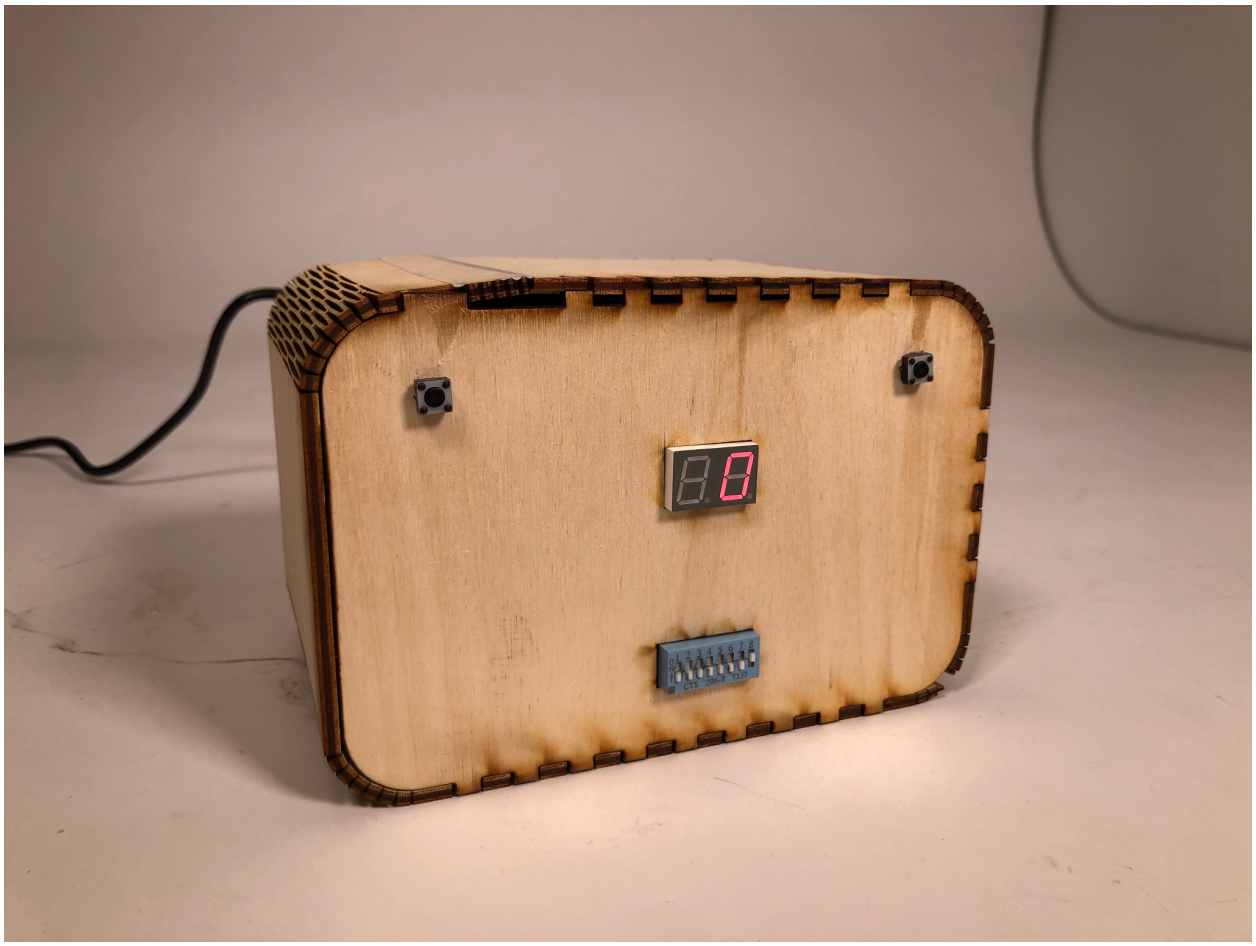
ES4 Final Project Write-Up

Daniel Butka
Mira Jain
Carla Byarugaba
Toki Nishikawa



Watch a video here
https://youtu.be/HZ9PClgV-0Y

1. Overview:

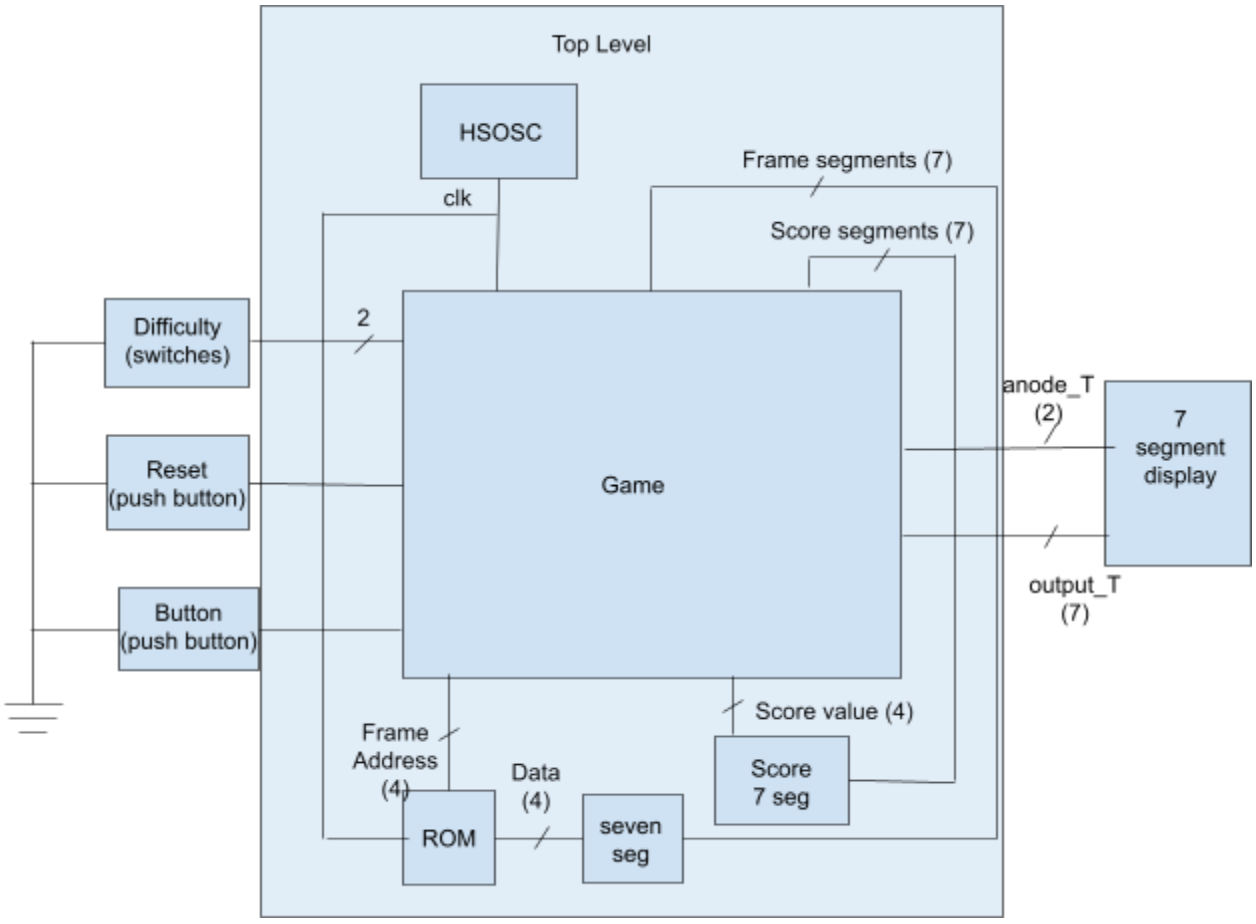
In this document, we describe the design and operation of a reaction game built using the iCE40 FPGA. The user is meant to watch a repeating animation and press a button at a specific point in the animation to score points. The animation consists of 1 segment of a 7 segment display going in a clockwise loop. The user could adjust the difficulty between four different animation speeds using two switches. We used one digit of a seven segment display to show the user a repeating animation, and the other digit to display their score. We used a push button switch to collect user input. The score is represented by a hexadecimal digit which automatically overflows from F (15) to 0. Additionally, the score can be reset to 0 at any time by pressing a different button.

The major components are shown in **Figure 1**:

- The Top Level module: (takes user's inputs, gives outputs)
    - Increments the animation frame's address at the speed of the selected difficulty
    - Stores, resets, increments the score when appropriate
    - Tells the 7 segment display how to show the animation and score
- HSOSC: creates the clock signal for the top level and ROM
- ROM: converts the current frame address into a data signal
- Sevenseg: converts the ROM's data signal into a signal that powers certain pins of the 7 segment display for the animation
- Score7seg: converts the score into a signal that powers pins of the display for the score

In reference to the above photo, everything starts running as soon as power is provided to the circuit. The left button is the one used for the game, and the right one resets the score. The two rightmost dip switches control the difficulty, and the other switches do nothing.

Figure 1: High level layout



Note: In the diagram, wires going into the top or left of a component indicate inputs of the component. Wires going out of the bottom or right of a component indicate outputs.

2. Technical description:

1. Animation Counters

The main component of our project was the animation. The animation was made up of 6 frames stored in memory and addressed with 4 bit addresses. Although we only needed 3 bit addresses for the number of frames we had, we decided to use longer addresses so that we could add more frames if necessary. To display the frames, we used four (one for each difficulty) counters in the top level module, each incrementing on each rising edge of the built in HSOSC clock. The counters only increment if the button is not pressed, causing the animation to freeze while the button is pressed. The

"easy" difficulty counter had 27 bits, and each successive counter had 1 less bit. The HSOSC clock runs at 48 MHz, so by selecting one of the 4 differently sized counters (controlled by the 2 difficulty switches), we were able to display our frames at different rates, speeding up or slowing down the animation. The frame address was the 4 most significant bits of the selected counter. The higher frequencies were generated by using smaller counters for the frame address, and the lower frequencies were generated using the larger counters. The speeds used are shown in Table 1.1. Because we only had 6 (0 to 5) frames, each counter resets to 0 when the frame address becomes 6, causing the animation to immediately restart after the last frame.

Table 1.1

Difficulty	**Bits used as address**	**Calculation**	**FPS**
Easy	26 down to 23	$\frac{48\ \text{MHz}}{2^{23}} = \frac{3 \times 2^4 \times 2^{20}\ \text{Hz}}{2^{23}} = 3(2)\ \text{Hz}$	**6**
Medium	25 down to 22	$\frac{48\ \text{MHz}}{2^{22}} = \frac{3 \times 2^4 \times 2^{20}\ \text{Hz}}{2^{22}} = 3(2^2)\ \text{Hz}$	**12**
Hard	24 down to 21	$\frac{48\ \text{MHz}}{2^{21}} = \frac{3 \times 2^4 \times 2^{20}\ \text{Hz}}{2^{21}} = 3(2^3)\ \text{Hz}$	**24**
Impossible	23 down to 20	$\frac{48\ \text{MHz}}{2^{20}} = \frac{3 \times 2^4 \times 2^{20}\ \text{Hz}}{2^{20}} = 3(2^4)\ \text{Hz}$	**48**

2. Score display / seven seg

One feature of our game is that it displays the animation and scoreboard concurrently on the same seven-segment display. We achieved this using a 19-bit counter that incremented on every rising edge of the clock. When the 19th bit of the counter is '0', the first digit is turned on, and we display the animation (by assigning the output_T signal to be the animation's 7 segment instruction). In contrast, when the bit is '1', the second digit is turned on, and we display the score

(by assigning the output_T signal to be the score's instruction). By selecting a counter size of 19 bits, the display alternates between the two digits fast enough that our eyes cannot perceive the flashing. It alternates on the display because the 19th bit is sent to one anode on the display, and the opposite of the signal is sent to the other.

3. User pressing button (rising edge detection)

Perhaps the most challenging part of the project was correctly incrementing the score with every button press. The idea was that on a clock rising edge, if the button signal is low (ie, a falling edge has occurred from somebody pressing it) while the frame address has a certain value, a counter holding the score will increment. However, we will explain later a process that must occur before the increment. To implement the button falling edge, we implemented the state machine shown in **Figure 2**. The states S1, S2, S3, and S4 are assigned the binary encodings '11', '01', '00', and '10', respectively. The variable 'b' represents the logic value being returned by the pushbutton – this value is LOW when the button is pressed, since pressing the button connects the pin to ground, and HIGH otherwise. Thus, the S1 represents the default state where the button has not been pressed. If the next 'b' input is HIGH, we stay in S1, but if it is LOW, we move to S2. In this state machine, S2 is the only state where we increment the score. As you can see, regardless of the next 'b' input, if we are currently in S2, we must leave S2 before we can return to S2 to increment the score another time. As a result, the state machine yields logic that increments the score a single time upon pressing the button. The idea for this part came from the "rising edge detector" assignment on VHDLweb, although it is more accurate to call this a falling edge detector for the button.

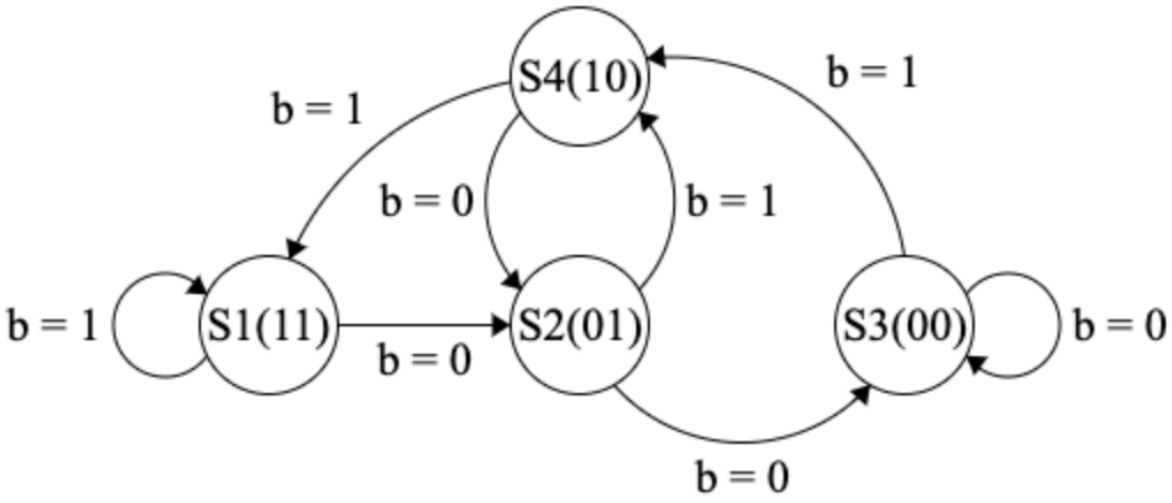


Figure 2: *State machine* used to identify a button press.

In regards to hardware, this is implemented using a shift register with two flip-flops as shown in **Figure 3.** Initially, a series of 1's are passed through the flip-flops, resembling the S1 state. Then, upon the press of the button, a '0' will be passed into the first flip-flop on the rising edge of the clock, changing the state to S2. On the next rising edge of the clock, either a '1' or another '0' will be passed in depending on whether the button is still being pressed or not. Either way, the state will change to S3 or S4, meaning that the score will not be incremented another time.

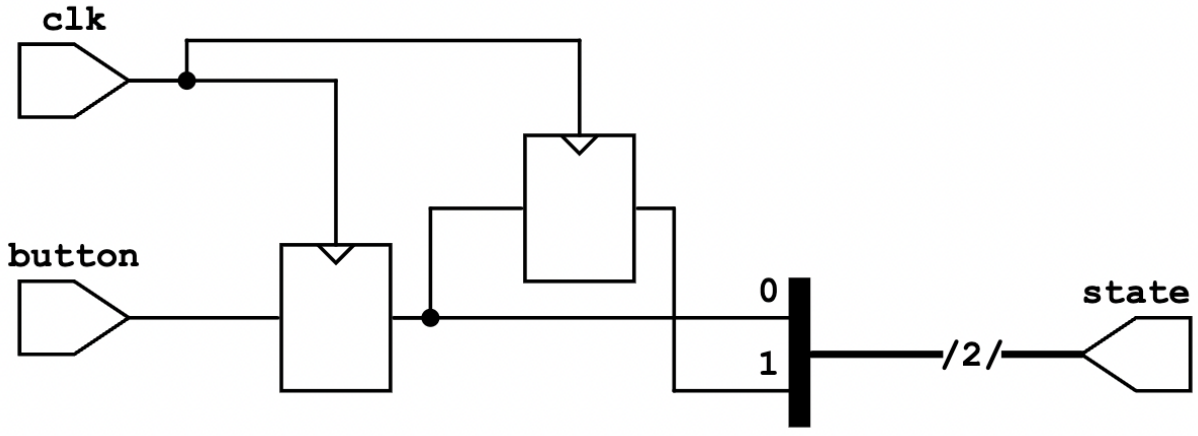


Figure 3: *Hardware* used to identify a button press.

In theory, this hardware should correctly increment the score every time the button is pressed – however, this is not what happens as hardware is seldom

perfect. Through careful analysis using an oscilloscope, we discovered that the pushbutton bounces with every push; that is, it makes multiple distinct connections with ground during the time it is pushed down. This is not ideal as we have designed a system that increments the score every distinct connection with ground. We must introduce some debouncing mechanism.

As it stands, the current design will cause the score to act unpredictably as an undefined number of ground connections will be made with every push of the button. In order to prevent this, we use a counter that will begin once the first ground connection has been detected. The counter will increment on every rising edge of the clock until a specified value has been reached, at which point the score will increment, the counter will reset, and if reentered, the S2 state will be allowed to trigger the counter again. As a result, if multiple ground connections occur after the initial ground connection, they are likely to take place while the counter is running and thus unlikely to increment the score. With that said, if the user holds down the button for long enough – ie, longer than the length of the counter – another ground connection could occur, meaning the score will increment more than once. However, this would require the user to hold down the button for an abnormally large amount of time.

One may question: why not just increase the length of the counter? Although increasing the counter reduces the odds of the score incrementing by more than a single point, it also increases the time between the user pushing the button and the score updating. Thus, lengthy counters would mean that the score may increment as long as seconds after the button has been pushed, which doesn't make for the greatest user experience. We used a counter of 24 bits, which seemed to provide a good balance between the accuracy of the scoreboard and the time it takes to update.

4. Difficulty

Table 1.2

Switch input	**Difficulty**	**FPS**	**Length of animation**
00	Easy	**6**	**1 second**
01	Medium	**12**	**0.5 seconds**
10	Hard	**24**	**0.25 seconds**

11	Impossible	**48**	**0.125 seconds**

5. Sevenseg and Score7seg

Both of these components create the 7 bit signals for the 7 segment display to directly “read”. In each signal, a ‘0’ instructs a certain segment to be on, and a ‘1’ instructs it to be off. Using a series of when else statements, each of these components outputs one of these 7 bit signals based on a 4 bit data input. For score7seg, this input was the number of the score, in binary.

We created a new module for seven seg, but the values needed for seven seg were taken from previous labs. We decided to use one digit and show the value using hexadecimal to show as many different values as possible in a limited space, so we did not change anything from that lab.

Table 1.3: Frames of the animation

Frame address	data	7 segment instruction	Lit segment
0000	0000	0111111	Topmost (A)
0001	0001	1011111	Top right (B)
0010	0010	1101111	Bottom right (C)
0011	0011	1110111	Bottom most (D)
0100	0100	1111011	Bottom left (E)
0101	0101	1111101	Top left (F)

6. ROM

Our project implemented a ROM module to store the data for the animation frames. The ROM entity has an architecture called "sim," which contains a process that specifies the behavior of the ROM.

The process waits for a rising edge on the clock signal and then decodes the value of the addr input to determine the data that should be output on the frame output. Since the address came from a counter, the address is the number of the frame in the order of the animation. The data is stored in a signal called "data," which is an unsigned 4-bit value. The ROM entity also includes a component called "sevenseg," which has two ports: an unsigned 4-bit input (S) and a 7-bit std_logic_vector output (segments).

The ROM architecture maps the "data" signal to the "S" input of the sevenseg component and the "segments" output of the sevenseg component to the "frame" output of the ROM entity. When the clock signal transitions from low to high, the process determines the value of "data" based on the value of "addr," and the value of "data" is then passed to the sevenseg component to produce the output on the "frame" output.

Since each frame of our particular animation is unique, the ROM was actually redundant. This is because the address “0001” just yields the same data signal “0001” and so on. However, it was kept in case we wanted a different animation.

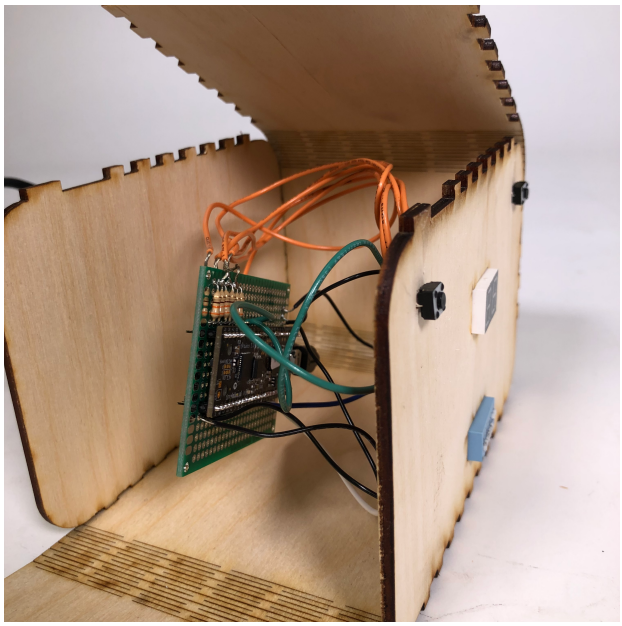
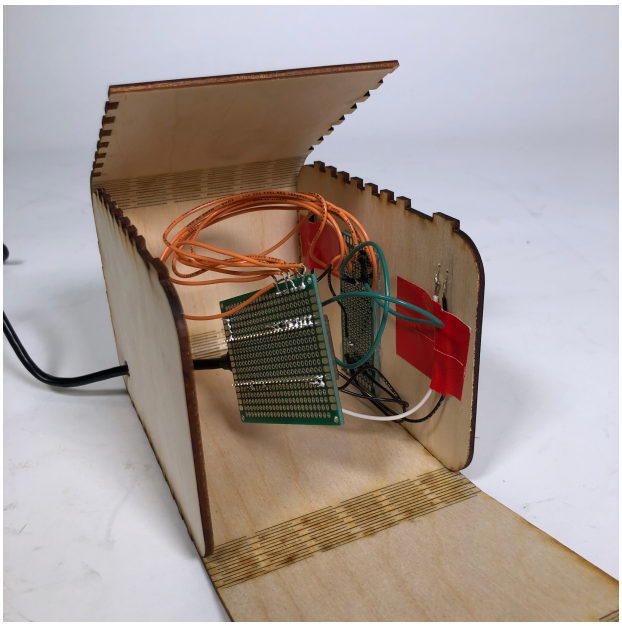
Table 1.4: Examples of displayed score

score	7 segment instruction	Displayed character
0011	0000110	3
1010	0001000	A
1011	1100000	b

3. Results:

Here are some other photos of the project:



As stated, the main bug is that the button sometimes detects more than one press when it is actually only pressed once. The solution we implemented prevents this, but not if the button is held down for too long.

Bits processed per clock cycle: 191
Frames of animation: 6
Difficulty settings: 4

If we had more time, we could have implemented different animations that the user could choose from. To do this, we could use the unused dip switches as the controls to send a signal to select the animation. This signal would be an input to the ROM, which would be a condition for mapping the address to the data signal in a different way. If we wanted an animation with a different number of total frames, we would have to handle resetting the counters differently.

Another addition to this project could be to include more seven-segment displays to create more complex animations. For example, if we had an array of seven-segment displays, we could create animations that were continuous between them, and we could use separate displays for the score so that the animations could flash between the two digits of each display.

Furthermore, we discussed the possibility of making it a two-player game. In this case, there would be multiple scoreboards and buttons (one for each competitor). They could compete on different speeds/patterns where each speed/pattern combination would play out twice. We could also count the number of incorrect button presses so that our game measures accuracy.

4. Reflection:

For this project, we built upon the work that we had completed in lab 7.

We were able to successfully implement a counter that would track the number of times a button was pressed. However, we encountered an issue where the counter would occasionally jump up by two or more when the button was pressed too quickly.

To troubleshoot this issue, we experimented with adjusting the delay between button presses and found that increasing the delay helped to reduce the frequency of the counter jumping up by two or more.

We also connected an LED to the score and noticed that the LED remained on when the score reached 29 or higher. This seemed to be an issue with the code, as the LED should only turn on when the score reaches a certain threshold.

Overall, this project provided an opportunity for us to build upon previous work and troubleshoot any issues that arose. While we were able to successfully implement the counter and LED, there were still some issues that needed to be addressed in order to improve the overall functionality of the project.

5. Work division:

Dan:

The core of the animation implementation in our game was built upon my VHDL code from lab 7, which only utilized one of the display's digits. This code included a counter that was used to increment the animation address, a case statement to implement a read-only memory (ROM), and when/else statements for the 7 segment instructions for the two sevenseg modules. In addition to this, I also implemented four counters and the difficulty selection logic. These elements were crucial for ensuring that the animation functioned correctly and that the player could customize the difficulty of the game.

In order to allow the player to interact with the game, I also implemented the button falling edge detector. This was necessary for detecting when the player had correctly pressed the button, and for triggering the appropriate response in the game.

Carla:

I took on the task of attempting to implement VHDL code from lab 7 into our game, using it as a foundation to build upon. My goal was to create an animation, a way to check if the user correctly presses the button, a scoreboard that increments every time the player scores, and an input to increase difficulty. In order to achieve this, I had to have a deep understanding of the VHDL language and be able to apply my knowledge to the specific requirements of the game.

For the scoreboard, I drew upon my work from a previous lab where we focused on creating a dual digit display. This required me to review and build upon the concepts that I had learned in that lab. The animation, on the other hand, utilized applications from Lab 7, so I had to familiarize myself with those as well. Overall, the process of implementing the VHDL code into the game was challenging but rewarding, as it allowed me to showcase my skills and knowledge in the field.

Toki:

In regards to the code, I was responsible for two sections of the top module. More specifically, I coded the mechanism that allows both digits on the seven-segment display to be on simultaneously (well, appear to be on simultaneously). Moreover, I coded the debouncing mechanism described in section 3 of the technical description. Dan and I spent many hours debugging the code and physical circuit.

Finally, I soldered everything using protoboards and laser cut a box to hide the electronics – the final product is shown in the results section.

Mira:

I worked on the top level module animation code. I made sure that the counter functioned correctly and would show the frames at the speed selected by the user. I created the code that would reset the counter at the right frame and the code that would generate the counter required for each difficulty. I also helped other members with their parts of the code.

I also coded a more complex animation to work with two digits, but we did not have a chance to utilize this in the final project because we decided to use only one seven segment display.