Square Game

ECE 350 Final Project

Spring 2025

Mathew Chu (mc866) & Patrick Zheng (pz65)

Note that, for future reference, “we” or other similar references such as “our” refer to the project group of Mathew Chu and Patrick Zheng (and by extension any work produced by said project group) unless otherwise specified. References may be made to code or files that serve to enhance the overall understanding of this project.

Project Design and Specifications Overview

Our project displays a randomly moving target square on a screen that players need to follow. Players will tilt their FPGA box to move their player icons on the screen, gaining points while their player icons are within the target square.

Players are first presented with a starting screen instructing them to select between three difficulties (easy, medium, hard). They can select the difficulty by using the onboard FPGA buttons as instructed in the diagram shown on the starting screen. The difficulties modify the game as follows:

1. Easy Difficulty: Lowest box speed, 9 lives provided
2. Medium Difficulty: Medium box speed, 5 lives provided
3. Hard Difficulty: Highest box speed, smaller target and player icon, 2 lives provided

During gameplay, players must have the entire player icon in the target icon. This target icon moves in a random direction, switching directions after a random amount of time. In the event this condition is not met, the game will begin an internal counter for an allotted period of time the player is outside the target icon. In the event the player is still not in the target icon when the internal counter ends, a life will be decremented. Upon reaching 0 lives, the game will end and return the player to the start screen. These lives are displayed on the VGA screen and can be viewed in real time during gameplay, however, the internal counter previously mentioned is not displayed to the user.

To keep track of the player’s performance and to test their knowledge of binary counting, a player’s score is shown at the bottom of the FPGA using the onboard LEDs. While the player is within the bounds of the target box, their score will increase. Players can view their score in real time, as well as upon loss. In the event the player cannot interpret binary, they can also view their score by viewing the seven-segment display on the FPGA.

In the event the player loses all of their lives and returns to the start screen, they can restart the game by selecting a difficulty. The game is entirely cyclical in this way, meaning the player can try out the different difficulties.

The FPGA is housed inside a 3D-printed enclosure that allows access to key ports on the FPGA for ease of use. Included in this enclosure is a speaker, which plays background music fitting of an 8-bit theme. In addition, the FPGA is battery-powered, ensuring that the enclosure only needs a connection to the VGA screen. The enclosure allows for easy access and viewing of the required elements for playing the game, and provides a place for a player to hold the FPGA for tilting purposes.

Included components are as follows*:

- 1x Nexys A7 Board
- 1x Toguard x D701 Display
- 1x VGA to mini-USB Cable
- 1x 9-volt Battery with Barrel Jack Cable

- 1x 5-volt S13V30F5 Voltage Regulator
- 1x Speaker
- 1x Plastic Enclosure with Lid

*This list is non-exhaustive (i.e includes important components only)

Player I/O

For convenience and ease of reading, the summarized player I/O is listed below:

Inputs
- Tilt Controller
- Difficulty Select Buttons

Outputs
- VGA Display
- Score LEDs
- Score Segmented Display
- Background Music Speaker

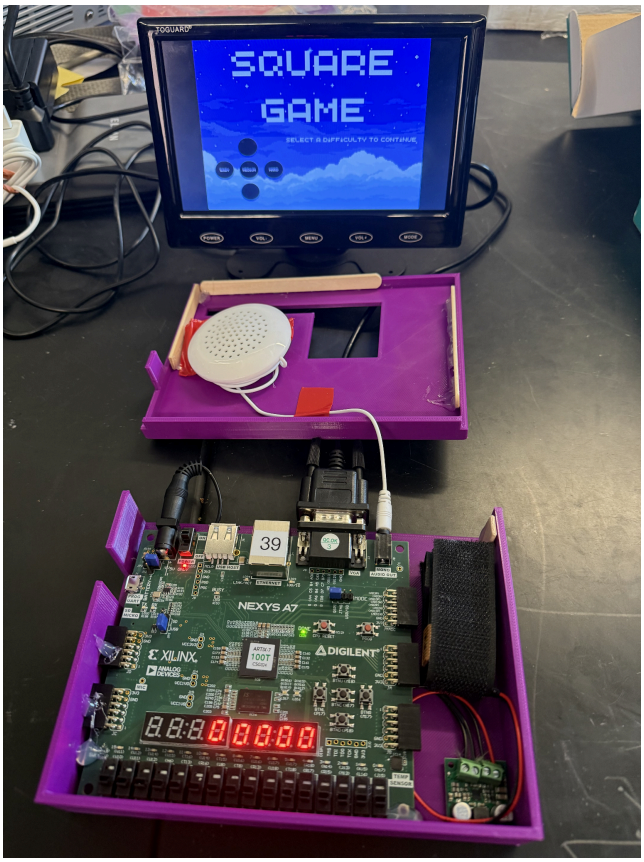


Figure A: Final Product with Visible Internals

Assembly

For ease of explanation, the general game loop and logic are presented in a table format. In the tables below are the custom instructions used, as well as the reserved registers.

Instruction	Type	Opcode	Purpose
upx	r-type	00000 (01000)	Update register 28 with accelerometer data in the x direction.
upy	r-type	00000 (01001)	Update register 26 with accelerometer data in the y direction.
rand $rd	r-type	00000 (01010)	Update register $rd with a 32-bit pseudo-random number.
diff	r-type	00000 (01011)	Update register 29 with difficulty selection from the buttons.

These custom instructions are all automatically assembled in assemble.py in the same way that the standard MIPS instructions are compiled.

Name	Number	Purpose
$gs	29	Stores game state.
$ax	28	Stores raw accelerometer data in the x direction.
$px	27	Stores processed center of player x position.
$ay	26	Stores raw accelerometer data in the y direction.
$py	25	Stores processed center of player y position.
$bx	24	Stores target box center x position.
$by	23	Stores target box center y position.
$bd	22	Stores target box direction (0: right, 1: left, 2: up, 3: down).
$bs	21	Stores box speed in pixels per update.
$pl	20	Stores remaining player lives count.
$ot	19	Stores number of cycles player icon is not within target icon.
$ps	18	Stores player score.

Using the previous table, we provide a general overview of the assembly file, gameloop.s. We will first wait for the player to select a difficulty via the FPGA buttons. The custom “diff” command is run to check for any

button presses. Once the game state register is non-zero, a difficulty has been selected, and the game can be initialized accordingly.

To initialize the game, the score register is first reset to 0 through an addi instruction. We then initialize two counters. First, register $t4 is initialized to 15000, representing the number of gameloops that should run before the target position is updated. Next, a random number is generated into $t7. By bit masking this 32-bit random number 2^{18}-1, we get a random number within 18 bits. We then add 2^{18}-1 to this number, effectively generating a random number between 2^{18}-1 and 2^{19}-1. This value represents the number of gameloops that should run before the direction of the target box is changed and is randomized every time a target box direction change occurs. This ensures a random period of time before the target box changes direction. This target box is then positioned to start in the middle of the screen. After this, an out-of-bounds timer is initialized to count the number of cycles the player can be out of the bounds of the target before losing a life. Finally, difficulty settings are adjusted based on which button the user pressed. This changes the speed and size (for hard difficulty) of the target box as well as the amount of lives the player has by loading the respective values into the respective registers.

After initialization, the game loop begins. First, the accelerometer data is pulled into their respective registers by calling upx and upy. Then, with a series of mult, div, and adds, both the x and y positions of the player box are scaled to ensure they fit within the bounds of the screen. Following this, the aforementioned counters for both target movement and direction are decremented, and the processes for both are jumped to in the event the counters have reached 0. Finally, the position of the player box is checked against the position of the target box to see if the player box is within bounds.

To change the target box direction, a random number is generated, with the 1st and 2nd bits used to decide first the axis of movement and then the direction of movement along that axis. The box direction register is updated accordingly, and the counter for changing the direction of the box is then reset.

When updating the target box direction, the direction register is compared to constants, and then the position registers of the target box are updated accordingly with an add instruction based on the target box speed register. Within the logic to move the box, if the box hits the boundary around the screen or is out of bounds, the box's direction is changed to move away from the edge. The movement counter is reset after the box is moved, and the game loop is restarted.

During the cycles where the target box is not moved, the validity of the player's position is checked. This is done with simple arithmetic and allows for the player to be outside the bounds of the target box for a set number of cycles. If the player is within the bounds of the target, then this cycle count is reset. Otherwise, once this cycle count reaches 0, the player's lives will decrease. If the player's lives register is ever negative, the lives are forced to 0. If this occurs, the game state is set to 0, restarting the game.

Processor Modifications

For the purposes of timing, the processor runs at a clock speed of 50 MHz through the provided IP PLL.

All custom instructions are R-type instructions; therefore, modifications to the processor to support them are relatively minor. Using the existing tri-state buffer logic to modify the latch to the memory stage from the execute stage, we can easily modify what is being passed down the pipeline. Therefore, each custom instruction adds an additional tri-state buffer with the respective data, and a select bit that checks both the traditional opcode as well as the ALU opcode. Since the custom instructions all define a $rd and are R-type, bypassing works as intended without any additional modifications.

The instructions upx, upy, and diff have data passed via the wrapper to the processor. Rand pulls data from a linear feedback shift register. This is defined in the linear_shift.v file, and is written with behavioral Verilog for ease of interpretation. It defines a 32-bit register with an initial value of 1. To decide the value on the next cycle, the most significant bit is an XOR of the 31st, 30th, 10th, and 0th bits. The remaining 31 bits are set to the previous 31st to 1st bits. This register is clocked on the positive edge of the processor clock.

External Modules and Wrapper Integration

We proceed to review the main external modules used in this project: VGAController, Audio, and Accelerometer.

The VGAController is adapted from the given lab code. To build the final image, the different components are layered on top of each other. For the background, the game state is passed into the VGAController. If the game is in play, the background is simply black, however, if the game is not in play, an image background is shown. While the game is in play, the VGAController will overlay the target box, player box, and lives counter. The boxes are drawn via an offset from provided center coordinates from the register file. The lives counter is shown via sprite memory provided in previous labs.

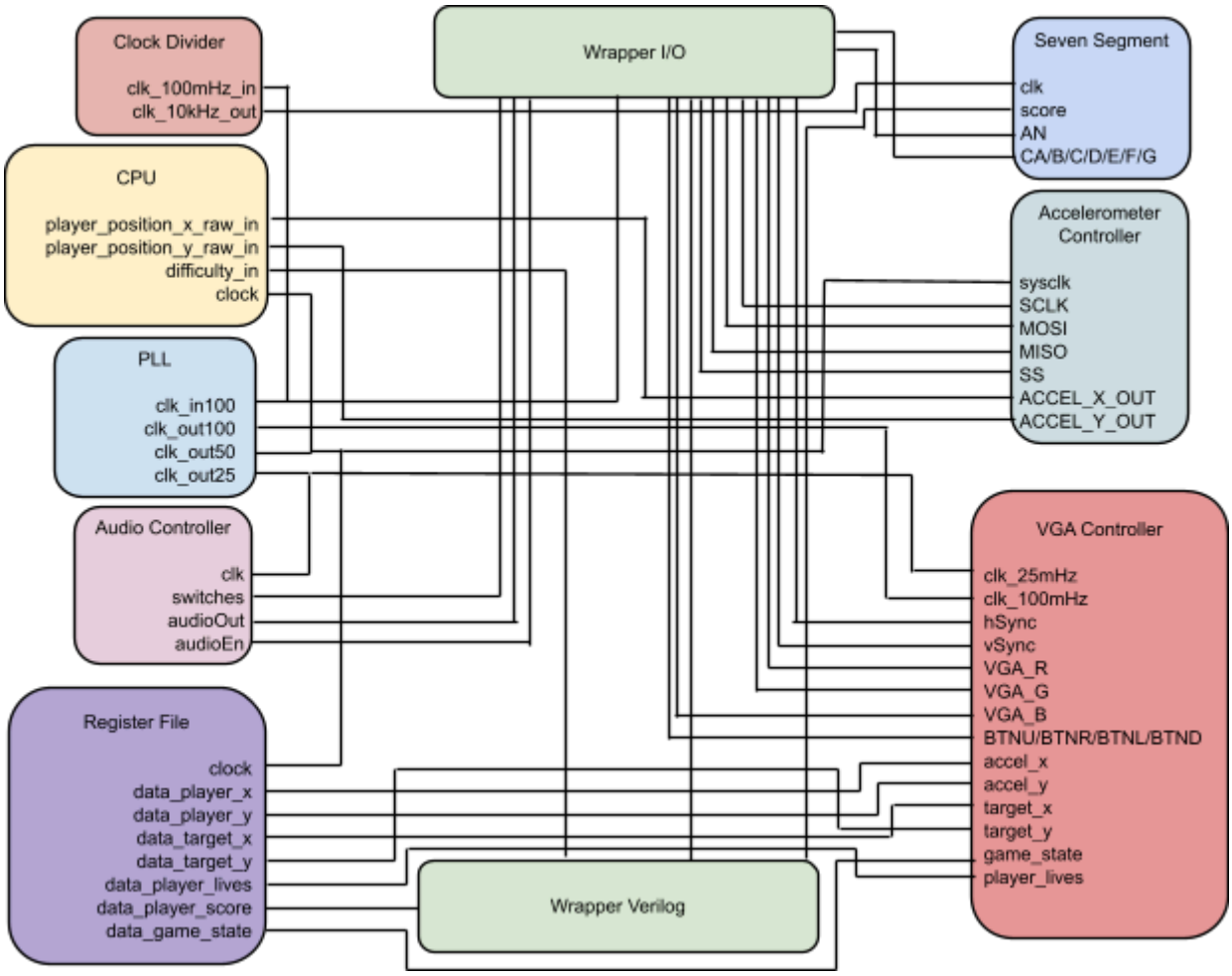
The Accelerometer data is transferred to the Wrapper file through a set of modules that communicates with the onboard accelerometer via the SPI interface, then performs scaling on the raw data to output X and Y axis readings between -1g to 1g in the form of 9-bit unsigned numbers between 0 and 511. To interface with the accelerometer, we used code from the Nexys A7 100T onboard demonstration program, which has publicly accessible code on their GitHub repository.[1] We attempted to rewrite and convert the interface files from VHDL to Verilog, although ultimately, the accelerometer data was not properly provided to the rest of the program.

The AudioController is further adapted from the code provided in the lab, the program is made to loop through the entire song in FREQS.mem on repeat. This song is inspired by Butterfly Catcher by shiru8bit[2] and is intended to be the background music playing while the game is running.

[1] https://github.com/Digilent/Nexys-A7-100T-OOB/tree/master
[2] https://www.youtube.com/watch?v=lZRcGWa0Rrw&ab_channel=shiru8bit

The SevenSegmentDisplay module utilizes the double dabble algorithm[3], also known as the shift-and-add-3 algorithm, to convert a 16-bit binary player score into 5, 4-bit values representing decimal digits. These values are then converted into BCD values then outputted to the 7-segment display.

The wrapper ties all of these elements together. For the sake of simplicity, we will ignore the provided wrapper signals and instead focus on modifications to the wrapper. Below is a diagram illustrating the movement of data among the modules.



The “Wrapper I/O” block represents the inputs and outputs to the wrapper that are handled by the FPGA board and specified in the master.xdc file. The “Wrapper Verilog” block includes small chunks of behavioral Verilog contained within the wrapper. This includes displaying the upper 16 bits of score to the LEDs as well as button presses to the difficulty number logic.

[3] https://en.wikipedia.org/wiki/Double_dabble

Hardware Components

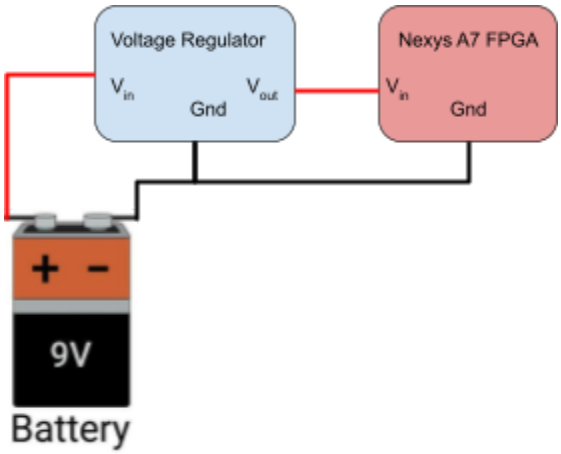
The external hardware components, in addition to the FPGA, VGA Screen, Speaker, and interconnecting cables, include the following:

- 1x 3D Printed Housing
- 1x 9-volt Battery
- 1x 5-volt S13V30F5 Voltage Regulator
- 1x Barrel Plug Cable

The housing was designed on Fusion 360 and 3D printed on an Ultimaker S3. This housing is the primary structure with which a user will interact with the project, and it contains all other components, with only a VGA to Mini USB connecting the FPGA inside to an externally placed VGA Display.

The housing consists of a bottom and a top lid. The bottom lid holds the FPGA and a voltage regulator, attached via hot glue, and it also holds a 9-volt battery, attached with velcro for ease of access. The top lid holds the speaker, also attached with hot glue. For all components with extended wires, the wire management is done with electrical tape. Popsicle sticks are also attached with hot glue to ensure the two lids mesh well together and do not slide alongside each other while the user is playing the game.

Electrically, the 9-volt battery is wired to the voltage regulator, which steps down the battery voltage to 5 volts. This regulator is then wired to a barrel jack and can supply 3A, enough to power the FPGA through the onboard plug.



Test Plan

During the workflow of the project, changes were tested first in the modules themselves, and then together as a unit through the wrapper. When possible, assembly was tested through the provided autotester.py file. However, as described previously, many data signals are passed through the Wrapper, so a majority of the testing had to be done with the complete product, presenting challenges.

Upon completion of the product, a simple test procedure is as follows:

1. Re-initialize the game completely through a bitstream program.
2. Ensure the game loads into the start screen. None of the other game elements should be present. Ensure that the score LEDs indicate a score of 0. Ensure the score segment display is empty.
3. Choose the easy difficulty.
    a. Ensure the target box, player box, and lives counter are present on the VGA.
    b. Ensure that tilting the controller results in the corresponding movement of the player box.
    c. Play the game as intended, and ensure that the lives do not decrement, and the score LEDs are incrementing.
    d. Ensure that, during gameplay, the target box moves in a random manner.
    e. Allow the player box to deviate from the target box. Observe the lives on the VGA screen decreasing in a constant manner.
    f. Realign the player box with the target box before the lives counter reaches 0. Observe that the lives counter no longer decrements, and observe the score LEDs resume incrementing. Observe the score segment display incrementing.
    g. Allow the player box to once again deviate from the target box. Let the lives counter reach 0, and observe that the game returns to the start screen.
    h. Ensure that the conditions from step 2 hold once again for the VGA. Observe that the score LEDs should still represent the score obtained during the previous playthrough.
4. Choose the medium difficulty
    a. Ensure that the target box moves at a higher speed than the easy difficulty, and the number of lives given initially is decreased.
    b. Follow the same testing procedure as step 3.
5. Choose the hard difficulty
    a. Ensure that the target box moves at a higher speed than the medium difficulty, and the number of lives given initially is decreased.
    b. Ensure that both the target box and player box have decreased in size.
    c. Follow the same testing procedure as step 3.

We have followed this test procedure and can confirm that our project conforms to the standards that we have set.

Challenges and Future Improvements

We faced challenges in various aspects of the project. The following challenges are ranked in terms of time spent resolving or working through the challenges.

1. The VGAController proved extremely challenging and difficult to resolve issues with. We quickly learned that the timing on all components in the file was prone to errors with even the slightest changes. While this issue was present everywhere, the VGAController exhibited this issue most prevalently as it contains a large amount of behavioral Verilog and decision logic. For example, we have examples where a select bit to a 2-input mux is the result of 2 layers of AND gates. Changing the AND gate from a 2-input to a 3-input AND gate would cause the game to freeze. This is likely due to an issue with stability, as the VGA works through the different x and y values.
2. Debugging any issues as the project came together proved to be difficult or nearly impossible. The autotester could be used to test the processor with constant inputs where there would normally be dynamic values from modules like the accelerometer. However, past this, the autotester proved to be limited without major modifications to the wrapper testbench. Thus, a majority of our debugging was completed with the game at scale. This required 20-40 minutes of synthesis runs for each minor, incremental change we wanted to test or debug. In addition, while we were aware of the ILA function of Vivado, we had issues implementing it.
3. Github and version management also proved to be a challenge. Initially, we worked together through VS Code Live Share, editing in real time. However, as scheduling became increasingly complicated, we began to work on separate branches. These branches all had sets of changes and often had varying states of functionality. Thus, merging proved to be a challenge, as well as keeping up with the changes in Vivado. A sample visualization of the repository can be found in the appendix under Figure B.
4. When creating the custom song and inputting desired frequency values into the memory file, there were certain notes that would not play correctly. These notes include F3, which seemed to simply be too low in frequency for the program to generate a proper signal to send to the speaker. This was resolved by upscaling every note upwards by 1 octave, although troubleshooting this problem in conjunction with other audio bugs was a challenge.

Given more time, we would implement the following features, broken down into functional categories, listed in no particular order.

Software

- A leaderboard could be added for high scorers. This would use the ram to load and store members on the leaderboard.
- More backgrounds through RAM optimizations, as we are missing a background for the game loop state.
- Overall cleanup/optimization of behavioral Verilog and assembly logic to help with timing and readability.
- Planning for the timing of various modules (i.e. VGAController) to allow for the addition of extra features.

Hardware

- Better UI for controller box for ease of use, which may include adding handles and buttons that fit into the housing.
- Better visibility of the score to enhance user experience.

- Make the controller box entirely wireless.

General

- Use the phone to control the player icon via phone's gyro and bluetooth modules.
- Audio improvements through sound effects and music synchronization.

Appendix

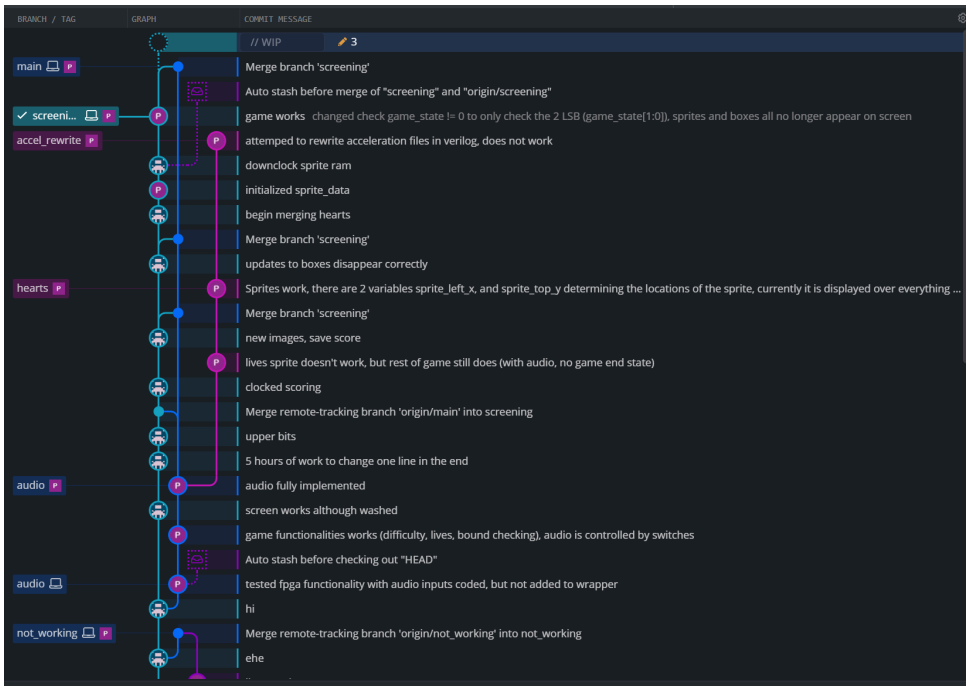


Figure B: GitKraken Visualization of Repository

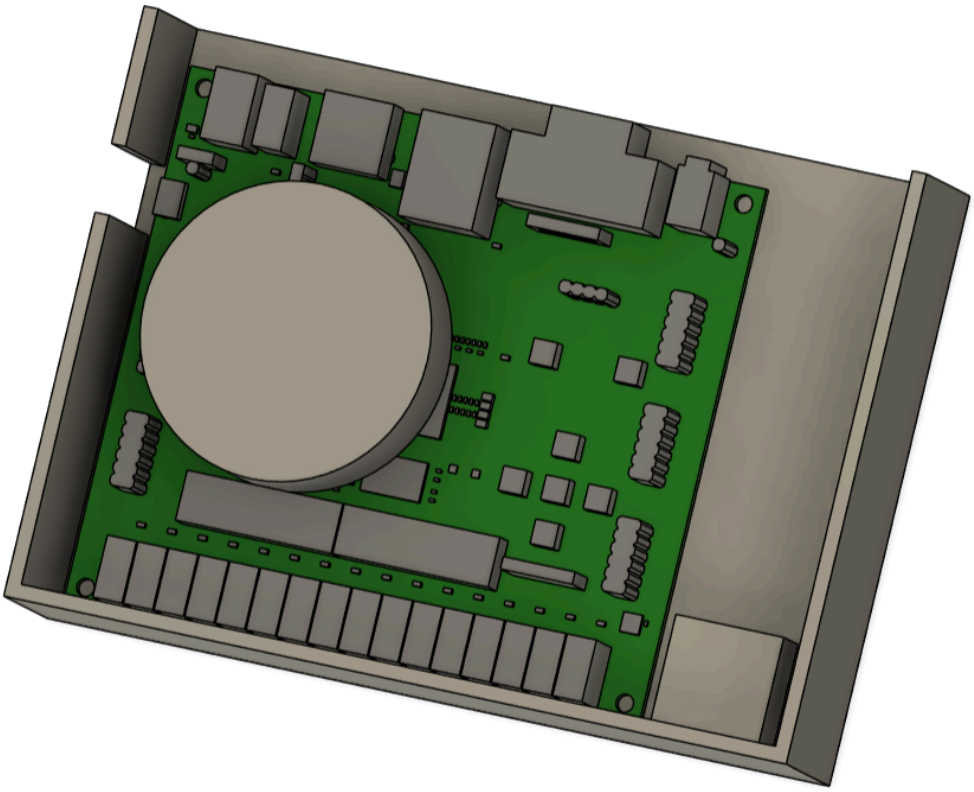


Figure C: CAD Assembly Illustrating Size Estimates of FPGA, Speaker, and Buck Converter

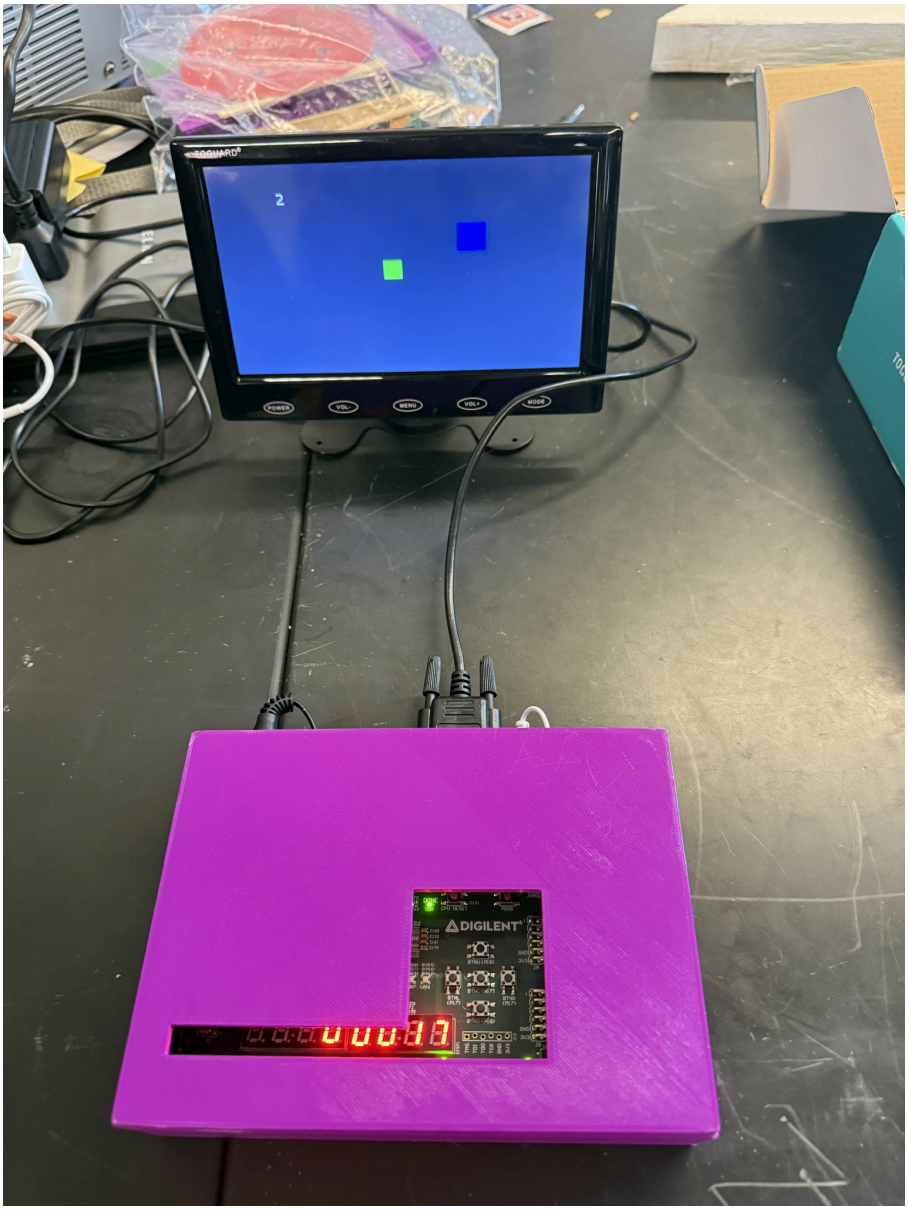


Figure D: Game in Progress with Visible Score