Rope Game Theory of Operations

Brett Thornhill, Chuck Faber, Tiffani Shilts

Maseeh College of Engineering and Computer Science

Rope Game was designed for a combined final project of

ECE 540 SOC Design with FPGAs and ECE 558 Embedded Systems Programming

with Professor Roy Kravitz at Portland State University Winter Term 2020.

Table of Contents

Abstract

The Rope Game is a video game which uses the Nexys A7 100T FPGA, an Android compatible device, and a VGA display to emulate jumping rope. The screen displays a line to symbolize the jump rope which either moves up — to signify ducking, or moves down — to signify jumping, to the beat of the music. The user “jumps" over the rope via an Android device which is secured to their waist detecting duck and jump movements. This peripheral is controlled via an app which uses the device to send accelerometer data back to the Nexys A7, as well as allowing the user to create a profile, and record their high score. During gameplay, the score is displayed to the user via the seven segment display located on the FPGA.

Motivation

We chose to develop this game because some of our members enjoy playing rhythm based games, and this seemed like a fun option to implement actual movement into a typical rhythm based game. Furthermore, with COVID restrictions we felt that many people are probably not getting enough exercise and activity since most gyms are restricted and fewer people are leaving their home. We think this project allows people to stay active while remaining in the comfort and safety of their home, and still have fun with the game and rhythm aspects of this project.

Our other goals included learning more about using the Bluetooth Low Energy protocol for wireless communication, hardware and SoC development with FPGAs, and developing dynamic Android apps making use of the features and hardware present in modern day smartphones for a new, interesting purpose.

Methods

A. Hardware

a. Overview

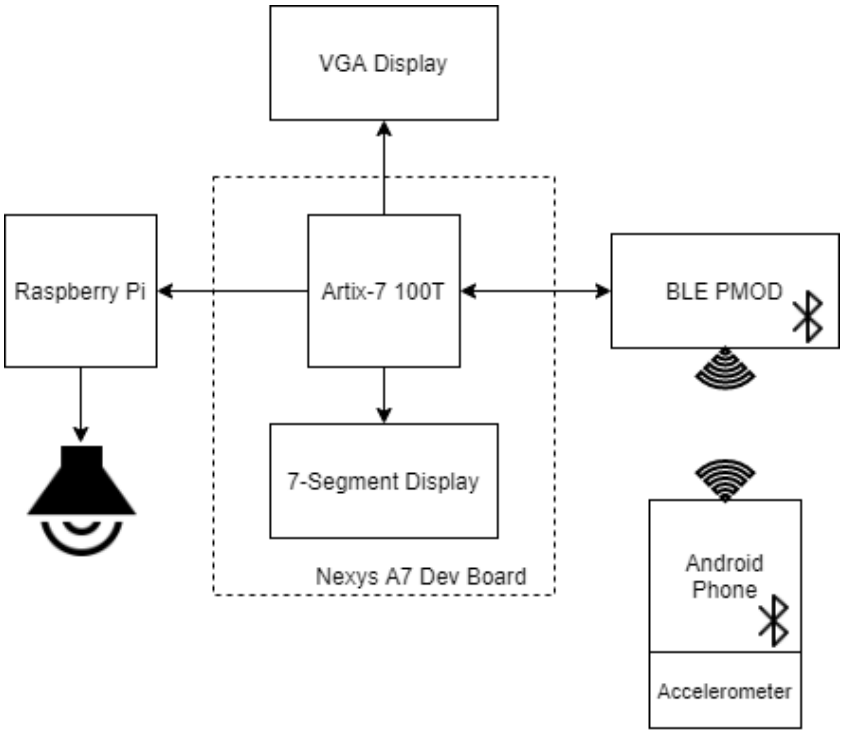

Fig. 1 Demo Implemented Hardware

When demonstrating our project, our hardware was configured as above in figure 1. We used the 7-segment display to display the score, and we connected the board to a VGA display for the gameplay cues, a Bluetooth Low Energy PMOD to communicate wirelessly with our Android app, and also we connected a Raspberry Pi to the board to act as our music player. The Raspberry Pi configuration was in accordance with our contingency plan should on-board audio not be implemented in time for the demo which we understood to be a risk considering the complexity

of implementing audio with the Nexys A7. Otherwise, our original planned configuration was as below in figure 2.

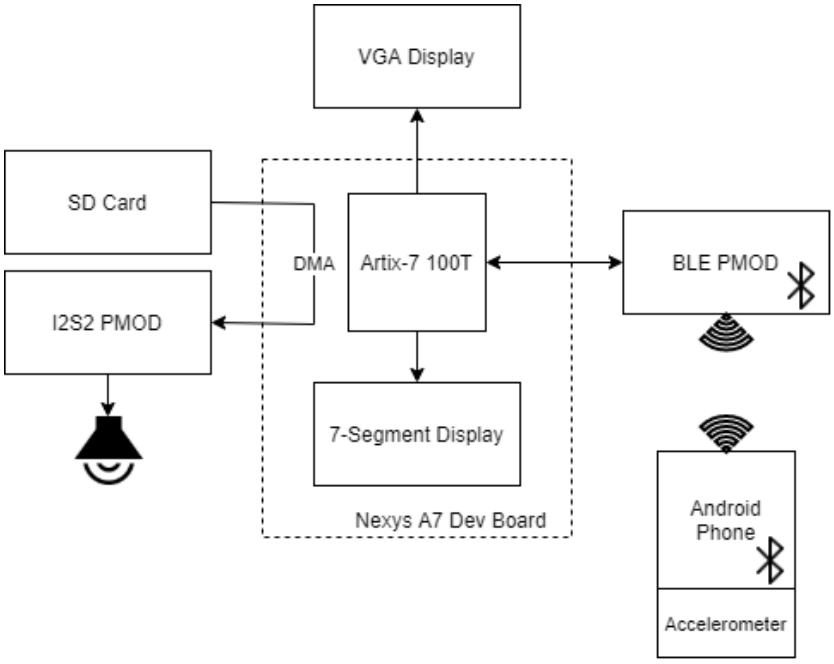

Fig. 2 Planned HW Implementation

In the configuration above, to implement on-board audio, we would have to create the HDL modules required to implement the I2S2 PMOD, and an SD card to read music from. This would be in anticipation of implementing multiple songs that the player could choose from. We imagined that the audio data would have to be streamed through a DMA of some sort as the on-board memory is limited.

b. HDL Organization

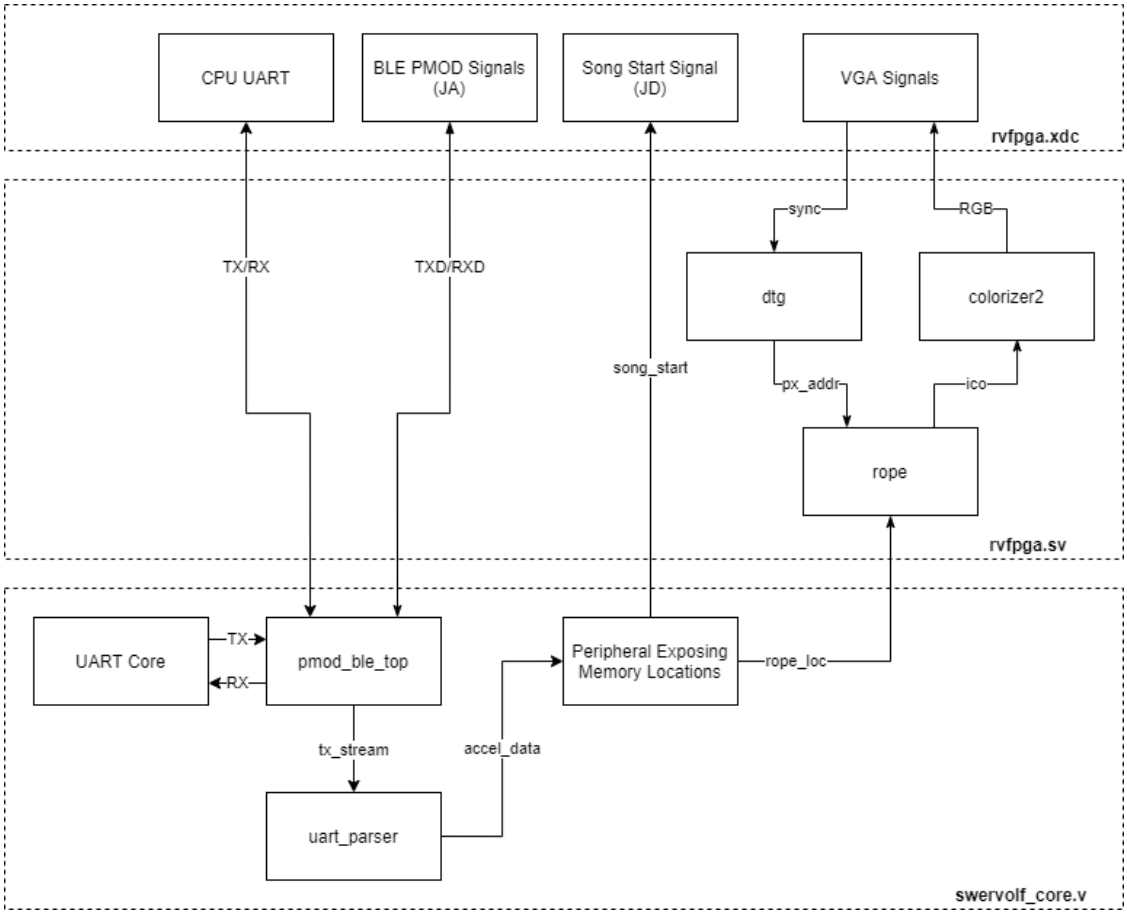

Fig. 3 HDL Module Connection Flowchart

Our HDL modules were organized as depicted in figure 3 above. Starting from the physical constraints level, we have our PMOD connected to Junction A on the Nexys A7 board. The JA pins as well as the UART pins from the USB connected are routed through rvfpga.sv and down into swervolf_core to a custom written pmod_ble module. This module is also connected to the existing UART core module already instantiated in swervolf_core to allow software initiated UART streams to send commands to the BLE pmod. We can toggle between sending UART commands via the UART core module, and the direct UART pins on the board though switch 1.

When the BLE PMOD receives data, it passes it to a series of custom modules (some borrowed from ECE 351 and some newly implemented as state machines) for the purpose of converting the UART stream into received bytes, and parsing that stream of bytes for the acceleration data to place it into a custom peripheral that exposes several registers to the software. One of these registers is for the acceleration values, so as the PMOD receives new acceleration values, the software can read them at any time. It also exposes two software-writable registers: the song-start register, and the rope-location register.

Writing a 1 to the song-start register sends a signal across one of the Junction D pins to the Raspberry Pi telling it to begin playback of the song for the game. Writing to the rope-location register sets the position of the rope vertically on the VGA screen.

c. BLE PMOD

The Bluetooth Low Energy PMOD has an RN4871 chip This chip has a set of pre-programmed UART commands that it responds to, and the chip itself has many features. It can act as either server or client, it can store scripts. For our purposes we ran the PMOD as a BLE client and central, communicating with our Android app which acted as the server and peripheral. We developed software that allowed us to send UART commands programmatically rather than having to manually enter them in to initiate connections, set the device up as a client, and in some cases initialize the transmit descriptor. However, Android security measures made it difficult for us to automate initiating the connection programmatically, though we were able to get this working with the ESP32 test module.

d. VGA Display

Much of the VGA implementation in the game is the same code used in the RoJo-Bot project, with the removal and addition of a few modules in order to simplify the functionality. The World_Map, Icon, and Scale modules have been replaced with a more streamlined module named Rope. The Rope module takes input from the software via the memory mapped i/o to determine the location of the rope, and provides information to the Colorizer module to move the line up, or down the screen accordingly.

e. Audio

i. Raspberry Pi Peripheral

In order to toggle the signal for the audio on/off, a bit was added to the wishbone bus which could be set via mmio. This signal is attached to pin JD1. When the bit is set high by the software, the RaspPi begins running its script to start playing music.

ii. PWM Signal

A PWM signal generating module for mono audio was implemented into the hardware, but not utilized in the final design. A new peripheral was added to the wishbone bus in order to control the data sent to the signal via the software. It was then instantiated in swervolf_core, and the pwm signal was passed up to the RVFPGA module directly to the pwm pin on the board. The module uses hardware interrupts to signal when it is ready for a new sample, and we were unable to implement those in time for verification. The module is capable of producing tones through the mono audio auxiliary port.

iii. I2S2 PMOD

The I2S2 PMOD was implemented as an audio passthrough module which takes audio information in through an audio auxiliary port, and outputs it as stereo audio. The audio's volume is regulated via a volume control module which uses switches one through four on the Nexys A7 board. Similar to the RaspPi implementation, the module contains a reset bit which can be toggled on via mmio.

B. Software

a. Nexys A7 Software

i. Main Function

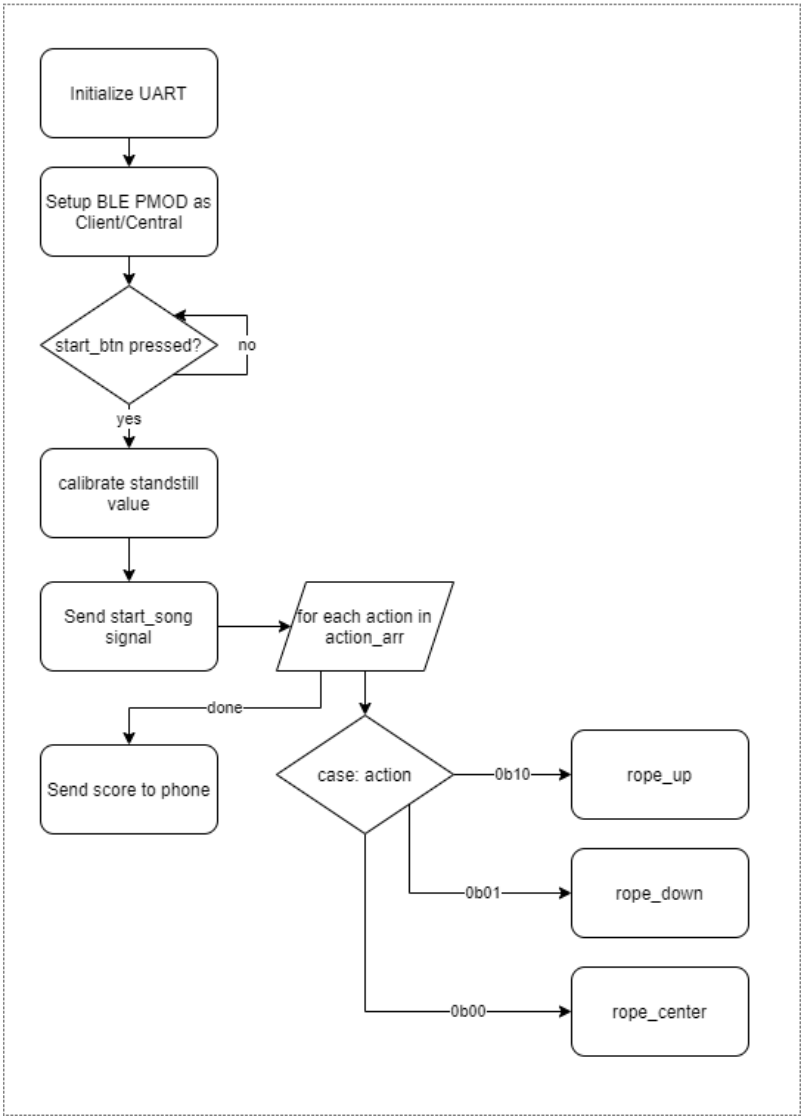

Fig. 4 Main Function Flow Chart

The main function of the software of our game was a fairly simple loop, whose main function is, upon receipt of the start signal, get a baseline acceleration to base the motion measurements

from, start the song playback and iterate through an array which defines the required player actions for the level. Depending on the value read from the array, one of three functions is called: rope_up, rope_down, or rope_center. This corresponds to the player actions: duck, jump, and do nothing respectively. An example of the game play array can be seen in figure 5 below.

```
char transition[433] = {

    0b00,
    0b00,0b00,0b00,0b00,

    0b01,0b00,0b00,0b00,    0b00,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b00,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b00,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b00,0b00,0b00,0b00,

    0b01,0b00,0b00,0b00,    0b00,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b00,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b00,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b00,0b00,0b00,0b00,

    /*Picking up*/
    0b01,0b00,0b00,0b00,    0b01,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b01,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b01,0b00,0b00,0b00,
    0b01,0b00,0b00,0b00,    0b01,0b00,0b00,0b00,
```

Fig. 5 Example Game Action Array

Values of 0b00 are defined as rope_center (stay still) actions, 0b01 are rope_down (jump) actions, and 0b10 are rope_up (duck) actions. There is one element in the array per each beat of the song, so the actions match up to the song playback.

ii. Jump Function

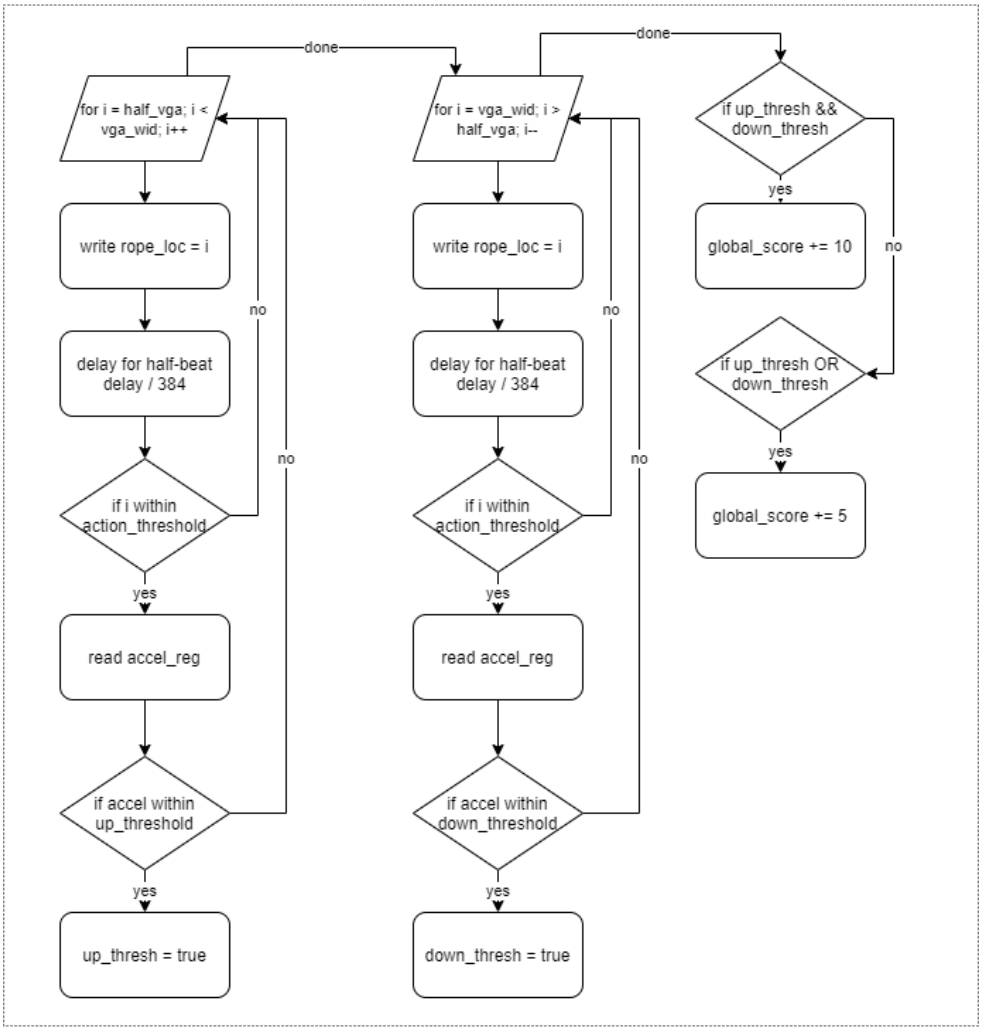

Fig. 6 Rope Down Example Function Flow Chart

The main work of the game was done in the rope functions. A high level depiction of the rope_down function can be found in figure 6 above. It can be broken down into three major parts. Two for loops, and then a threshold check to assign points.

The first for loop moves the rope from the center of the screen to the bottom. There is a window in which it starts to read the acceleration value from the exposed acceleration register in hardware which is receiving the acceleration values via the BLE pmod. If any read acceleration

value meets the threshold for awarding points in this window, a boolean is set true. The same check occurs as the rope makes its way back up. After both for loops have concluded (actually a little before to speed up the awarding of points), a check is done on both booleans. If both are true, the game assumes the jump was well timed and awards full points. If only one is true, the game assumes the jump was present but mistimed and only half points are awarded. If neither are true, we assume the jump did not happen and no points are awarded.

The score function writes the current global score to the 7-segment display including the newly awarded points.

As shown in the diagram, the delays are implemented in these action functions as well, making these crucial for synchronization of the actions to the song based on the BPM of the song. The delays are calculated using the BPM so that the game is adaptable to songs of different BPM values, so long as the BPM does not change mid-song. The calculations done to calculate the delay values can be seen below in figure 7.

```
const double bpm = 125.0;                              // beats per minute
const double bps = bpm / 60.0;                         // beats per second
const double spb = 1 / bps;                            // seconds per beat
const unsigned long int uspb = spb * 1e6;              // us per beat
const unsigned long int hbd = uspb / 2;                // Half beat delay in us
const unsigned long int rdl = (hbd / HALF_VGA_WIDTH)
    + COARSE_CORRECT;                                  // rope up/down movement delay
                                                       // for half screen movement
unsigned long int cdl = (hbd / 2) / (CENT_DISP)        // rope center movement delay
    + (COARSE_CORRECT * VGA_CENT_RATIO);
```

Fig. 7 BPM calculations for rope delays

b. Raspberry Pi Software

The Raspberry Pi script running during the demo was a simple script which reads the input from a GPIO pin in a while loop, and when it goes high, it will start the song. Below you can find the main loop of the Raspberry Pi script:

```
# Main script body
while True:
    if (GPIO.input(4)):
        print("Playing Transition.wav")
        pygame.mixer.music.play()
        while pygame.mixer.music.get_busy() == True:
            continue
```

Fig. 8 Raspberry Pi Code

Some initialization was also involved in ensuring that the WAV file was sampled at the appropriate frequency to prevent the song playback being too slow or too fast.

C. Android Application

a. Overview

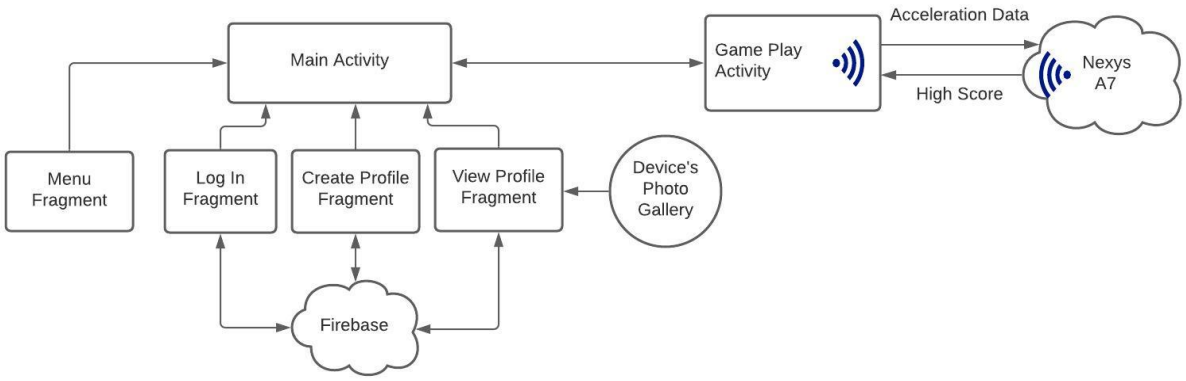

Fig. 9 Android App Layout

The Android application manages two central elements of the game; sending peripheral data back and forth to the BLE PMOD, and the creation of user profiles for tracking player statistics. It accomplishes this by using two activities — the Main Activity, which uses Firebase to manage user activity, and the Gameplay Activity, which manages Bluetooth Low Energy reads and writes. The Main Activity is supported by fragments, which easily allows the user to view the menu, and create or manage a profile in both portrait or landscape mode without having to include any supporting activities.

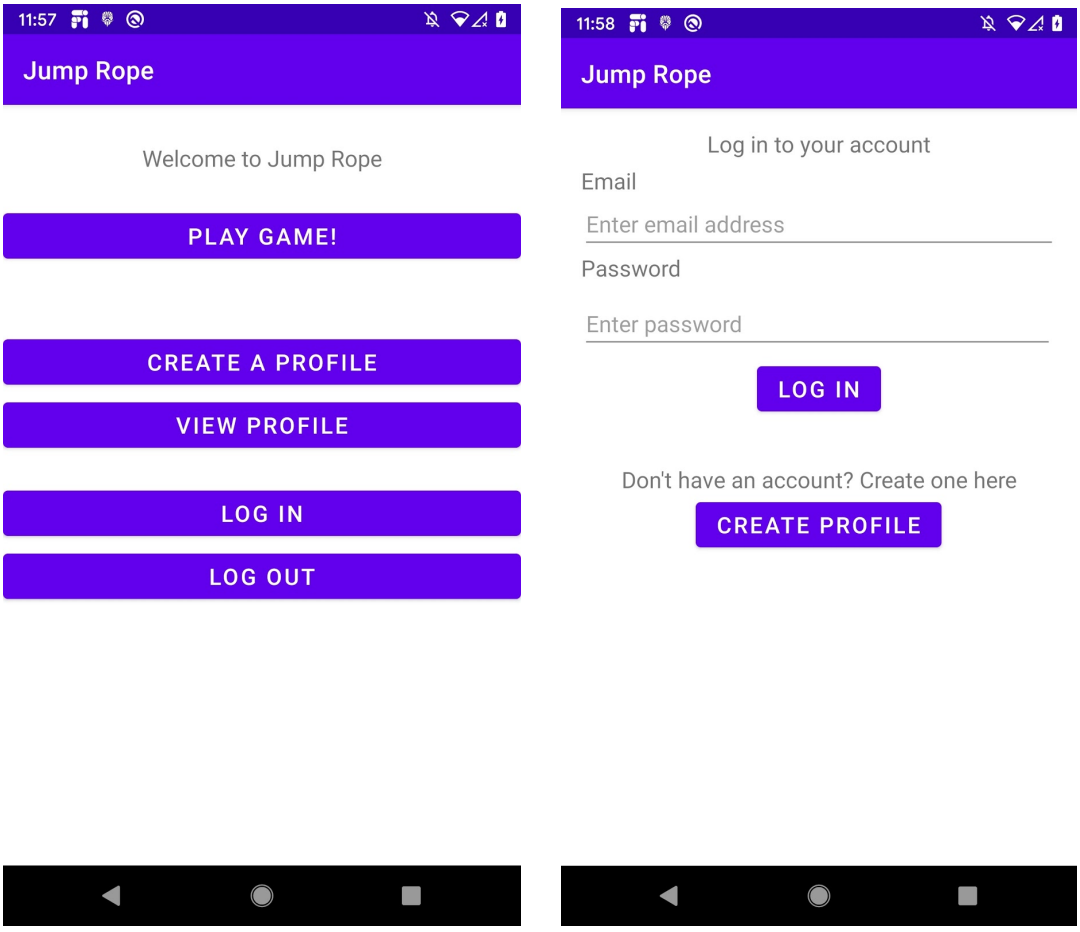

Fig. 10 Main Activity Layout and Login Screen

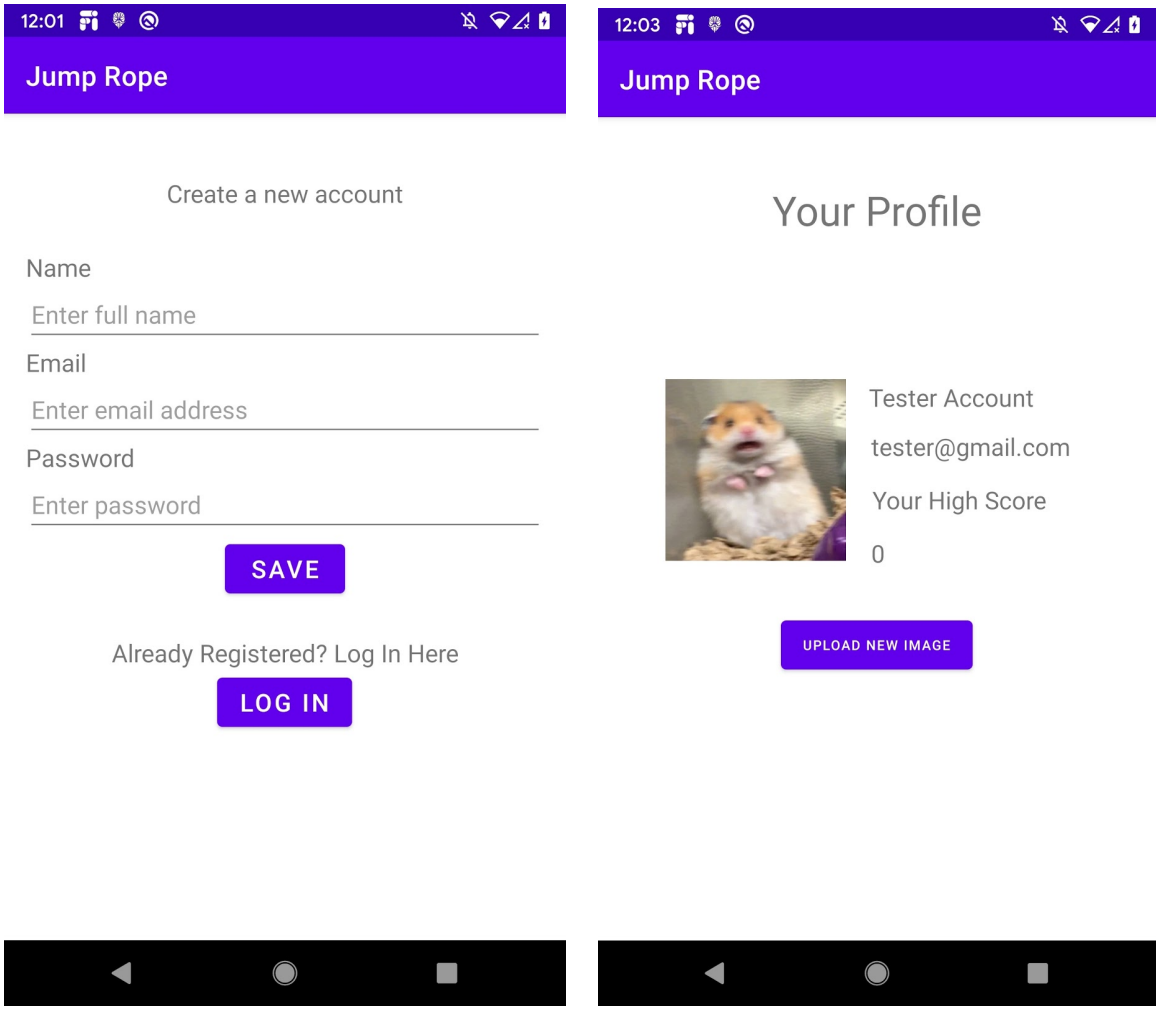

Fig. 11 Account creation screen and User Profile view

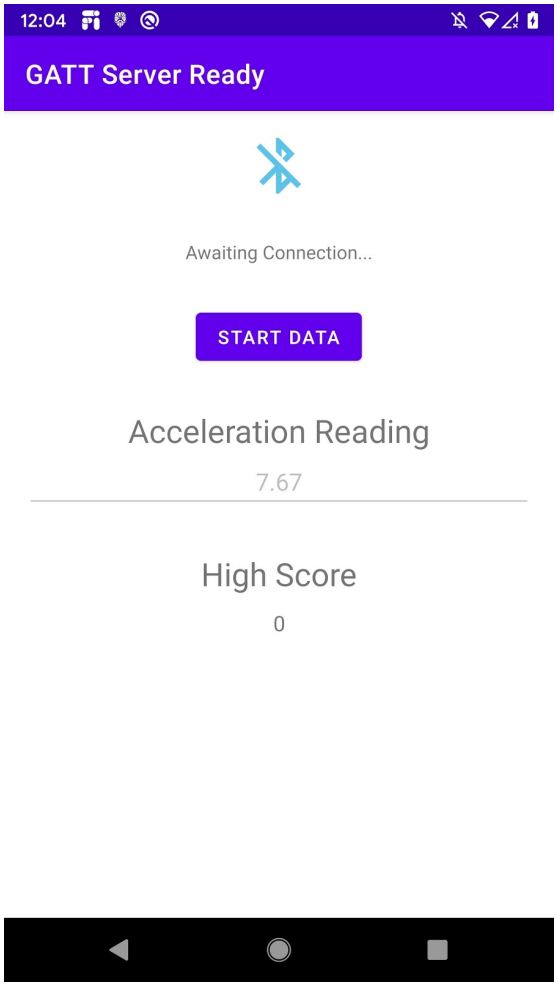

Fig. 12 Gameplay Activity

b. Main Activity

The main activity hosts the user profile fragments and manages the fragment display. It implements the OnMenuSelectionListener interface which allows the fragments to communicate back to the activity. In OnCreate() the activity determines if the display is 600dp or larger by reading the layout id. If the display is small, it hosts one fragment container and if it is larger, it hosts two fragment containers. In onMenuSelection() the activity responds to strings sent from the fragments and adjusts which fragment is currently being displayed.

c. Gameplay Activity

The game play activity manages the bluetooth low energy and reading the device's acceleration data. The acceleration data is read by implementing the SensorEventListener that updates when the value changes. In OnCreate(), the user's Firebase ID is accessed to read the high score saved for that user and is displayed on the screen. If the user is not logged in, the display goes back to the main activity and a toast prompts the user to log in first. The device is checked to ensure it is bluetooth capable and able to advertise. If it passes those checks a GATT service is created and set up for read and write transactions then it starts to advertise. There is a button that when pushed toggles the ifSendData variable allowing the user to control when the acceleration data is being sent via bluetooth. When the song is done and the game is finished the BLE PMOD will send the score and the gameplay activity will compare this number to the saved high score from the user's profile. If the score returned is higher then the number saved in the user's Firebase account, the score is then updated with the new value.

d. Device Profile Class

This class stores the UUIDs needed for bluetooth communication and has several helpful functions for converting to and from bytes.

e. Fragments

i. Menu

The menu fragment is a list of buttons that communicate with the main activity via the OnMenuSelectionListener interface.

ii. Login

The login fragment checks in OnCreate() if the user is already logged into Firebase and returns the user back to the menu with a toast to indicate if the user is already logged into an account. If the user is not logged in, it takes in the email address and password from the user and authenticates with Firebase. If the account does not exist or the user entered incorrect data a toast will pop up saying “Authentication failed”.

iii. Create Profile

The create profile fragment checks in OnCreate() if the user is already logged into Firebase and returns the user back to the menu with a toast to indicate if the user is already logged into an account. If the user is not logged in, it takes the inputted name, email address, and password and creates a Firebase document reference and uploads the information.

iv. View Profile

The view profile fragment checks in OnCreate() if the user is already logged into Firebase and returns the user back to the menu with a toast to indicate if the user is not logged into an account. The user id is fetched from Firebase and the associated photo, name, email, and high score are displayed. There is a button that accesses the device’s gallery which allows the user to pick a photo for their account. This photo is resized using Picasso and is uploaded to Firebase storage. The storage is connected to the unique user id so only one photo is stored per account.

Challenges

Throughout the development of the game, we faced several difficulties such as interrupts, bluetooth connectivity, and fragment implementation.

The PWM module which was developed for the mono audio relies on hardware interrupts to indicate when the next sample is required. In implementing the hardware interrupts, we tried to closely emulate the unused gpio interrupts in the Swervolf Core. Unfortunately, while we got the module to successfully trigger the interrupt, we struggled with getting the interrupt bit to reset once being set. With more time to study the Swervolf Core we could have overcome this, but as time on the project was short, we switched focus to a contingency plan. The synchronizing of the audio with the game also turned out to be more complicated than initially anticipated, and took advanced JTAG verification.

The Android app was originally designed with activities for each of the different views. The activities worked but we wanted to showcase more of our Android skills so we refactored the user profile activities into one main activity that dynamically hosted fragments. It was challenging to convert the design and it became apparent why Big Nerd Ranch recommends starting with fragments at the start. The biggest issue was when the screen rotated it was difficult to manage the fragment stack. A future improvement would be to implement more checks during rotation and to manage the back button more efficiently.

Another challenge was working with the BLE PMOD. There was very little documentation regarding it, and getting it to work took a lot of trial and error. The BLE protocol is complex and the specific steps that needed to be taken with the RN4871 weren't always clear. In addition, we struggled a little bit with getting the motions of the rope to sync well with the song. By doing some creative debugging using an oscilloscope, and triggering a junction pin high and low at the start and end of each function we found that our three rope movement functions didn't all take the same amount of time. The center movement function was twice as long as the other two.

Making some adjustments after figuring this out made the software behave how we'd expected it to.

Lastly, the UART parsing modules were another challenge. The Nexys A7 board is a bit of a black box when it comes to figuring out exactly what is happening between each module. Even though our simulations of our modules were working perfectly when moved to the actual hardware, things weren't behaving as expected. It took a full day to learn how to use the ILA tools to debug exactly what was occurring in the hardware, but it was worth it and helped us get things working.

Conclusion

The completion of this project allowed us to apply, and fully conceptualize the most important concepts we have learned throughout the term. Integrating PMOD modules into the RVFPGA hardware tested our understanding of the Swervolf Core, and the Wishbone bus. The rapid development of these modules demonstrated the fundamentals of adapting an FPGA to suit the needs of a project. Using an Android device as a peripheral allowed us to expand, and refine the app development process we learned by applying the important concept of fragments. Integrating Bluetooth and Firebase into the app gave us real world experience in using the Android development support material and community. Overall, the process of building this game was a valuable experience in both FPGA, and Android Application development.