# Swish Slash Sword Swinging Simulator

Final Design Report

ECE532 - Digital Systems Design

Gavin Gu

Harry Hopman

Nicholas Tran

Samuel Zheng

# Table of Contents

# 1.0 Overview

## 1.1 Background and Motivation

Inspired by the Legend of Zelda Skyward Sword, we aimed to implement a motion-control based sword-fighting game similar to how the Wii remote could replicate sword-swinging motions [1]. This was motivated by recent trends in game design focused on improving player immersion. This can be achieved through many means, such as motion controls, haptic feedback, virtual reality, and more. We hoped to explore this field of immersive gaming in our project, leveraging the I/O capabilities of FPGAs.

Figure 1: Motion-control sword-swinging gameplay from the Legend of Zelda Skyward Sword [1]

## 1.2 Project Goal

The goal of the project was to create a fun and immersive motion-control-based game on the FPGA, utilizing its I/O capabilities to provide various ways for player engagement with the game. The main input is an accelerometer attached to a fake sword, through which players would be able to input direction-oriented swings and blocks. Feedback to the player is done through four main outputs: 1) visual VGA display, 2) audio output to play sound effects, 3) 7-segment display to show the score, and 4) an off-board, Bluetooth Arduino system controlling vibration motors attached physically to the player to simulate the player getting hit. Additionally, a menu system will be integrated and controlled via USB keyboard input.

## 1.3 Block Diagram

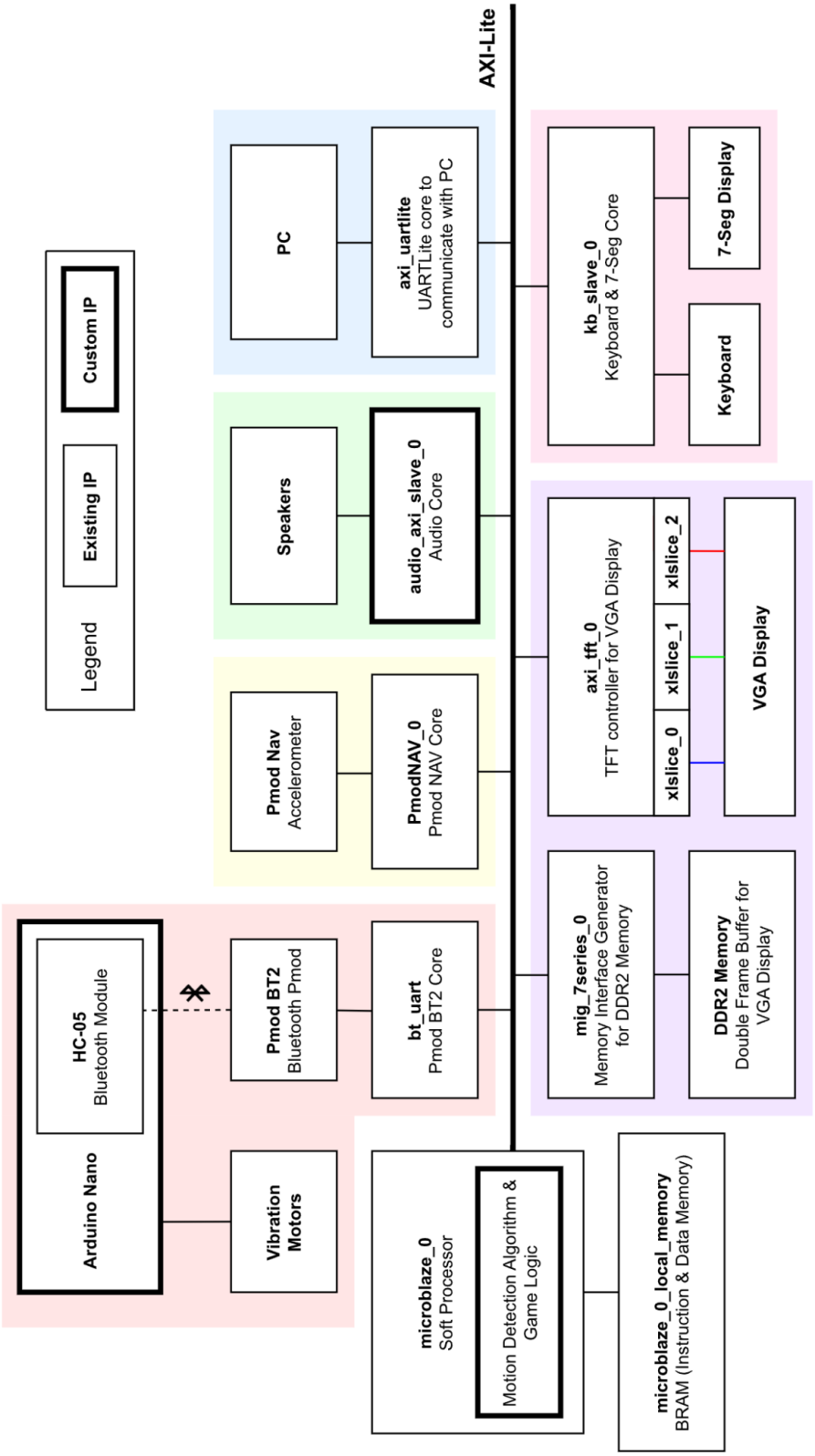



Figure 2: Block Diagram (High-Quality SVG File)

## 1.4 Brief Description of IPs

All components are organized into three categories, listed in alphabetical order within. *Project-Specific IPs* contain IPs that are either custom IPs or IPs used in specific contexts for this project. *Standard Project IPs* are generic IPs associated with most FPGA projects.

| IP Name | Description | Source |
|---|---|---|
| **Project-Specific IPs** | | |
| audio_axi_slave_0 | Generates music and sound effects according to the current game state. Plays encoded PWM waves that correspond to specific musical notes. | Custom |
| axi_tft_0<br>- xlslice_0,1,2 | Handles VGA timing signals, streams data from memory address to the VGA display. X_slices truncate 6-bit R,G,B channels to 4-bit to match Nexys DDR VGA spec | Xilinx |
| bt_uart | UARTLite connection to communicate between PMOD BT2 and the FPGA | Xilinx |
| kb_slave_0 | Debounces and processes incoming input from the keyboard before presenting it on the AXI to be used by the software.<br><br>Additionally, handles the 7-segment display, displaying the current game score and last pressed keyboard keycode. | Diligent (Modified) |
| microblaze_0 | Soft processor that holds game logic, motion detection algorithm and library of drawing functions. | Xilinx/ Custom |
| PmodNAV_0 | Continuously polls data from Pmod NAV and presents an interface compatible with AXI. | Diligent |
| **Standard Project IPs** | | |
| axi_uartlite_0 | UARTLite connection to communicate with PC. | Xilinx |
| clk_wiz_1 | Clock generator. | Xilinx |
| mdm_1 | Microblaze debugging module. | Xilinx |
| microblaze_0_axi_intc | Microblaze interrupt handler. | Xilinx |
| microblaze_0_axi_periph | Microblaze AXI bus; handles AXI communication between the MicroBlaze and connected IPs. | Xilinx |

| | | |
|---|---|---|
| microblaze_0_axi_local_memory | BRAM; houses memory-mapped AXI memory used for communication between Microblaze and IPs. | Xilinx |
| microblaze_0_xlconcat | Concatenates UART interrupt signals sent to the MicroBlaze | Xilinx |
| mig_7series_0 | Memory interface generator; controller for DDR2 Memory. | Xilinx |
| rst_clk_wize_1_100M | Soft IP that handles reset conditions for the FPGA. | Xilinx |
| reset_mig_7series_0_81M | Soft IP that handles reset conditions for the external memory interface generator. | Xilinx |
| **External Interfaces & Peripherals** | | |
| Arduino Nano<br>- HC-05<br>- Vibration Motors | Arduino Nano receives string data from FPGA through the HC-05 Bluetooth module to initiate five vibration motors to vibrate in a horizontal or vertical line. | Custom |
| DDR2 Memory | Memory holding the frame buffers the TFT AXI Controller streams from/to the VGA display; double buffered. | Xilinx |
| PC | Connected to FPGA via UART interface; used for board programing and debugging. | - |
| Pmod BT2 | Connects to FPGA via UART interface; used as master and auto-connects to HC-05 Bluetooth module slave address | Diligent |
| Pmod NAV | Connects to FPGA via SPI interface; 9-axis IMU used to measure applied acceleration to the “controller” (foam sword) | Diligent |
| Speakers | Connected to FPGA via 3.5mm audio jack; plays game music and sound effects according to current game state. | - |
| USB Keyboard | Connected to FPGA via USB; Controls menu, selects game difficulty, and restarts game. | - |
| VGA Display | Connected to FPGA via VGA; Displays game interface. | - |
| 7-Segment Display | On-board I/O; displays score and keyboard debug. | Diligent |

# 2.0 Outcome

## 2.1 Results

The game was completed meeting all the functions that were proposed as well as some extra. This included an extra “Hard mode” difficulty, live G tracking, as well as directional haptic feedback. The game works quite well, after fixing some of the stability issues we were facing around the time of milestone 5. With the keyboard integration, we have a smooth gameplay experience as the player can reset the game after completing one round without needing to touch the FPGA. The motion detection is very accurate and the VGA output works without any visual glitches, resulting in an immersive gameplay experience.

## 2.2 Possible Improvements

Most of the areas of improvement that we found would be in new features that we thought of but did not have time to implement. We were considering adding diagonal slashes and blocks, however due to time constraints and the need to add new sprites to support them, this feature was skipped. Furthermore, more fluid animations would be another area to improve. In its current form, the animations are a single frame showing if the player completed the slash or block. If we were to start over, we likely would not do much differently. Though we faced a few setbacks, there was nothing major and the project was actually quite smooth. The biggest issue was the need to switch from the Nexys Video board to the DDR4 board but porting our project was resolved within a week.

# 3.0 Project Schedule

In the following, the tables show initial milestone goals along with actual accomplished progress.

## 3.1 Milestone 1 (Jan 29 - Feb 2)

| **Milestone Goals:** | **Actual Progress:** | **Team Member** |
|---|---|---|
| Acquiring External Components (Completed) | ● Obtained vibration motors, resistors, transistors, breadboard, jumper wires, and an Arduino Nano for our haptic feedback circuit | Gavin Harry |
| Researching plan for | ● Reviewed Pmod NAV datasheet on the Diligent | Gavin |

| | | |
|---|---|---|
| implementing Pmod NAV (Completed) | website and extracted IP block into a project<br>● Studied the SDK source code that came with IP | |
| Researching plan for implementing Bluetooth communication (Completed) | ● Researched resources on the Pmod BT-2 from Diligent website and forums.<br>● Researched compatible serial Bluetooth modules for Arduino, decided on HC-05 due to easily accessible documentation | Gavin |
| High-level draft algorithm for detecting motion on hardware (Completed) | ● Studied motion detection presentations on Wii remotes<br>● Drafted high-level algorithm to process Pmod NAV data | Harry |
| Researching plan for implementing HDMI Output (Completed) | ● Research demos and tutorials on HDMI displays to understand necessary blocks<br>● Preliminary planning of control + data flow | Nick |
| Integrate onboard audio output on Nexys 4 DDR (Completed) | ● Created custom hardware IP blocks to produce notes of any frequency through square waves | Sam |

In our first week, all goals were reached. We only had one Nexys 4 DDR which Sam used to integrate audio, so the rest of the time was primarily spent researching and acquiring necessary components.

## 3.2 Milestone 2 (Feb 3 - Feb 9)

| **Milestone Goals:** | **Actual Progress:** | **Team Member** |
|---|---|---|
| Integrating and testing Pmod NAV with FPGA (Completed) | ● Created Microblaze project with constraint file for Pmod NAV and tested SDK demo code to verify sensor data accuracy | Gavin |
| Testing hardware algorithm for motion detection (Completed) | ● Implemented orientation detection algorithm in software for vertical and horizontal blocks.<br>● Adjusted threshold on accelerometer to detect swing motion | Harry |
| Integrate HDMI output to display blocks (Completed) | ● Tested HDMI Demo on Nexys Video - Only able to display on older Vivado version 2016.4 | Nick |

| | | |
|---|---|---|
| | ● Researched potential VGA interface controller | |
| Play Musical Notes (Partially Completed) | ● Researched differences between audio output on Nexys Video vs Nexys 4 DDR<br>● Audio generation requires I2C master communicating with audio codec and software controlling said I2C connection | Sam |

In our second week, our team achieved goals pertaining to testing the Pmod NAV and testing it for motion detection in forms of blocks. However, we faced challenges with HDMI and generating musical notes where we discovered that working HDMI demos only worked on older versions of Vivado and that audio output on the Nexys Video is much more complex than the Nexys 4 DDR. Made a decision to switch back to the Nexys 4 DDR and use VGA instead to avoid uncertainties pertaining to using an older version of Vivado and because we already had working audio on the Nexys 4 DDR.

## 3.3 Milestone 3 (Feb 10 - Feb 16)

| **Milestone Goals:** | **Actual Progress:** | **Team Member** |
|---|---|---|
| Acquire additional components (Completed) | ● Acquired HC-05 - Bluetooth module for Arduino | Gavin |
| Integrate Bluetooth communication (Completed) | ● Created a separate Microblaze project integrating Pmod BT2 with a constraint file through a UARTLite IP block.<br>● Verified BT2 communication using a serial emulator on phone to send and receive data<br>● Researched Bluetooth setup for HC-05 | Gavin |
| Optimize motion detection algorithm and draft algorithm for game loop in Microblaze (Completed) | ● Added a FIFO queue to existing algorithm to buffer accelerometer data<br>● Modified motion-detection to sense vertical or horizontal swings from accelerometer<br>● Brainstormed game loop algorithm | Harry |

| | | |
|---|---|---|
| Draw basic shapes on VGA<br>Add Double Buffering<br>Try drawing sprites<br>(Completed) | • Set up VGA controller through AXI TFT using a single DRAM buffer<br>• Draw basic shapes<br>• Started adding enemy sprite | Nick |
| Play Musical Notes<br>(Completed) | • Created audio core and related hardware blocks to play menu sounds, game effect sounds, and a victory tune through PWM signals | Sam |
| Integrate Keyboard communication (Completed) | • Created and tested separate keyboard module to communicate with other blocks | Sam |
| Display meaningful information on screen (Incomplete) | • As a team, decided to revisit later (G-tracker) once priority core features are polished | All |

In our third week, our team achieved most goals and also finished our previous goal of playing musical notes from the previous milestone. We also modified the motion detection Microblaze project to perform on the Nexys 4 DDR from the Nexys Video. However, we decided to reschedule our goal to output peripheral information on the GUI to Milestone 6 as we wanted to focus more time and effort towards integrating the project together and leave ample time for troubleshooting.

## 3.4 Mid-Project Demo (Feb 17 - March 3)

For the Mid-Project demo, the motion detection, VGA output, Bluetooth communication, and audio output were integrated together into a master project with a basic game loop.

Moreover, we also worked on including additional features such as:

- Drawing a static enemy sprite with basic action indicators
- Drawing player and enemy lives on screen
- Double Buffered frame buffer
- Upscaling for sprites and switched to 15-bit colour + 1 bit transparency to save on BRAM space
- Completed first iteration of game loop algorithm with pre-programmed actions
- Completed Bluetooth haptic feedback Arduino circuit with breadboard and vibration motors
- Verified Bluetooth communication from the FPGA through the Pmod BT2 (Auto-connect as Master) to the Arduino Nano through the HC-05 (Auto-connect as Slave).

We then demonstrated our mid-project demo to our lab TA without issues where we were able to run through an instance of the game where the character and enemy character exchanged vertical and horizontal attacks and blocks with the vibration motors buzzing when the user is hit as well as audio output for each motion and when winning the game.

## 3.5 Milestone 4 (March 3 - March 9)

| **Milestone Goals:** | **Actual Progress:** | **Team Member** |
|---|---|---|
| Modify haptic feedback code (Completed) | ● Implemented and tested new haptic feedback code where vibration motors will vibrate in a vertical or horizontal line depending on the attack on user if hit | Gavin |
| Optimize game loop algorithm (Completed) | ● Second iteration of game loop algorithm to include different states and other logic | Harry |
| More drawing states for swings and enemy (Completed) | ● Added enemy animations and lives to GUI | Harry, Nick |
| Troubleshoot audio and keyboard custom hardware blocks (Completed) | ● Exposed audio and keyboard custom hardware blocks to the overall system through AXI<br>● Fixed minor audio bugs when trying to add long songs into audio where previously it played inconsistently | Sam |
| Demonstrate sword-swinging and hit detection through GUI, audio output, and haptic feedback (Completed) | ● Finished during Mid-Project Demo | All |

In our fifth week, our team completed our last goal a week early for the mid-project demonstration and achieved all other goals. We also planned additional milestone goals for milestones 5 and 6.

## 3.6 Milestone 5 (March 10 - March 26)

| **Milestone Goals:** | **Actual Progress:** | **Team Member** |
|---|---|---|
| Experiment with motion detect for additional features such as diagonal motions (Completed, Ongoing) | ● Due to the temporary loss of our Pmod NAV, was estimating sensor values for diagonal motions and modified motion detection algorithm to include draft code | Gavin |
| Testing and modifying game loop or motion detecting algorithm as needed (Completed, Ongoing) | ● Cleaned up main file, removed unnecessary code, fixed game loop bugs<br>● Wrote draft code for an additional menu state in third iteration of game loop algorithm | Harry |
| Work on a additional screens for GUI (Completed) | ● Collaboratively worked on drawing a start screen, win and lose screens, and adding additional frame animations for the enemy | Harry, Nick |
| Work on integrating keyboard with game (Completed, Ongoing) | ● Fixed bugs related to keyboard communication with a clashing output port of uart_usb used in serial terminal communication | Sam |

In our sixth week, we faced a roadblock where we only had one Nexys Video board as our second board, Pmod NAV, and Arduino haptic feedback system was temporarily misplaced for the week. Moreover, another member was sick and we all had a midterm in the same week. However, we were able to maintain progress and complete all goals with a bit more coordination and communication than most weeks.

## 3.7 Milestone 6 (March 17 - March 23)

| **Milestone Goals:** | **Actual Progress:** | **Team Member** |
|---|---|---|

| | | |
|---|---|---|
| Experiment with diagonal motion detect (Feature Dropped) | • Tested code with diagonal blocks and swings, ensure functionality is working<br>• Team decided to drop this in our final game as little time to modify visual components to include new animations | Gavin |
| Fix and retest Arduino haptic feedback system (Completed) | • Rewired loose connections tested horizontal and vertical hit detection haptic feedback | Gavin |
| Random Actions<br>Add second difficulty<br>Final Testing (Completed) | • Implemented randomizer for enemy actions<br>• Added “hard mode” with less time for action and increases number of enemy lives | Harry |
| Output run time information onto terminal or GUI (Completed) | • Added real-time G tracker to visualize PMOD NAV data & threshold for action to register<br>• Added GUI timer for remaining action time | Nick |
| Work on integrating keyboard with game (Completed) | • Added keyboard functionality to start the game and select a difficulty mode and restart the game when either win or lose screen appears | Sam |
| Test and polish full functioning game in preparation for final demo (Completed) | • Tested full game, made plans for final polish in preparation for final demo | All |

In our seventh week, our team completed all goals except for integrating additional motion detection actions for the enemy and user. The decision to drop this feature was made when deciding that we want to prioritize our time on ensuring our current game model will work on the final demo with no errors rather than risk adding an additional feature that may be error-prone. However, we did add a G tracker and timer to display some important run information on screen, a feature previously postponed from Milestone 3.

## 3.8 Final Demo (March 24 - March 30)

In our last week, we were completing this project on track with our proposed timeline and thus primarily focused on testing and fine-tuning our overall game. This also included:

- Drawing additional lives + shortened timer bar for “hard mode”

- Increased clarity by adding text to display enemy action
- Tested and fine-tuned motion detection thresholds
- Added on 7-seg on FPGA display a game score for successful user actions
- Ensured security of vibration motor connections will not come undone during demo
- Attach Pmod NAV to a foam sword and attach Arduino haptic feedback system on a piece of cardboard for the user to wear over their neck with the vibration motors taped on to their body in a vertical and horizontal alignment

We demonstrated our final project demo to the professor and all lab TAs without issues.

# 4.0 Description of Blocks

## 4.1 Nav Accelerometer

### 4.1.1 Supporting Hardware

To detect motion from the user, we used the Pmod NAV which contains a 3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometer, and barometer and operates at 3.3V [2]. The associated custom IP block is borrowed and first extracted from Diligent's Pmod Library for Vivado version 2018.2 and added to the folder for our Microblaze project [3]. After including it in our IP repository, we attach the NAV to the JA Pmod port both physically and in our project because it supports SPI connection which is the Pmod NAV's communication protocol. The last step is to connect a 50 Mhz reference clock which we added by enabling a second clock output on our clocking wizard from our 100 Mhz clock input.

After validating and generating a bitstream for our design, we launched the SDK demo code that is included from the Diligent Library folder. This outputs temperature, pressure, and real-time sensor values from all 3-axis sensors. We chose the Pmod NAV because we intended to use both the accelerometer and gyroscope for motion detection, however, to ensure reliable results we decided to rely solely on the accelerometer and detect for a specific acceleration threshold to indicate movement. The gyroscope would have helped for orientation detection where after calibrating it upon start-up and accounting for drift, we could integrate over time to determine the angle of the sword to determine how the user is holding it but finding the angle was difficult, and we were interested in instantaneous values, not changes over time.

### 4.1.2 Motion Detection Algorithm

The algorithm to detect the player movement was implemented in C. Due to the complexity of sensor-fusion algorithms, and having no prior experience working with them, the team opted to first try a simpler approach for the motion detection and then determine if that would be sufficient. The algorithm is split into two components: block and slash detection. To detect a block, the algorithm finds the direction of the acceleration vector (in this case, due to gravity). If the angle between the acceleration vector and the cardinal axis of choice, X for vertical, -Z for horizontal, is below a certain threshold (A final value of 25 degrees was used, but the value was tweaked throughout testing) then a valid block angle is registered. To ensure that this was not a false positive from a slashing motion, the valid block would have to be held for at least 3 frames, before the player block action was accepted.

```
float degreesFromVerticalBlock(NAV_RectCoord r) {
   // Determine the magnitude of the vector r.
   float rM = sqrtf(powf(r.X, 2) + powf(r.Y, 2) + powf(r.Z, 2));
   if (rM == 0)
      return 0.0;
   return acosf(r.X / rM) * (180.0 / M_PI);
}
```

Figure 3: Function to determine the angular difference between the current orientation and pure vertical

To determine if the player slashed, a simple threshold check was used. From testing, an instantaneous value of 2.5 G's was found to be a good threshold while attached to the foam sword. The default maximum value for the accelerometer is 2 G's so this had to be changed to 8 G's to accommodate this. To find a vertical slash, the acceleration in the Z axis must go above 2.5 G's and for horizontal slashes, the acceleration in the Y axis must go above 2.5 G's or below -2.5 G's thus horizontal slashes work in both directions, and vertical slashes work only slash downwards. From testing, this algorithm was determined to be sufficient for the gameplay and the team chose to skip doing a more complex sensor-fusion type algorithm which would have integrated the gyroscope, to instead focus on adding other features.

## 4.2 BT2 Bluetooth Controller

To communicate data on the Nexys 4 DDR through Bluetooth, we opted to use the Pmod BT2 due to its simplicity as the data we would be sending would only be a one-character letter that indicates "H" for a horizontal slash and "V" for a vertical slash inflicted on the user. This Pmod employs RN-42 and communicates with our FPGA through UART on the JB Pmod port [4]. In our Microblaze project, we accomplish this through a built-in UARTLite IP block that is connected to our AXI Interconnect IP.

In the SDK, to communicate with another Bluetooth device, our code borrows the function print_bt from a different project on GitHub to output a character to the Arduino and signals the direction of the vibration motors to vibrate [5]. We tested this first by connecting to the Pmod BT2 from our phone and downloading a serial USB terminal on our phone to send strings [6]. After verifying that the function works as expected, we then followed a diligent tutorial to set up the BT2 into auto-connect master mode through commands used in RN42 [7]. This is completed by connecting our computer to the Pmod BT2 and through a terminal emulator such as Tera Term, we entered into “Command Mode" and configured our BT2 to auto-connect as a master to our HC-05 slave address at a high transmission baud rate of 115200.

## 4.3 Arduino System

For the Arduino system, the circuit contains both wiring for the HC-05 Bluetooth component and the vibration motor control component.

### 4.3.1 HC-05 Bluetooth

To communicate data on the Arduino Nano through Bluetooth, we opted to purchase an HC-05 due to its popularity and its ability to communicate serially [8]. This module communicates with the Arduino Nano through UART and to configure it as a slave device, we followed instructions to enter into AT mode and entered additional AT commands to set it up as a slave and obtain its address to provide to the BT-2 [9]. We also verified that the baud rate was 115200 to match the BT2 through an AT command. Next, we operate the HC-05 from the Arduino Nano at 5V but when receiving data from the HC-05, the Arduino Nano input pins only accept 3.3V. Thus, to step down the incoming voltage to a safe 3.3V, we created a voltage divider with a 1kΩ and 2kΩ resistor which can be seen in Figure 4 along with the rest of the wiring between the HC-05 and Arduino Nano.

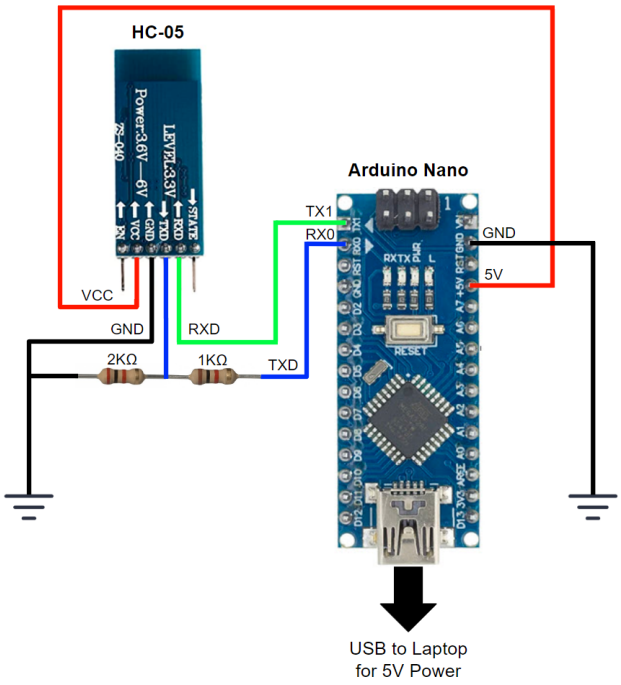



Figure 4: Wiring schematic between HC-05 and Arduino Nano

For the Arduino code, we include the header <SoftwareSerial.h> to create an instance of the SoftwareSerial class called EEBlue and assign it to the RX and TX pins on the Arduino Nano. Next, in our main loop as seen in Figure 5, we repeatedly check whether data is available from EEBlue and when it is, we add the received character into our data string. Lastly, we terminate once we see the character for carriage return which has an ASCII value of 13 and indicates that our incoming data is finished and resets our data string variable to blank. We tested it by connecting it to our phone through a serial USB terminal and sending text for Arduino IDE to print on the terminal and after verifying it can receive data, we let it auto-connect to the BT-2 as a slave. We can verify the connection is established once the status LED on the HC-05 begins to blink slowly at 2s intervals and thus our HC-05 is ready to receive a character from the BT-2 and signal the Arduino Nano to control the vibration motors to either vibrate vertically or horizontally.

```
while(EEBlue.available()==0)
{}

char character;
while(EEBlue.available()>0)
{
  character = EEBlue.read();
}

data += character;

if (character == 13) {
  data = "";
}
```

Figure 5: Main loop code to receive string data

### 4.3.2 Vibration Motor

The goal for our vibration motor component is to enable our Arduino to emit a digital PWM signal to an NPN transistor, which will act as a switch to connect with different vibration motors depending on the value of the character received through Bluetooth from the FPGA. The circuit diagram can be seen in Figure 6 which illustrates the digital PWM pin being connected to the transistor through a 1kΩ resistor to prevent excessive current from flowing through the output pin. The transistor is then connected to a motor and in cases where we use two motors, we can connect the second one in parallel.

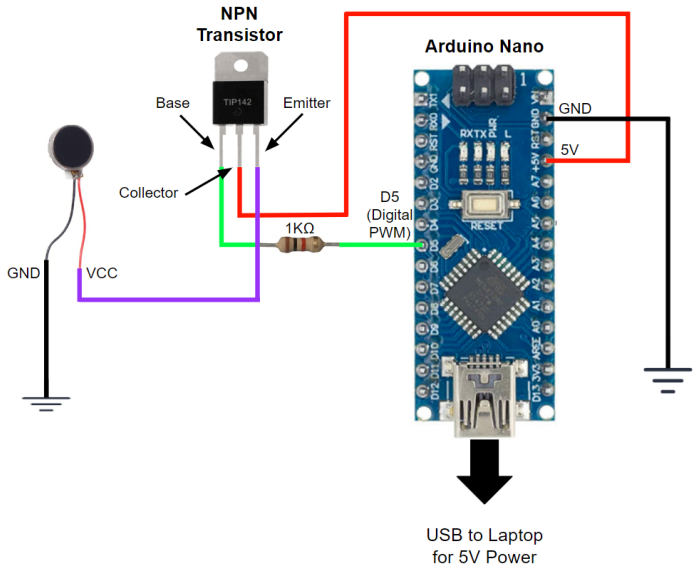



Figure 6: Wiring schematic between one vibration motor and Arduino Nano

Three NPN transistor units were used to control the vibration motors. Figure 7 illustrates the setup of our vibration motors when attached to a user where the digital pin 3 from Arduino controls the two horizontal vibration motors, digital pin 6 controls the two vertical vibration motors, and digital pin 5 controls the center vibration motor that is always on when the other two are activated.

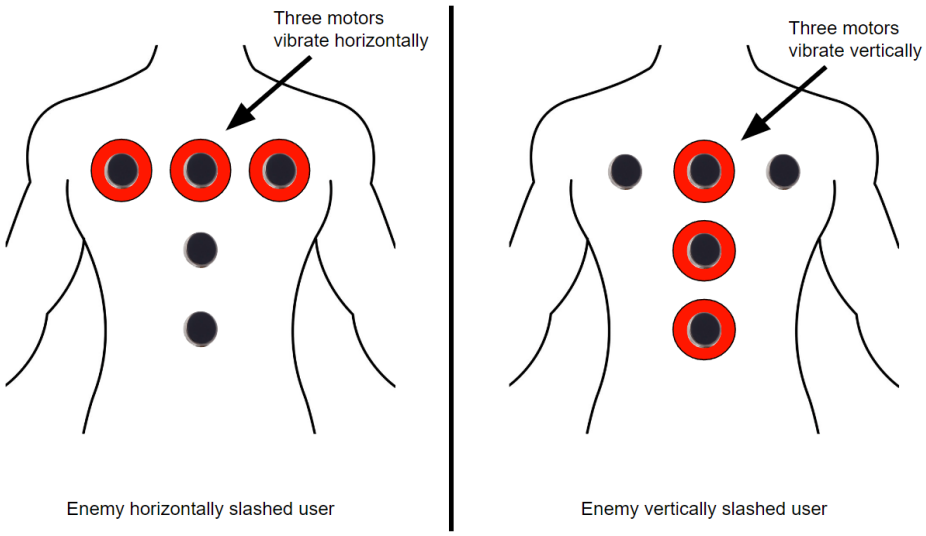



Figure 7: Vibration motors setup to vibrate either horizontally or vertically

The code written in Arduino IDE for the main loop to control these motors to vibrate for 2 seconds before turning them off as shown in Figure 8.

```
if(character == 'H') {
  digitalWrite(3, HIGH);
  digitalWrite(5, HIGH);
  delay(2000);
  digitalWrite(3, LOW);
  digitalWrite(5, LOW);
}

if(character == 'V') {
  digitalWrite(6, HIGH);
  digitalWrite(5, HIGH);
  delay(2000);
  digitalWrite(6, LOW);
  digitalWrite(5, LOW);
}
```

Figure 8: Main loop code to vibrate horizontally or vertically for 2 seconds depending on the character.

We tested the code in SDK by using the print_bt function to send either a “H” or “V” from BT2 to HC-05 once the connection was established and verified that only three vibration motors were vibrating at a time. Once this was confirmed, it was integrated into the game loop to respond to missed blocks by the player.

## 4.4 Audio Core

To create music and sound effects for the game, we had to first produce musical notes. Outputting a specific note from the audio output is as simple as feeding it a square wave that has the frequency of a specific note. For example, middle C has a frequency of 261.63Hz (at the A440 standard pitch scale [10]); thus to produce a middle C sound, a square wave that toggles between one and zero 261.63 times per second needed to be fed through the audio output on the board.

However, these notes are sharp and generally sound unpleasant. To make them sound better, we needed to apply pulse wave modulation (PWM). This meant that the transition from a one to zero or a zero to one needed to be a gradual decrease or increase in the duty cycle, rather than a sharp jump, as shown in Figure 9. This meant that every note not only has a specific frequency and a specific duty cycle duration but also a certain period of time that is considered the interval between neighbouring duty cycles.

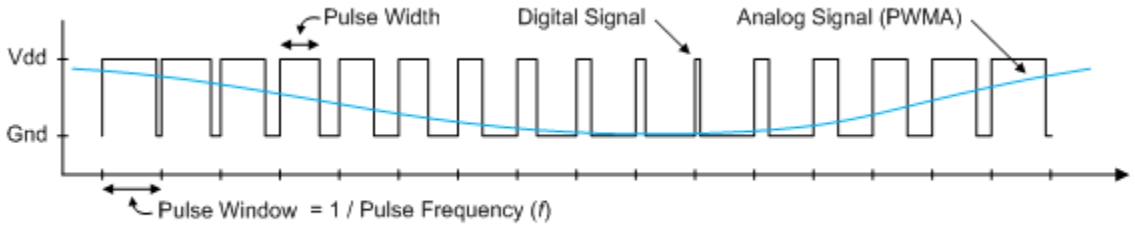



Figure 9: Pulse Wave Modulation [11]

To prevent run-time calculations, especially as this would be dealing with division, we devised to encode all the values a note would need to be played into a single 32-bit value. The encoding is seen in Figure 10.

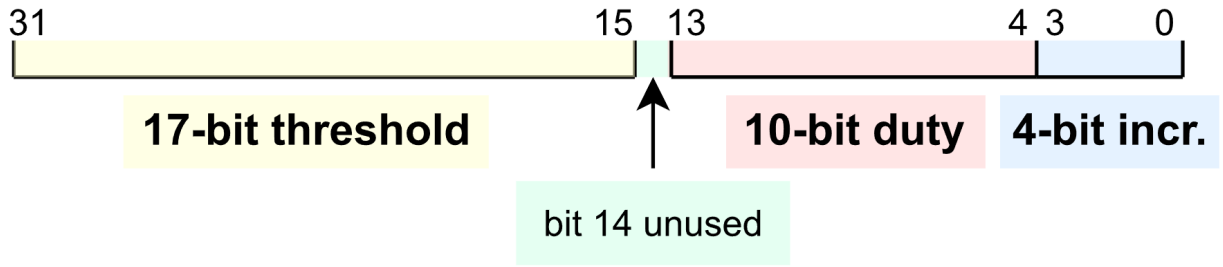


Figure 10: Encoding used for audio notes

Given a 50MHz clock and a certain note, “threshold” is the number of clock cycles an increasing or decreasing period of duty cycles would last (or half the note’s frequency). “Resolution” was the term to used to define the number of unique duty cycles a note would have to go through to go from 0% duty to 100%. The resolution was 64 for all notes such that the encoded values could fit inside a 32-bit value. “Duty” is the duration for one of these duty steps. “Incr”, short for increment, is the increment duration for how long a logic 1 needed to be held. For example:

A C5 has a frequency of 523.25. Thus in a 50MHz domain, each period is (50,000,000 / 523.25) ≈ 95,556 clock cycles long. This means the duration of an ascending or descending portion of the period would be 95,556 / 2 ≈ 47,778— this is our “thresh” value. To achieve a resolution of 64 distinct “duty periods”, each duty period would last 47,778 / 64 ≈ 747 clock cycles— our “duty” value. Lastly, the duration in which the signal is held high needs to increment from 0 cycles during the 1st duty period, to 747 cycles in the 64th duty period. Thus, the increment of the time where the signal is held high between each duty period is 747 / 64 ≈ 12 cycles— our “incr” value, meaning C5 would be encoded as shown in Figure 11.

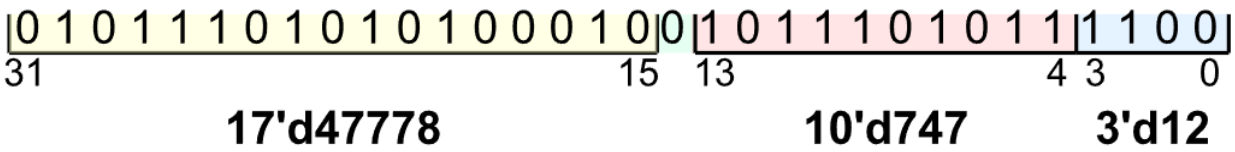
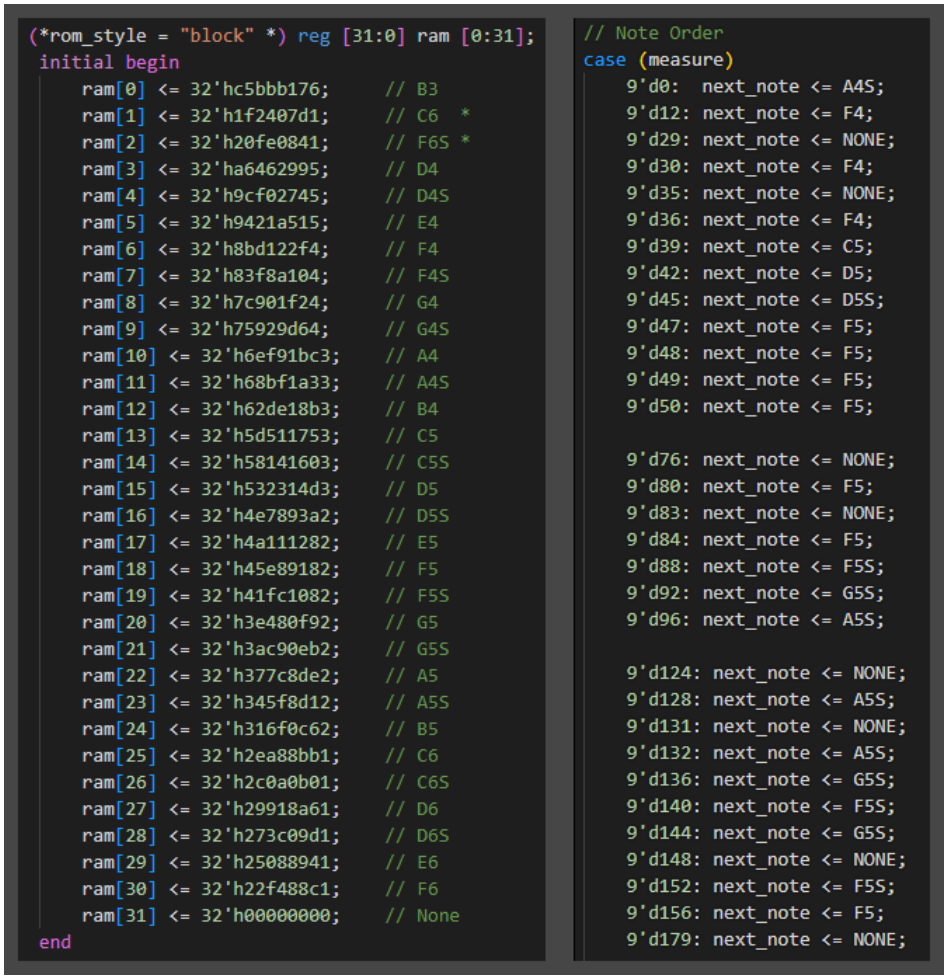



Figure 11: 32-bit encoding of C5 in A440 pitch scale

This encoding was done for all the notes we expected to use and put into a block ROM. Thus, each tune and sound effect was simply a pre-coded series of notes under a specified bpm (beats per minute). At specified times, the audio core would generate the specified note; several of these played in sequence creates the tunes and sound effects you can hear in-game.



```
(*rom_style = "block" *) reg [31:0] ram [0:31];
initial begin
    ram[0] <= 32'hc5bbb176;     // B3
    ram[1] <= 32'h1f2407d1;     // C6  *
    ram[2] <= 32'h20fe0841;     // F6S *
    ram[3] <= 32'ha6462995;     // D4
    ram[4] <= 32'h9cf02745;     // D4S
    ram[5] <= 32'h9421a515;     // E4
    ram[6] <= 32'h8bd122f4;     // F4
    ram[7] <= 32'h83f8a104;     // F4S
    ram[8] <= 32'h7c901f24;     // G4
    ram[9] <= 32'h75929d64;     // G4S
    ram[10] <= 32'h6ef91bc3;    // A4
    ram[11] <= 32'h68bf1a33;    // A4S
    ram[12] <= 32'h62de18b3;    // B4
    ram[13] <= 32'h5d511753;    // C5
    ram[14] <= 32'h58141603;    // C5S
    ram[15] <= 32'h532314d3;    // D5
    ram[16] <= 32'h4e7893a2;    // D5S
    ram[17] <= 32'h4a111282;    // E5
    ram[18] <= 32'h45e89182;    // F5
    ram[19] <= 32'h41fc1082;    // F5S
    ram[20] <= 32'h3e480f92;    // G5
    ram[21] <= 32'h3ac90eb2;    // G5S
    ram[22] <= 32'h377c8de2;    // A5
    ram[23] <= 32'h345f8d12;    // A5S
    ram[24] <= 32'h316f0c62;    // B5
    ram[25] <= 32'h2ea88bb1;    // C6
    ram[26] <= 32'h2c0a0b01;    // C6S
    ram[27] <= 32'h29918a61;    // D6
    ram[28] <= 32'h273c09d1;    // D6S
    ram[29] <= 32'h25088941;    // E6
    ram[30] <= 32'h22f488c1;    // F6
    ram[31] <= 32'h00000000;    // None
end
```

```
// Note Order
case (measure)
    9'd0:  next_note <= A4S;
    9'd12: next_note <= F4;
    9'd29: next_note <= NONE;
    9'd30: next_note <= F4;
    9'd35: next_note <= NONE;
    9'd36: next_note <= F4;
    9'd39: next_note <= C5;
    9'd42: next_note <= D5;
    9'd45: next_note <= D5S;
    9'd47: next_note <= F5;
    9'd48: next_note <= F5;
    9'd49: next_note <= F5;
    9'd50: next_note <= F5;

    9'd76: next_note <= NONE;
    9'd80: next_note <= F5;
    9'd83: next_note <= NONE;
    9'd84: next_note <= F5;
    9'd88: next_note <= F5S;
    9'd92: next_note <= G5S;
    9'd96: next_note <= A5S;

    9'd124: next_note <= NONE;
    9'd128: next_note <= A5S;
    9'd131: next_note <= NONE;
    9'd132: next_note <= A5S;
    9'd136: next_note <= G5S;
    9'd140: next_note <= F5S;
    9'd144: next_note <= G5S;
    9'd148: next_note <= NONE;
    9'd152: next_note <= F5S;
    9'd156: next_note <= F5;
    9'd179: next_note <= NONE;
```

Figure12: Encoding for notes used (left); a snippet of the song played in the menu (right)

These pre-coded songs and sound effects were given an arbitrary encoding and exposed as memory-mapped AXI slaves to the MicroBlaze. Thus, when the audio core's memory-mapped address was written to by the software, the tune or sound effect that corresponded to the value written would play, allowing full control from the game logic side. Testing was done as a standalone hardware block before being integrated into the system via AXI  and tested as a whole system.

## 4.5 Keyboard and 7-Segment Core

This core is largely derived from the keyboard demo provided by Diligent [12] and modified slightly for this project. AN7 and AN6 have been changed to display the score, while AN5 and AN4 are not used. AN3-AN0 are not modified and still display the last pressed key and confirmation of debouncing.

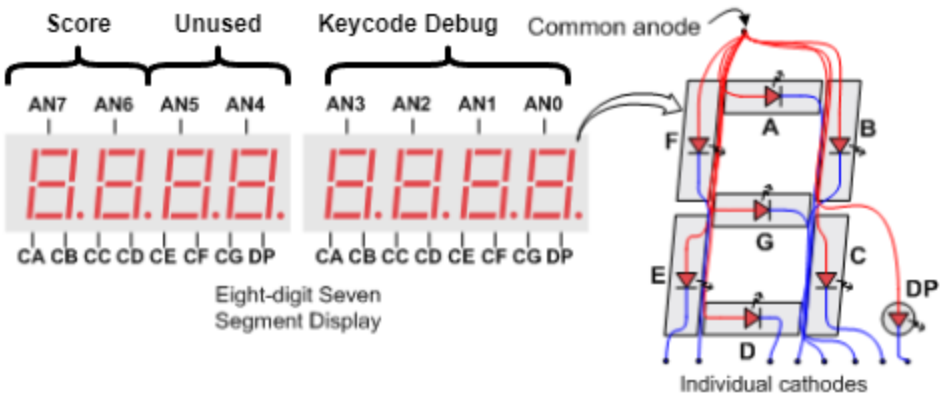



Figure 13: Usage designation of 7-segment display [11]

The core is exposed as a memory-mapped AXI slave to the MicroBlaze. Within the menu, the game loop will poll the memory-mapped location for a specific keycode. 4'hF016 (1 Key) to start the game in easy difficulty, and 4'hF01E (2 Key) to start in hard difficulty. After the game ends, it polls for 4'hF02D (R Key) to restart the game or 4'hF076 (ESC key) to end the program. These keycodes are pre-established scan codes (Figure 14) and are unmodified from the demo project. The presence of an F0 indicates the key has been released.

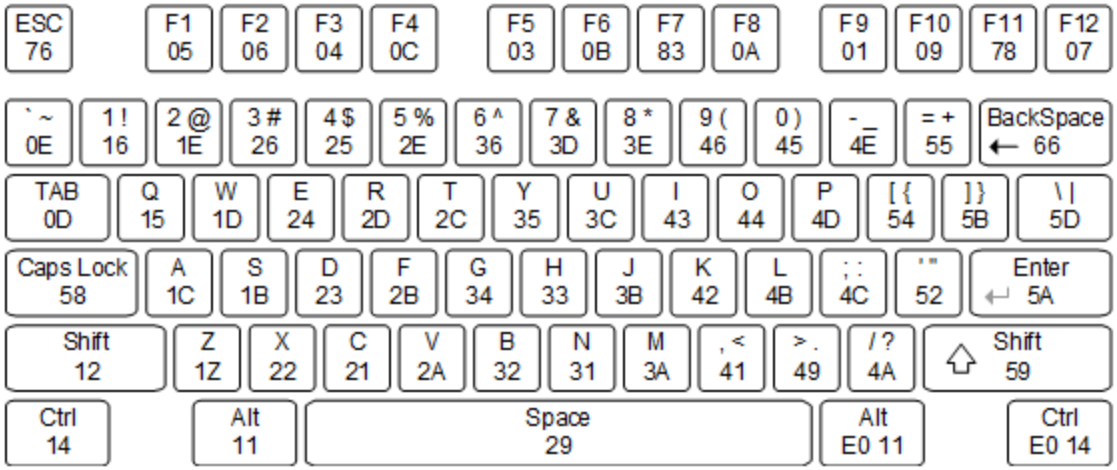



Figure 14: Predetermined keyboard scan codes [11]

During the game loop, the player score is written to the memory-mapped location on every update and is read by the core to update the 7-segment display. Both the keyboard and 7-segment display testing was done as a standalone hardware block before being integrated into the system via AXI and tested as a whole system.

## 4.6 Visual System

The function of the visual system was to use the VGA display for visual cues and feedback to the player. It combined both hardware and software, with the hardware system handling the timing and sending of signals to the VGA output, and the software system handling what to draw and when to draw it. The hardware system was designed around the AXI TFT Controller, a Xilinx display controller IP that was found when searching for display controllers. However, the rest of the circuit and the software control were all designed by the team, with help from the documentation and an online example [13], [14].

### 4.6.1 VGA Hardware System

The VGA controller was implemented using the AXI TFT (thin film transistor) Controller. It is a hardware display controller IP capable of VGA and DVI control, with register access via AXI. It handles the generation of h_sync and v_sync signals, and transportation of pixel information from memory to the VGA display. Writing pixel information to the DRAM frame buffer was done through the pixel address, defined as "*TFT Base Address + (4096 * row) + (4 * column)*". Within the TFT register space is also a status and control register. The status register indicates when a previous frame has been displayed completely, and the control register starts/resets the TFT. To clock the VGA system, a 25Mhz Pixel Clock was used, enabling an image of 640x480 at a 60Hz refresh rate. The AXI TFT uses an 18-bit colour scheme. However, the VGA I/O on the Nexys DDR displays in 12-bit colour, so x_slices were used to truncate each of the 6-bit R, G, B channels to 4 bits each. Finally, connection to the PMOD Nav was needed to implement a real-time G-Tracker onscreen, which would show current G levels in each coordinate axis, and the threshold needed for an action to register.

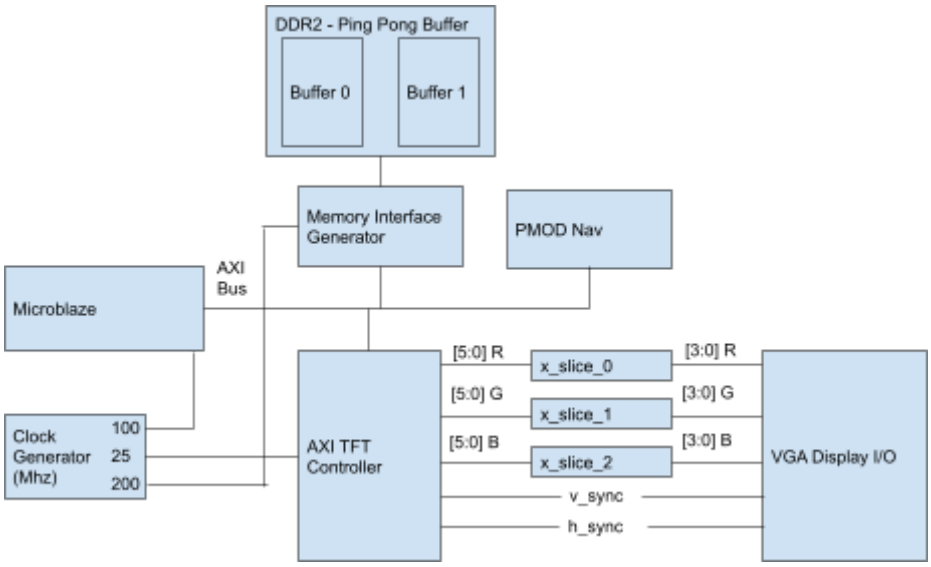



Figure 15. Block Diagram of VGA Display System

### 4.6.2 Software Library

To handle drawing to the Pixel Address of the TFT, which would write to the DDR2 frame buffer, the AXI TFT was controlled by the MicroBlaze. Colour information and image bitmaps are found in the "colors.h" file, and the library of functions is found in "draw_tools.c". The library was split into 2 main areas: control, and drawing functions.

Control refers to functions outside of drawing: the initialization step and call to swap pointers for double buffering. Initialization is important to ensure the smooth operation of the double buffer and starts up the TFT. The dedicated Swap function is also important since it draws from the status register to ensure that the double buffer functions properly without tearing.

Drawing functions made up the majority of the library, with simple functions to draw lines and shapes, but also more intricate functions to draw the enemy, lives, and other on-screen items. On-screen items were stored within the MicroBlaze, with 15-bit colour + 1-bit transparency to save memory space, and a function was written to convert the 15+1-bit palette to a true colour 24-bit scheme. Pixels were also upscaled by certain factors to help with memory usage. For example, our skeleton sprite is 29x70 = 2030 pixels, and at 16bpp, a total of 32,480 bits. During runtime, the skeleton is scaled to 4x the size. If this was stored without any memory saving initiatives: 29x4 x 70x4 = 32,480 pixels at 24bpp = 779 520 bits. In other words, these functions translated to a **95% reduction** in bits needed to store the enemy image. Finally, a G tracker and timer were also drawn on screen. The G tracker samples data from the PMOD Nav and shows the current G level for each axis, as well as a threshold for actions to register. The timer shows the amount of time left to make an action. A list of all the functions is found in the table below.

| Header Definition | Description |
|---|---|
| init_tft() | Initializes and starts the TFT, defines ALT pointer for 2nd buffer |
| SwapDrawBuffers() | Swaps buffer pointers, then waits for display to signal complete update |
| DrawRectangle (int xstart, int ystart, int width, int height, unsigned int color) | Given a start position, size, and colour, draws rectangle |
| DrawLine(int x_0, int x_1, int y_0, int y_1, int scale, int | Follows Bresenham's line drawing |

| color) | algorithm |
|---|---|
| DrawBackground() | Fills screen w/ background colour |
| DrawMenu() | Draws start menu items |
| DrawWin() | Draw win screen + items |
| DrawLose() | Draw lose screen + items |
| DrawLives(int num_player_hearts, int num_enemy_hearts, int hardmode_on) | Draws both enemy and player hearts, hardmode dictates the max number of hearts |
| DrawPixel( int xpos, int ypos, int scale_factor, unsigned int color) | Writes to TFT Pixel Address. Pixel is scaled up by scale_factor, with xpos, ypos acting as (0,0) of scaled pixel. Calls Convert16bppTo24bpp |
| Convert16bppTo24bpp(int color) | 16-bit to 24-bit colour converter. Separates out each 5-bit channel, extrapolates to 8 bits each, rearranges in 24-bit colour format |
| DrawEnemy(int pstate) | Enemy has different states:<br>0: enemy vertical attack<br>1: enemy horizontal attack<br>2: enemy vertical block<br>3: enemy horizontal block<br>4: enemy vertical attack hit<br>5: enemy horizontal attack hit<br>6: enemy idle state |
| DrawGtrack(float x_data, float y_data, float z_data) | Given PMOD Nav data, draws corresponding g-levels and threshold level of 2.5 G's |
| DrawTimer(int counter, int action_period) | Given current time, and amount of time available, draws a timer bar |

Testing of these functions was fairly straightforward. Other than the G-tracker, the rest of the functions were independent, and not reliant on other parts of the design. This allowed for constant, iterative testing, and also helped to ensure a smooth and painless integration with the other components. Functions were written on a basic level, tested, then iteratively improved to reach the level of polish that was wanted. Once a function was complete, it was added to the main game loop and made sure that it functioned and looked like it should, and that the rest of the system still functioned. The functions were also written in close coordination with the game loop designer's expectations of how they wanted to call the functions.

## 5.0 Description of Design Tree

Posted to GitHub (https://github.com/NicholasTran/G8_SwishSwooshSwordSimulator) is the entire project repository. This includes the Vivado project, image assets, final presentation, report, and a video demo in action. Below is a breakdown of the Github repository, with notes on some of the notable files.

.

```
├── docs/
│   ├── Presentation.pdf              # Final Project Presentation
│   ├── Final_Report.pdf              # Final Report
│   └── Video_demo.txt                # Link to Video Demo
├── images                            # Folder of image sprites used for on-screen objects
├── src/
│   ├── ip_repo/                      # Custom / Imported IP for Vivado
│   │   ├── audio_axi_slave_1.0       # Custom Audio IP
│   │   └── kb_slave_1.0              # HID Keyboard IP
│   │   └── bluetooth_multi.ino       # Bluetooth Arduino Code
│   ├── mid_demo/
│   │   ├── mid_demo.cache            # Vivado Cache Files
│   │   ├── mid_demo.hw               # Vivado Hardware Files
│   │   ├── mid_demo.ip_user_files
│   │   ├── mid_demo.runs             # Vivado Runs
│   │   ├── mid_demo.sdk/             # Vivado SDK Folder
│   │   │   ├── RemoteSystemsTempFiles
│   │   │   ├── main/                 # Main SDK Source Files
│   │   │   │   └── src/
│   │   │       ├── colors.h          # Header File defining colors, on screen image bitmaps
```

```
│   │   │   │   ├── draw_tools.c      # Main Library for Drawing Functions
│   │   │   │   ├── draw_tools.h   # Header File for Drawing Library
│   │   │   │   ├── helloworld.c   # Main game loop w/ Motion detection
│   │   │   │   ├── lscript.ld        # Linker Script
│   │   │   │   ├── platform.c
│   │   │   │   ├── platform.h
│   │   │   │   └── platform_config.h
│   │   │   ├── main_bsp/             # Board Support Files
│   │   │   │   ├── Makefile
│   │   │   │   └── system.mss
│   │   │   ├── mid_project_wrapper_hw_platform_1
│   │   │   │   └── mid_project_wrapper.hdf              # Hardware Description File
│   │   ├── mid_demo.srcs                                # Vivado Verilog Source Files
│   │   ├── mid_demo.tmp
│   │   └── mid_demo.xpr                                 # Vivado Project File
│   └── vivado-library-hotfix-PmodOLED_RGB      # Directory Containing PMOD NAV IP
└── README.md
```

# 6.0 Tips and Tricks

## 6.1 Pmod NAV

If using the Pmod NAV, we highly suggest viewing the following Diligent tutorial for working with Diligent Pmod IPs https://digilent.com/reference/learn/programmable-logic/tutorials/Pmod-ips/start. This is helpful because it shows you how to extract and integrate their custom IPs as well as the reference clock frequency required and if interrupts are needed. Moreover, when creating a new Microblaze project, we suggest immediately increasing local BRAM memory to 512 KB or more through the address editor as this is required to run the Diligent demo in SDK and avoid issues generating the elf file. Moreover, the addressing of the Pmod NAV is automatically assigned after the end of the BRAM memory address and thus there are addressing overlap issues if you do not increase the BRAM memory first. Lastly, if you are using Diligent Pmod IPs, the warning that the IP was packaged with a different board value can be ignored as they only serve to inform the user that the IP core was made with a different board https://forum.digilent.com/topic/8728-messages-in-vivado-about-pmod-packaging/.

## 6.2 Pmod BT2

If using the Pmod BT2, we suggest the following guide from a former ECE532 student where they outline the steps to add an AXI UARTLite block and modify it to support the Bluetooth connection as well as debugging problems with an ILA core [15]. Moreover, to auto-connect two Pmod BT2's together, we also suggest viewing Diligent's tutorial as they outline the steps to initialize the BT2 in the terminal and what commands to send [7].

## 6.3 Nexys 4 DDR Audio Output

For this board specifically, there are two pins that must be connected to create audio: AUD_PWM on pin A11, and AUD_SD on pin D12. The former is where the audio wave needs to be written to, and the latter is an enable signal that needs to be held high. Unfortunately, the reference manual for the Nexys 4 DDR board on Diligent's website does not mention the existence of AUD_SD or pin D12 at all and we only learned about it by looking at the schematic. Less a suggestion, but if one wants to hear audio output, do connect pin D12 and hold it high.

## 6.4 USB Keyboard

For reasons unknown, most modern USB keyboards will not be recognized by the Nexys 4 DDR board. The team tried numerous self-owned keyboards and none of them worked with the board. The only keyboards that did work were the bare-bones simple keyboards connected to the lab computers in the FPGA labs. The team suggests trying these keyboards first when debugging anything keyboard related as it could not be a code problem, and simply a hardware compatibility issue on the FPGA's end.

## 6.5 Video System

With the video system, our initial plans of using the Nexys Video for HDMI output was thwarted after realizing the Digilent Nexys Video HDMI demo does not work for the given version of Vivado. It constantly ran into errors with the DVI protocol handshake that were extremely difficult to debug. Hours were spent scouring forums, debugging, and searching through the limited documentation, to no avail. The team was able to get the demo working on older versions of Vivado, but not the 2018.3 version used in the course. Our group decided to switch to the Nexys DDR board, but we lost a week of progress debugging the HDMI system. With that in mind, future students who aim to use the HDMI output on the Nexys Video should expect the HDMI demo to be complicated and hard to work with, and anticipate using an older / newer version of Vivado to get even a basic image displayed.

### 6.6 Random Number Generation

As a part of the gameplay, we wanted to randomize the enemy actions. While the C rand() function does work, the typical method of seeding using the time library does not work as the time functions are not available on microblaze. To properly seed the rand function some other method must be used. The team used a counter on the menu screen and the amount of frames spent on the menu was used as a seed. This achieved the pseudo-random number generation that we wanted.

## 7.0 Video

Project Video Overview: https://youtu.be/H3OmKn2vGKM\

## 8.0 References


[1] "Gameplay -- The Legend of Zelda™: Skyward Sword HD -- Nintendo Switch™ – Official Site." https://www.zelda.com/skyward-sword-hd (accessed Apr. 13, 2023).

[2] "Pmod NAV reference manual - digilent," *digilent.com*. [Online]. Available: https://digilent.com/reference/_media/reference/pmod/pmodnav/pmod_nav_rm.pdf. [Accessed: 13-Apr-2023].

[3] Digilent, "Releases · Digilent/Vivado-Library," *GitHub*. [Online]. Available: https://github.com/Digilent/vivado-library/releases. [Accessed: 13-Apr-2023].

[4] "Pmod BT2 Reference Manual," *digilent.com*. [Online]. Available: https://digilent.com/reference/pmod/pmodbt2/reference-manual. [Accessed: 13-Apr-2023].

[5] S. khalil Ahmed, "BT2PmodExample/helloworld.c at master · Shariqkhalilahmed/BT2PmodExample," *GitHub*, 13-Apr-2017. [Online]. Available: https://github.com/shariqkhalilahmed/BT2PmodExample/blob/master/src/bt_test_proj/bt_test_proj.sdk/bt_simple/src/helloworld.c. [Accessed: 13-Apr-2023].

[6] "Serial USB Terminal - apps on Google Play," *Google*. [Online]. Available: https://play.google.com/store/apps/details?id=de.kai_morich.serial_usb_terminal&hl=en_CA&gl=US. [Accessed: 13-Apr-2023].

[7] M. Skreen, “How to Auto-Connect 2 PmodBT2's Together,” *diligent.com*. [Online]. Available: https://digilent.com/reference/learn/programmable-logic/tutorials/auto-connect-2-bt2_s/start. [Accessed: 13-Apr-2023].

[8] “DSD Tech HC-05 bluetooth serial pass-through module wireless ... - amazon,” *amazon.ca*. [Online]. Available: https://www.amazon.ca/DSD-TECH-HC-05-Pass-Through-Communication/dp/B01G9KSAF6. [Accessed: 13-Apr-2023].

[9] Mybotic and Instructables, “How to configure HC-05 bluetooth module as master and slave via at Command,” *Instructables*, 25-Sep-2017. [Online]. Available: https://www.instructables.com/How-to-Configure-HC-05-Bluetooth-Module-As-Master-/. [Accessed: 13-Apr-2023].

[10] “Frequencies of Musical Notes, A4 = 440 Hz.” https://pages.mtu.edu/~suits/notefreqs.html (accessed Apr. 13, 2023).

[11] “Nexys 4 DDR Reference Manual - Digilent Reference.” https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual (accessed Apr. 13, 2023).

[12] “Nexys 4 DDR Keyboard Demo - Digilent Reference.” https://digilent.com/reference/learn/programmable-logic/tutorials/nexys-4-ddr-keyboard-demo/start (accessed Apr. 13, 2023).

[13] “AXI Thin Film Transistor Controller v2.0,” AMD Adaptive Computing Documentation Portal, 18-Nov-2015. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg095-axi-tft. [Accessed: 13-Apr-2023].

[14] AksharJ, “Interfacing the AXI TFT controller with the MiG IP and using it to display using the VGA port on the Nexys4 DDR Board,” Instructables, 02-Oct-2017. [Online]. Available: https://www.instructables.com/Interfacing-the-AXI-TFT-Controller-With-the-MIG-IP/. [Accessed: 13-Apr-2023].

[15] S. K. Ahmed, “Shariqkhalilahmed/BT2PmodExample,” *GitHub*. [Online]. Available: https://github.com/shariqkhalilahmed/BT2PmodExample. [Accessed: 13-Apr-2023].