifically speaking, end-of-level was chosen to be signified by a four-bit location of all ones, 4'b1111. Once this note was read in from `level_selector`, an else-if statement taking precedence over the mid-game note creation code set a register *ended* to one repeatedly, ending the level playthrough in a simple but effective manner.

In the case of `note_spawner`, the input *rst* signal was actually taken from the *mode* output from `button_controller` such as to restart the note creation process every time a new level playthrough begins. When *rst* was triggered, the module spent nine clock cycles loading initial

notes. For the first clock cycle, register values needed for spawner operation were reset and the initial location load-in process is started by storing one in *rst_pressed*, a register used as a sort of pseudo-boolean to trigger the process. *rst_pressed* remained set to one for the next eight clock cycles until *note_count* reaches eight. During this process, one of the eight 'note' instantiations' initial locations were loaded into the output *locations* string from the *next_note* register, which connects to the output 'note' of `level_selector`, for each clock cycle. Within the output *locations*, each consecutive four bits corresponded to one of the eight 'note' instantiations' locations, with the four least significant bits being the location for 'note1' and the four most significant bits being the location for 'note8.' Each four bits of the 32 were loaded in separate clock cycles using a three-bit *rst_counter* tied to a switch case with eight separate cases for each of the four-bit chunks. Additionally, a four-bit register *current_note* was used to track which note should be animated next by incrementing it from 4'b0000 up to 4'b1000 during this process.

As far as note animation goes, the code did limit the number of on-screen notes and stagger their appearances. Looking at the final else-if statement of `note_spawner`, the code checked to see if the input 'animate' is not zero, meaning that notes should be animated on the screen. Inside this code block, another if statement was used to check if *notes_on_screen* (a register keeping count of the number of notes animated on the screen) was already greater than or equal to 6 and just set the output register *note_spawn* to all zeroes in this case. Single bits were used of this register *note_spawn* by each of the eight *note* instantiations in order to tell the instantiation when to animate ("spawn") themselves on the screen. In the situation where *notes_on_screen* was less than six, a delay counter *note_delay* would run for 180 clock cycles (tied to *pixelClk*) such as to stagger the animation of each new note by 180 animation cycles. Given that the computer display operated at 60 Hz and that each animation cycle corresponded to one frame, a theoretical three-second delay was created between each note being generated on the screen. Once that counter reached 180, a switch case based on *current_note* was used to send a signal of one through *note_spawn* to the next of eight note objects that should be (re)animated with the freshly-loaded location. Once this occured, *current_note* and *notes_on_screen* were incremented as deemed appropriate (only values between 4'b0001 and 4'b1000 were used with *current_note*).

In the case of mid-game note location loading, the spawner depended on the input string *noteAction* used as an output by the eight notes. Note animation was staggered such that only one note should disappear off the screen at any given time, so *noteAction* was simply compared in a switch case against its eight possible grey-code values in any situation (each one corresponding to one of the eight notes disappearing off-screen). Each of these cases would load a new substring into the *locations* output depending corresponding to which note disappeared (e.g. if the first note disappeared, the four least significant bits had a new location set) from the register *notes_on_screen*, which received its output from `level_selector`.

2 Scoring Mechanics

2.1 Normal gameplay

Score was kept during normal gameplay within the `score_keeping` module. This module took in the *NoteAction* and *NoteSuccessState* strings generated from the eight separate instances of

`note`. As far as scoring mechanics of the game are concerned, played started gameplay sessions when they loaded a new level with a *score* of zero points and a *multiplier* of 1. For every correct note hit (triggered within the note's hitbox), the player's score was incremented by the multiplier's value.

As for determining the value of *multiplier*, a simple algorithm was designed where the multiplier would be incremented by one every time the player hit the same number of notes in a row, missing none of the notes in between. The number of notes hit was stored in a register called *score_streak*, and the register was reset every time the multiplier incremented, so the game did get rather hard to achieve higher scores. Whenever the player did miss a note, *score_streak* was reset to zero and the multiplier was reset back to one, thus further contributing to the game's difficulty. Theoretically speaking, the scoring mechanics of the game were designed to work up to 9999 (the max that could be displayed under our implementation in base-10), so the game could support longer levels.

2.2 Displaying score

The current score was displayed on the seven-segment display using knowledge gained from the previous lab. The score was changed to a 16 bit BCD number in the `BinaryToBCD` module, which used a simple "right shift then add 3" algorithm that was acquired from outside research (and referenced in the code comments). This BCD number was then wired to the `seven_segment_display` module, where a display clock of 763Hz was used to simulate the numbers on the seven-segment display, giving us a display frequency of 190.75Hz.

2.3 High score tracking

The highest score achieved for each individual level was saved using the `high_score` module, where the current level number was used as input *level* to select a high score register corresponding to that level. During gameplay, that register was then continually compared to the current score of the gameplay session , which was sent as an input *in_score*. This score was passed through directly as *out_score* for display purposes, but the high score register for that level was updated for every new score achieved during the level playthrough higher than the existing high score register for that level.

3 Menu

At any point during gameplay, the player could stop the current level and access the menu by pressing number 5 on the keypad. In this state, the player could change the current level, difficulty, and color scheme using certain buttons on the keypad that are detailed in each feature's section below. Note that due to how the code was designed to receive input from the keypad, each button has to be separately debounced with a half second delay timer to prevent the key input from being registered multiple times.

While in menu mode, the current level was reset, and the previous score was wiped. Instead, the player could see their highest score for the currently selected level where the score during

gameplay would otherwise be on the seven-segment display. The `high_score` module handled the display of high scores, as described in the above section.

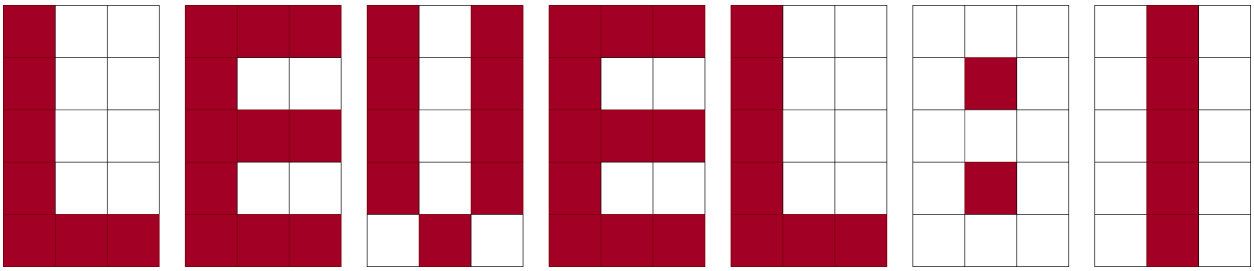

Figure 3: Level line in menu using blocks method

The menu was displayed as two lines of text with varying colors on a black background: LEVEL: level number and DIFF: difficulty number. Levels can range from 1 to 3, and difficulty can be between 0 and 1. These two lines of text were created by drawing the letters and numbers as boxes in a 3x5 grid, handled by the `letters` and `numbers` modules respectively. Each instance of these submodules handled a separate character, including the colon, and maintains their own 3x5 grid. The figure above shows a sample layout for the level line of the menu using the blocks method described.

The two modules took in an x and y coordinate pair that indicate the top left corner of the character, a 3-bit value that determined the character to be created, the current x,y position of the VGA pixel counter, two box width/height inputs that determined how large the individual blocks, and subsequently the entire character, would be, and returned an output wire that went high every time x, y was in that character range. Using this output value, the menu module was able to tell the VGA controller when to display the menu.

4 Sound generation

The game's main backbone for proper sound generation was `buzzer_driver`. This module was connected to a driving module called `sound_player` which instantiated it and set its input *sound* to *noteSuccessState* whenever *noteAction* wasn't zero, meaning that either a note reached the bottom of the screen and disappeared or was hit by the player. If a note just disappeared off the screen, it set its respective *noteSuccessState* bit to zero, so `buzzer_driver` was coded to outputs sounds when a bit of its *sound* input was changed to one.

Within `buzzer_driver`, whenever a new string was set on the *sound* input, it was stored in a register *sound_d* and a simple counter was run to keep the desired output sound stored in *sound_d* for a quarter second before setting it to the new 'sound' input—most likely zero unless perfectly timed. In the case where a new sound input arrived within the quarter second window, the value of *sound_d* was changed and the counter reset, so output sounds could be lost under this implementation for part of their time window, but the game didn't run fast enough for overlapping sounds to be an issue. The delay counter naming conventions were used within this context despite the counter being a duration counter as this code was reused from an already coded delay counter within the game's codebase.

The buzzer itself interfaced with the board by connecting its positive end to one of the output *Tx* pins on the board's Uart-compatible Pmod port, JD, and the negative end to an inline 300-Ω resistor that connected to one of the aforementioned port's ground pins, such as to lower the output volume to an appropriate level. Within `buzzer_driver`, four different clock dividers intended to output 50% duty-cycle square waves to the buzzer output pin were set up and tied to different one-hot encodings of *sound_d* in a switch case. One-hot encodings were utilized here as *noteSuccessState* (which the input *sound* was tied to) should only have one successful bit active at a time due to the notes' appearances on the screen being staggered. Under this design, when triggered, the first and fifth notes output sound at 1525.9 Hz, the second and sixth notes output sound at 762.9 Hz, the third and seventh notes output sound at 381.5 Hz, and the fourth and eighth notes output sound at 190.7 Hz.

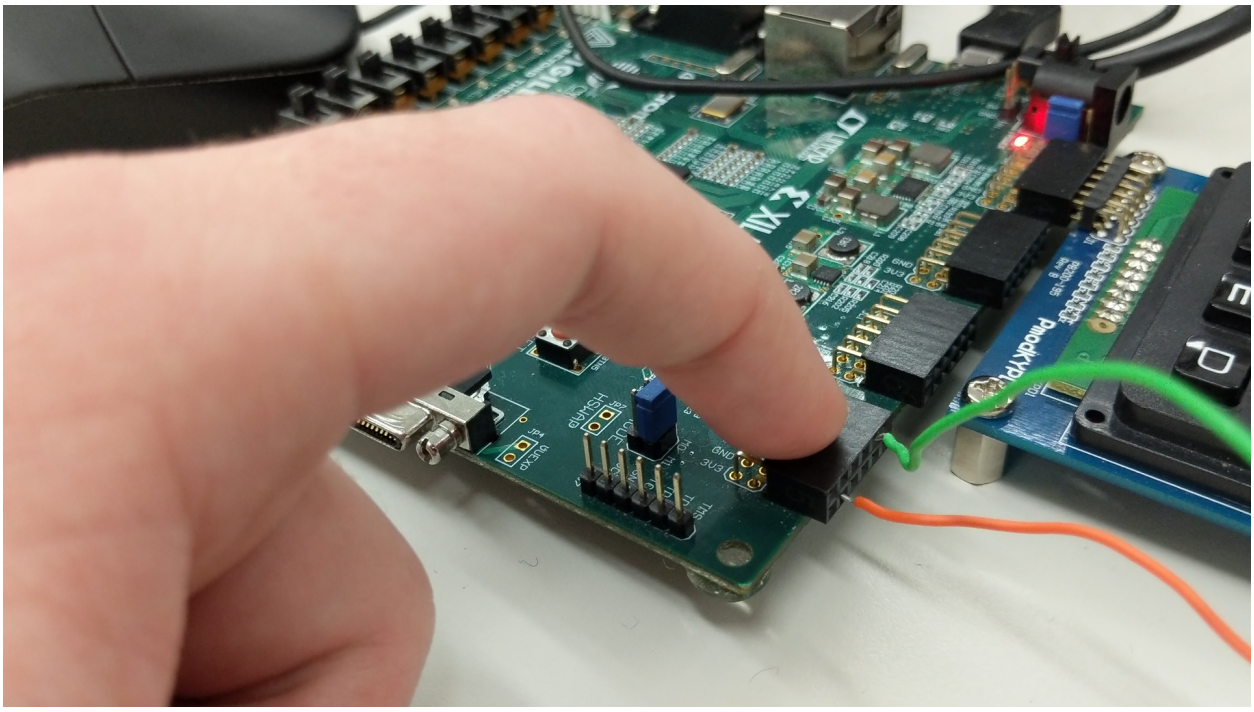

Figure 4: This figure depicts an image of the wiring used to connect the buzzer to the FPGA board. The green wire connected a "Uart Tx" pin on the board to the positive side of the buzzer, while the orange wire connected a ground pin to a 300-Ω inline resistor that connected on the other side to the negative side of the buzzer.

5 Difficulty Option

In order to change the difficulty of the game, players used the D button on the numeric keypad while in menu mode as a means to toggle between the default difficulty level of 0 and the harder difficulty of 1. This option was output from `button_controller` (used to handle all of the menu options) as 'difficulty,' where it was then sent to menu for menu display purposes and `difficulty_converter` to be converted to a value usable by the individual note instantiations as

slowFactor.

In essence, *slowFactor* was a two-bit parameter passed into the 'note' modules to be used for a delay counter for the note's movement when animated. In example, when passed in one for *slowFactor*, the 'note' instantiations would move down the screen every animation cycle whereas a value of three for *slowFactor* would lead to a movement down the screen happening every three animation cycles, thus making the notes appear to be moving slower to the player. The default difficulty level of 0 corresponded to a *slowFactor* value of three whereas the harder difficulty level, 1, corresponded to a *slowFactor* value of one. As a very simple module, `difficulty_converter` simply took in a one-bit input 'difficulty' and outputted a two-bit register *difficulty_converted* corresponding to the aforementioned *slowFactor* values.

6 Color Scheme Option

Using the E button on the keypad while in menu mode would cycle the player through some of the preprogrammed color schemes for the note, the player bar, and the menu text color. These color schemes were implemented using `color_change` where the button input from the player would be used to cycle through a list of preset colors, from which the VGA controller used to set the colors for each element of the game.

7 Reset

Upon receiving the asynchronous positive reset signal from the board, the game state was set back to the beginning: level 1, difficulty 0, no high score saved for any of the levels, and the default color scheme. The game would also start out in menu mode, giving players time to start the game at their leisure.

Simulation Documentation

In order to test our design, some simple test benches were created to test each module in isolation. Modules explicitly tested using simulation include `note_spawner`, `level_selector`, `score_keeping`, and `sound_player`. Other modules were directly tested on the board by synthesizing the completed code and observing the final output. Below are screenshots showing the design being tested on the board using the VGA connector with the display. For the moving parts of the display, trial and error was the main driving force behind our simulation attempts, and the final results are shown here in these two videos: Difficulty of 0 and Difficulty of 1.

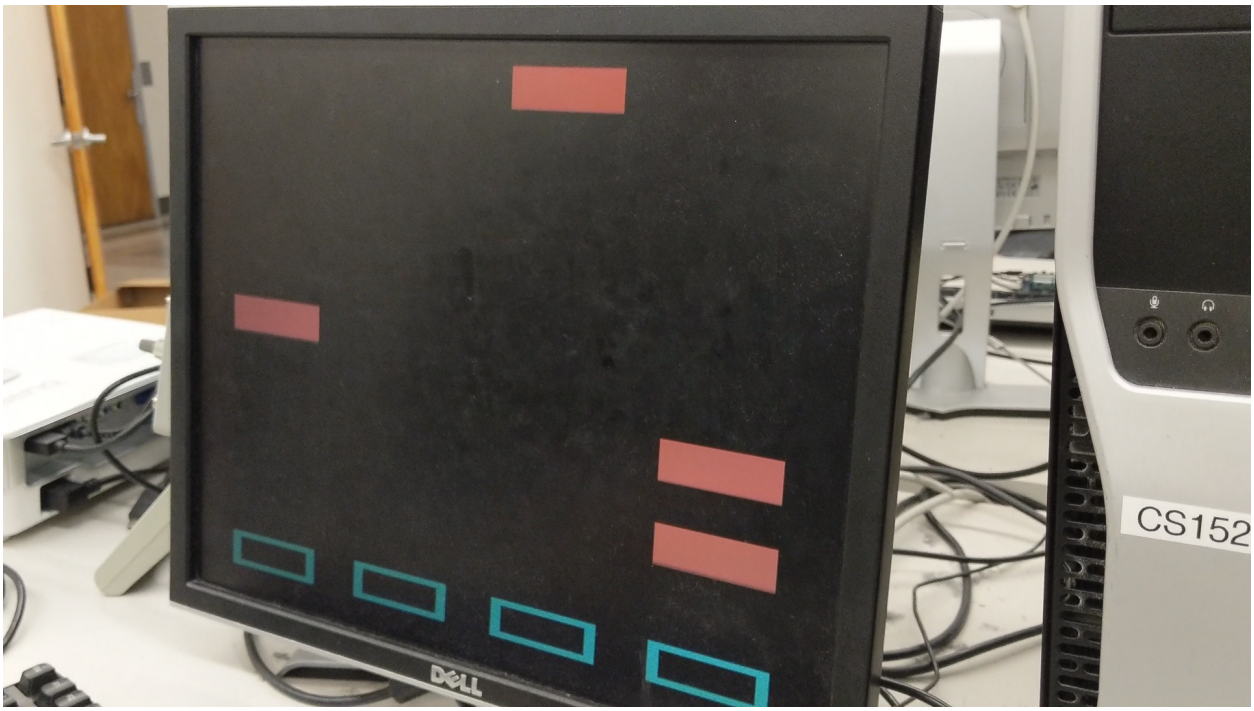

Figure 5: This figure depicts the main playing screen as seen in a typical game. At the bottom of the screen were four hollow cyan rectangles representing the player bar, and the four orange rectangles falling down were the first four notes of level1.

Figure 6: This figure depicts the same playing screen as seen in 5, but with an alternate color scheme for the note blocks.

Figure 7: This figure depicts the menu screen on startup, with the default settings of level 1 and difficulty 0 in place.

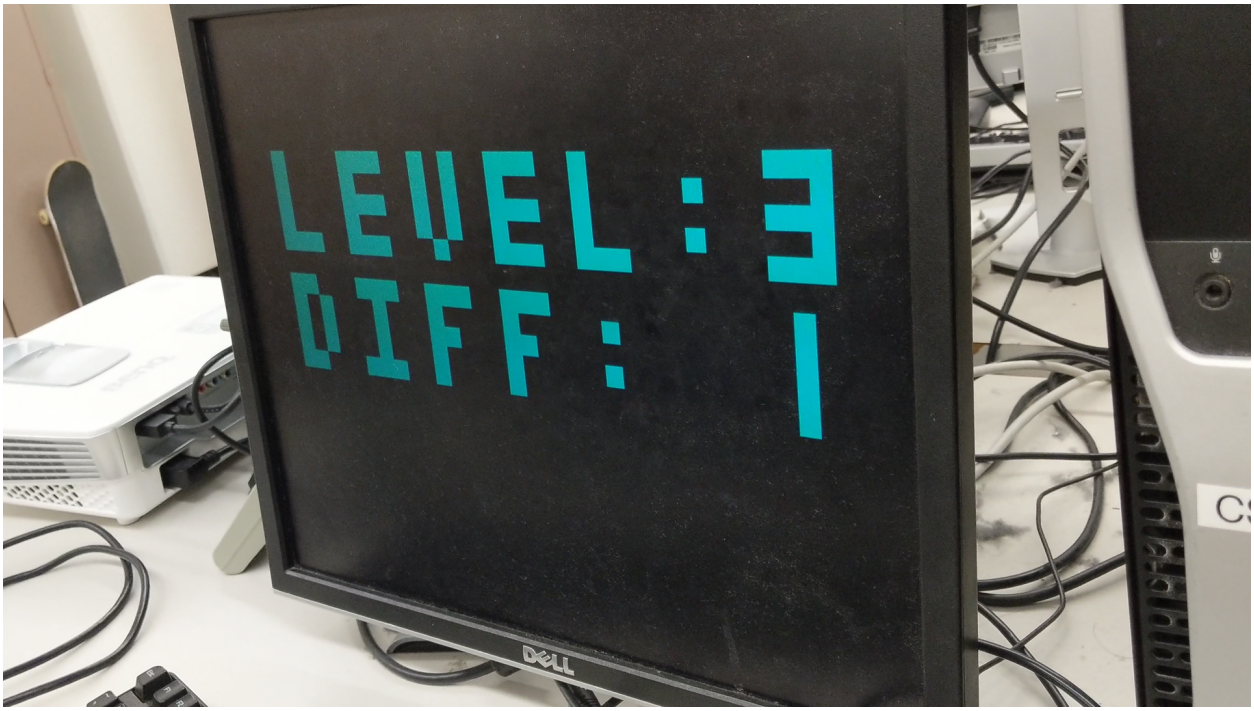

Figure 8: This figure depicts the same menu screen as in 7, but with an alternate color scheme for the level text. The level and difficulty was also different, showing that the level changing and difficulty section of the menu was working correctly.

Conclusion

Although this lab did result in a fully functional game that fit the initial proposal, several difficulties were encountered along the way that there simply was not time to create optimal solutions for in the given timeframe. As far as the sound output goes, the original goal was to utilize the Pmod I2S (which the group didn't know about until after writing the proposal) to connect to the amplified desktop speakers stored in the lab cabinet. However, numerous issues were encountered in attempts to create a working prototype project given the very limited amounts of example code existing for the peripheral. Subsequently, given Ryan's previous knowledge of how to implement a buzzer, the group instead elected to just implement a buzzer, as was originally described in the proposal.

Similarly, level data was originally intended to be programmed onto the board, hence the file names of the level modules. Issues arose from this (currently) unneeded complexity, so the group elected to hardcode the levels into the initialization blocks of each level module instead. Optimally, a random number generator could have been created as a means to implement random level generation. Although this choice would have led to the high score tracking being irrelevant, the game would have had a significantly larger amount of replay value with this feature.

Under the existing display setup, only the score is displayed using the seven-segment display. Although the multiplier was fully functional, players would not really know how the multiplier works without prior explanation given that there is no visual indicator of its existence besides score increasing in larger and larger increments. Given the existence of the Pmod seven-segment display, a better way to implement this feature for usability without entirely reformulating the VGA graphics implementation would have been to implement a separate seven-segment display dedicated to the multiplier.

Although the game was fully functional at its core, more rigorous testing of different odd scenarios could have been conducted. Occasional snags were run into in final preliminary testing with resets not fully registering and levels taking an oddly long time to load on rare occasion. However, similar to the other aforementioned possible improvements, the codebase simply had become so massive by the time the game was fully realized as initially intended that the group could have spent a week of class time alone probably tracking down tiny issues that did not really have a big impediment on core functionality of the game.

Given the relative degree of difficulty that this project ended up entailing, the group has been very much satisfied with how the game turned out. From the smooth graphics and consistent sound generation to user-defined difficulty and color scheme options, the final product that the group delivered for lab four seemed to fully meet/exceed the expectations set with the initial proposal, and the game even came out as something rather entertaining to play (as any good game should be) despite its rather simple appearance.

References

The modules for the VGA and Pmod Keypad were gathered from external sources, then modified with our own additions. The letters and numbers used for the menu were inspired by another

publicly available source, linked to below:

VGA: https://timetoexplore.net/blog/arty-fpga-vga-verilog-01

Keypad https://reference.digilentinc.com/reference/pmod/pmodkypd/start

Letters and Numbers http://www.webdiaz.com/fpga%20number.html