Group Report:

EEG Brick Breaker

ECE532

Group 11:

Natalia Chelmecki

Marion Jan

Tian Lan

1.0 Overview

For the project of this course, our group decided to implement a Brick Breaker game on FPGA. Our initial objective was to control the game with an EEG headset. Unfortunately, we were unable to borrow one in time for the project. Therefore, we carried on our contingency plan to get a working demonstration by the end of the project deadline, and we replaced the EEG headset with a joystick Pmod from Digilent.

Our Brick Breaker game consists of bricks, a horizontal paddle bar and a bouncing ball, as shown below:

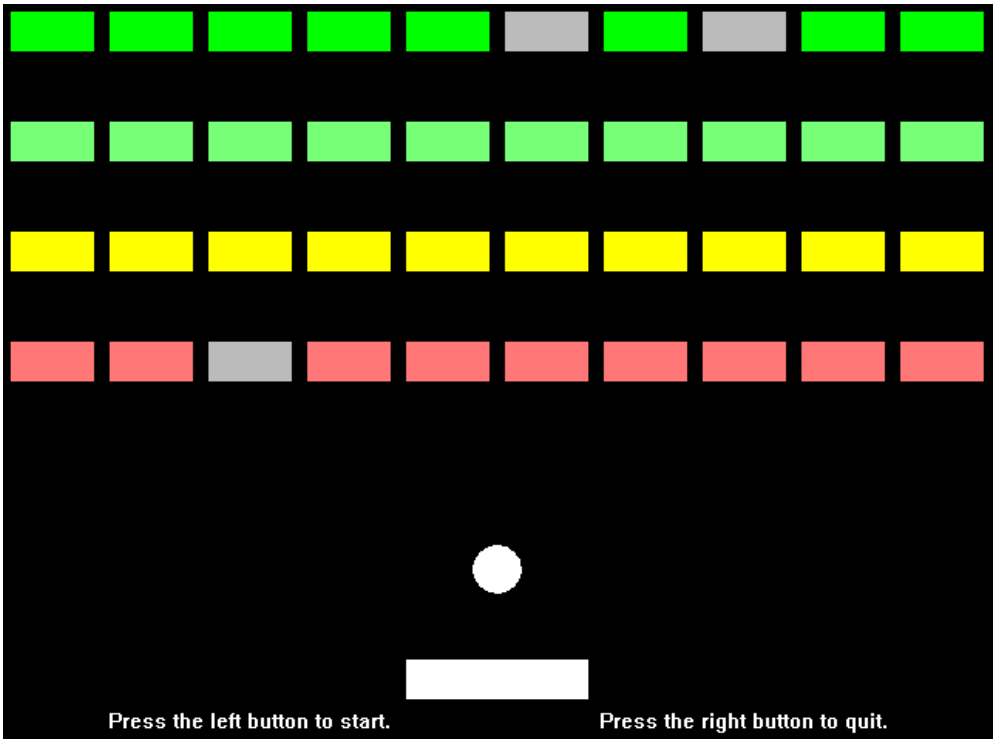

Figure 1: The graphics of our game.

The game's objective is to destroy as many bricks as possible with the ball, and the player can use the joystick to control the right-left movement of the paddle bar. All bricks have to take several hits to get destroyed. If the ball touches the bottom side of the screen, the player loses a life. The game also features three power-ups that the player can collect and use:

- Fireball: For 10 seconds, the bouncing balls will move straight through and destroy any bricks.
- Extra life: The player gets one more life to lose before the game is over.
- Extra balls: Spawns two additional bouncing balls on the screen.

Since we couldn't realize the EEG functionality, we made up for the complexity score of our project by adding a Bluetooth module. We hope that by doing so, we can play music over Bluetooth earphones as background music for the game. But the Bluetooth module ended up serving a different functionality in the latter parts of our project: It is used to control the play and stop of the game music (played over a conventional audio port), with the control signals sent by a smartphone over Bluetooth.

Another hardware functionality we implemented is to display the player's score and remaining lives over the embedded 7-segment display of the FPGA. And the two buttons on the joystick were used to start, end and exit the game. Since the game graphics are mainly rectangular shapes, we chose VGA instead of HDMI or UHD to display the game, as VGA's lower definition is sufficient for the needs of our game.

Our motivations for doing this project were that none of us knew wireless hardware well, and we wanted to learn more about it in the process of realizing the Bluetooth connection for our system. We were also very interested in using EEG to control our game because it was new for us and probably new for most students in this course as well.

2.0 Outcome

2.1 Results

For this project, we met most of our functional requirements, and as such, we consider the project a success. Two of our three requirements were met, and the remaining one was tentative. The requirements are summarized below:

Table 1: Functional Requirements

Functional Requirement	Status
The game runs properly using a VGA display	Complete
A joystick can be used to start/stop the game	Complete
The player can use an EEG device to control the game (tentatively)	Incomplete

Our completion of the third functional requirement was dependent on actually receiving an EEG sensor. Due to the scarcity of EEG headsets at the University of Toronto, we were unable to secure one. Halfway through the semester, we purchased a headset, but it was delivered a few days before the demo, and we were unable to integrate it into our system.

We anticipated being unable to locate an EEG sensor and designed two versions of the potential system, one without an EEG. The non-EEG project consists of the following: A Brick Breaker game functioning identically to the classic with additional power-ups, as introduced in the Overview section.

The game is displayed on a monitor using a VGA display. The joystick controls the movement of the bar, where pushing on the joystick with more force moves the paddle bar faster and vice versa. The buttons on the joystick also control starting/ending of the game. The score is displayed on the 7-segment displays, along with the number of lives the player has left.

Unfortunately, we were unable to merge some of our major components, and as such, we have two projects working separately. The description above is the first system, and the second system is a Bluetooth-controlled audio player. The Bluetooth module polls an Android smartphone for input. The player can start/stop the song or choose between songs on the phone, and the songs are played through the audio port on the Nexys 4 DDR.

The block diagram of our intended system design is shown below:

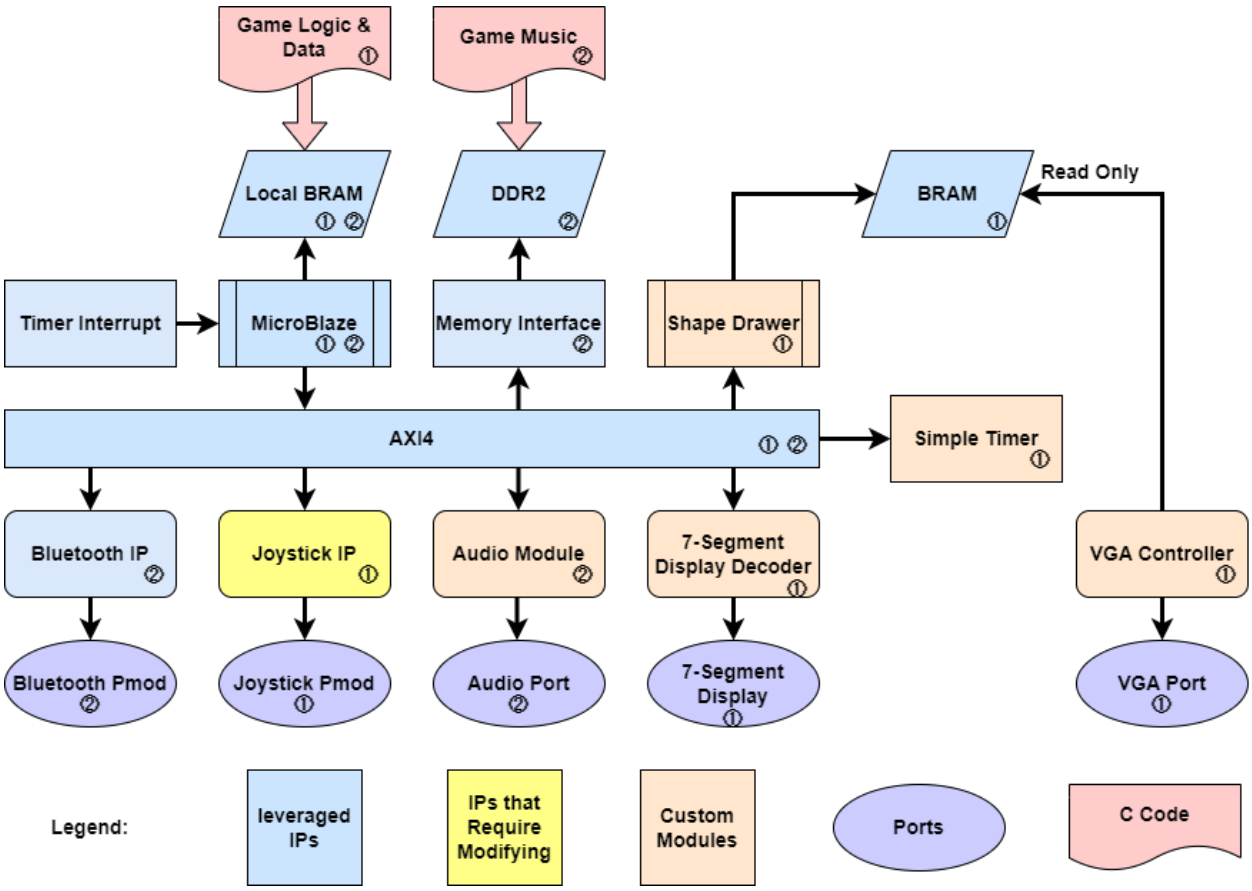

Figure 2: Our intended system design. The blocks with ① are only functional in the first system. The blocks with ② are only functional in the second system.

2.2 Further Improvements

To improve our system, it is obvious that the two "half-systems" should be merged so the game and its music can be played simultaneously. We should also have ordered our own EEG headset from the start, that way, we would have had some time to try and integrate it.

The game requires no modification (in our opinion). However, some new peripherals could be added to make the game more interesting, for example, replacing the VGA display with a VR headset, 3D printing a case for the joystick so it could be held away from the FPGA board, or adding a wifi module that would allow players to compare high scores.

3.0 Project Schedule

Table 2: Our Original Milestones

Milestone Number	Milestone	Date	Description
1	Game Logic	Feb 2	The game logic will be partially completed.
2	Functional Joysticks	Feb 9	The FPGA will accept input from the joysticks. When the joysticks are pressed, an LED will light up on the FPGA. This will ensure the joysticks are functional for game input.
3	Functional Mouse	Feb 16	The USB Mouse will be functional and tested in a similar way to the joystick.
READING WEEK + DEMO		Feb 23	
		Mar 2	
4	VGA + Game Logic Completed	Mar 9	The game logic will be completed so that the peripherals can be connected and tested in future milestones. The VGA will allow the game to be displayed on the monitor.
5	Bluetooth Functional	Mar 16	The Bluetooth module will be completed, allowing notes to be played on wireless headphones.

6	Peripherals Connected, Game Completed, (Optional EEG Sensors)	March 23	With the game logic completed from milestone 4, all the peripherals will be connected to the game so that the game is fully functional. The game will be tested and we will attempt to connect the EEG sensors.

The actual execution of our plan is listed here:

Week 1: The collision detection part of the game logic was done. The VGA controller was almost done, too, and we started working on the joystick. Week 1 goals have been achieved on time.

Week 2: The joystick was functional as planned in the original proposal. The 7-segment display was used instead of LEDs to test the joystick IP on the FPGA. A rotating colours module has been created to complete the VGA controller, and the game logic has been completed.

Week 3: We displayed a menu on the screen over VGA, but the shapes were drawn by manually setting each pixel, and that will have to be changed in the future. We also started to work on the Bluetooth IP. These goals didn't match the original proposal because we were not using a USB mouse anymore.

Week 4: The VGA controller was fully functional, as wanted in our proposal. We also created a sub-module of the VGA controller that reads the contents of the frame buffer BRAM. We were now able to create different frames: Initialization, Running Game, Lose and Win. We also created the 7-segment display outputs that name the current frame on the screen. Eventually, Bluetooth IP was completed and transferred data from an Android phone to the SDK terminal. The game logic was completed. Our work matched our expectations for this week.

Week 5: We added a Binary to Binary Coded Decimal converter to display the score and remaining lives on the 7-segment display. We also created the shape drawer module that is capable of drawing rectangles and circles following the commands of C code running on MicroBlaze. And finally, we started the integration of the different modules by packaging the Joystick IP and creating the AXI + MicroBlaze architecture. We also modified our plans from playing notes on wireless headphones (as in our proposal) to playing notes on the audio port of the FPGA board, and the new plan has been completed. Although we were on time for our

proposal deadline, we started to realize that we didn't have a lot of time to work on the integration of the project.

Week 6: Concerning the integration of the project, we connected the MicroBlaze with the 7-segment display. Unfortunately, we discovered that the logic required by the graphics modules is hard to make synthesizable. Putting AXI, the custom audio module, and MicroBlaze together was too much work to finish in one week, too. We were behind the goals of this week.

In conclusion, we almost respected the deadlines we fixed in our proposal, but the proposal was not realistic because one single week to work on the integrations of the modules together was not enough and was very stressful.

4.0 Description of the Blocks

4.1 Game Logic

To implement the Brick Breaker game, we implemented the game's "physics model" in C, and it runs on MicroBlaze. Our game features two basic types of 2D objects, circles and rectangles, each represented by a type of C struct. Two types of interaction, collision (touching) and elastic scattering, can happen between these objects. A circle in our game can collide and scatter with another circle or a rectangle, while the rectangles in our game never touch each other, so they don't interact with themselves.

In calculating outcomes for elastic scattering, our game treats circles as massless objects. And in most cases, the outcomes for elastic scattering in our game are accurate in Newtonian mechanics, including when the circles collide into the corners of rectangles.

The main loop of our game runs at 60 Hz, and in every loop cycle, it applies the physics model introduced above to all objects in the game. The main loop also uses a Simple Timer (connected to our system with AXI), such that if it takes less than 1/60 s to run a cycle (by far, it's always the case), it will wait until the time is up. That way, our game can have a stable frame rate of 60 Hz.

In every loop cycle, our game polls our Joystick IP to determine the horizontal velocity of the paddle bar (controlled by the player). The two buttons on the joystick Pmod also control the starting, ending and exiting of our game, as shown in Figure 1.

As for the game mechanics: When a ball hits a brick in our game, the brick loses health. If the health of a brick reaches 0, it will be destroyed, and the player will get one score. (The score will be displayed by our 7-Segment Display IP.) The remaining health of a brick is reflected in its colour: The bright green bricks have the highest health, the dark red ones have the lowest, and the grey bricks are indestructible.

There is a 1/5 chance that a power-up will drop when a brick is destroyed. The power-ups are rainbow-coloured balls that the player must catch using the paddle bar to come into effect. The types of power-ups are introduced in the Overview section.

If the player manages to destroy all destructible (i.e. non-grey) bricks on the screen, all bricks will be regenerated, with each brick having a 1/10 chance of being indestructible. That way, our game is not winnable: You can keep playing the game until you quit or lose it.

4.2 Simple Timer (Available in our GitHub repo.)

We made a simple timer with an AXI Slave interface. It increments its counter every clock cycle, and writing 0 into its counter register can reset it. The MicroBlaze can poll it to determine the time elapsed since its last reset. We used this timer to stabilize the frame rate of our game.

4.3 Joystick IP (Available in our GitHub repo.)

To use the joystick, a Digilent Example Project was referenced, where Verilog sample codes were provided. It can be accessed at [1]. (Note that the reference link to the code does not exist on the Digilent website any longer, but the code headers indicate it was originally written by Digilent.) Minor modifications had to be made to the code so that the button presses and joystick data could be sent in one output.

The joystick IP communicates with the joystick Pmod over a serial port at a clock frequency of 66.67 kHz. And we added an AXI Slave interface to the joystick IP for it to communicate with our MicroBlaze. It stores the button presses and joystick position data in its registers for MicroBlaze to poll when needed. The range of the possible joystick data is 0 (for right-most) to 1023 (for left-most).

4.4 Frame Buffer

We used BRAM blocks on the FPGA to create a display frame buffer for our system. Our system has only one frame buffer (i.e. no multiple buffering), and we implemented it with a Xilinx Block Memory Generator IP, operating under simple dual-port mode. The writing port of

the frame buffer connects to our Shape Drawer IP, and the reading port of the frame buffer connects to our VGA Controller IP. We will describe both of these custom IPs below.

The frame buffer stores 640*480 = 307200 pixels. And the colour of each pixel is represented by 13 bits of data. The lowest 12 bits of a pixel store the RGB values of its colour, with 4 bits to store each prime colour, respectively. The 13th bit of a pixel means "rainbow": If it is a binary 1, then the colour of this pixel will rotate around the colour wheel. The colours a rainbow pixel will pass through are shown in the figure below (from left to right), and it takes 3 seconds for it to wrap around and become red again. Our VGA Controller IP is in charge of realizing this rainbow functionality, more on that later.

Figure 3: The changing pattern of the "rainbow" pixel's colour.

As our TA pointed out, the implementation of the rainbow functionality proves to be a strategic mistake. While it can produce some cool-looking graphics, it also takes up 37 kilobytes of BRAM space, and we are very tight on BRAM space in the latter parts of our project.

During the programming of the FPGA, the frame buffer will be loaded with an initialization file containing the pixel pattern of two text strings: "Press the left button to start." and "Press the right button to quit." They will show up near the bottom side of the screen, as shown in Figure 1, and they will stay there until the game finishes running (i.e. until the player chooses to exit).

4.5 VGA Controller (Available in our GitHub repo.)

We need a VGA controller to display our game over a VGA monitor, and we created one following the VGA specifications [2], using a mixture of VHDL and Verilog. It works autonomically and is not subject to external control except for the reset and clock signals.

Our VGA controller has one line counter and one column counter, with 10 bits width each. They change incrementally to scan through the Active and Inactive Areas, row by row. When the line counter and the column counter have values corresponding to coordinates in the Inactive Area (not visible on the screen), our VGA controller will set the appropriate synchronization signals (H Sync and/or V Sync) to low on the VGA port, as shown in the figure below:

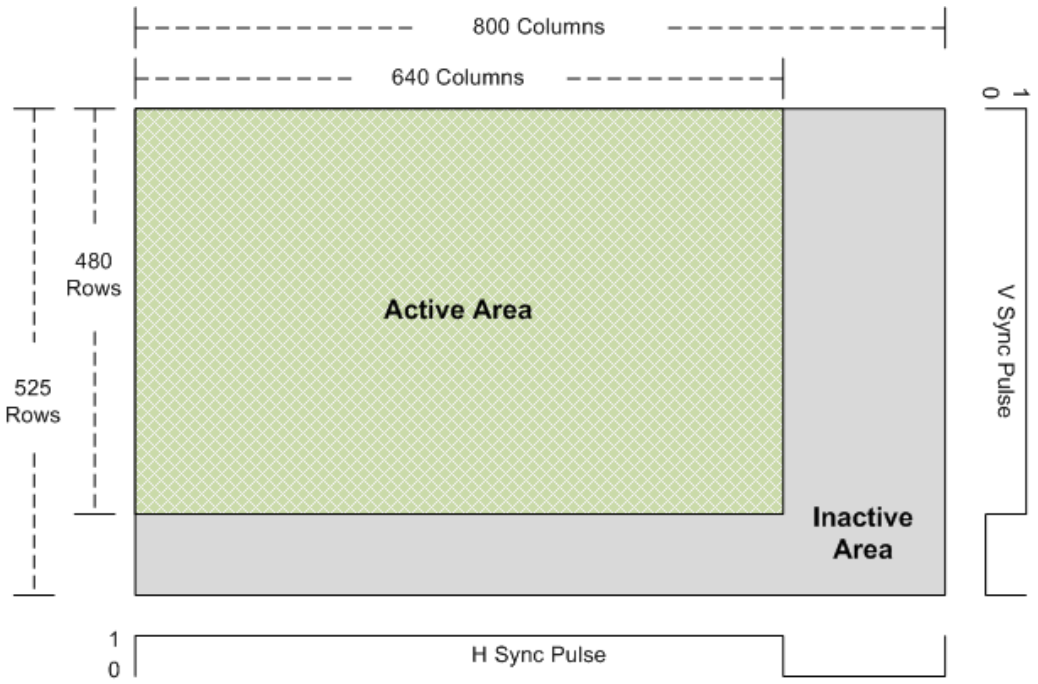

Figure 4: The Active and Inactive Areas in VGA specifications.

When the line counter and the column counter have values corresponding to coordinates in the Active Area (visible on the screen), our VGA controller will read directly from the Frame Buffer through the latter's reading port. It will compute and output the reading address according to the current pixel's x and y coordinates on the screen, and the pixel's data will arrive from the Frame Buffer 2 clock cycles later. Since the rated frequency in the VGA standard is 25 MHz, we distributed the reading and other operations of the VGA controller into four clock cycles (for each pixel).

After the VGA controller receives a pixel's data from the Frame Buffer, and if the data indicates the pixel should be "rainbow" (i.e. if its highest bit is 1), the VGA controller will pick a colour from a colour lookup table based on a counter (the lookup table is shown in Figure 3 and is implemented using distributed RAM to save BRAM space). The picked colour will then be sent to the VGA port, and the lower bits in the pixel's data are ignored. Otherwise, if the pixel is not a rainbow one, the original red, green and blue components in the pixel's data will be sent directly to the VGA port.

4.6 Shape Drawer

The C code of our game (which runs on MicroBlaze) does not access the Frame Buffer of our system directly. Instead, we created a shape drawer IP to set the pixels' bit values in the Frame Buffer. That way, MicroBlaze does not need to take time to set the colour of every pixel, and it can do other work when the shape drawer is doing that.

Our shape drawer IP supports an AXI Slave interface and can draw two shapes, circles and rectangles. Before asking the shape drawer to draw a circle or rectangle, MicroBlaze needs to set the shape's colour over AXI. The desired colour must be written into a control register at the address offset 0x10. To draw a rectangle, the shape drawer also needs the x values of the rectangle's left and right edges and the y values of the rectangle's upper and lower edges. These coordinate values must be written into their corresponding control registers at address offsets 0x00, 0x08, 0x04 and 0x0C, respectively.

As for circles, our shape drawer IP only supports drawing perfect circles with a fixed diameter of 33 pixels. We designed our shape drawer this way because our game features no circles of other diameters. This design also simplified the shape drawer, as instead of computing the pixel pattern of a circle on the spot using the Pythagorean theorem, we can store the pixel pattern in a lookup table (which is implemented using distributed RAM to save BRAM space). And to draw a circle, our shape drawer only needs the coordinate of the upper left corner of the square bounding the circle, written into its control registers.

When our shape drawer IP draws a shape, it writes directly into the Frame Buffer through the latter's writing port, at a rate of one pixel per clock cycle. Since the shape drawer does not support drawing letters or numbers, we decided to use the 7-segment displays on the FPGA board to show the player's score and remaining lives.

4.7 7-Segment Display (Available in our GitHub repo.)

In the beginning, we used these displays manually, turning on and off the anodes and cathodes to control the segments one by one. With this method, we were able to display constant letters. Then we decided to show the player's score and remaining lives, which are changing with the game. For this reason, we added a binary to the BCD converter in our project to display any number. The algorithm we used for the converter is called the "double dabble" algorithm [7]. The original algorithm we found does not use a clock signal and uses at least 12 blocking assignments in one always block. Therefore, we modified the algorithm to distribute its operations into 24 clock cycles.

The eight digits of the two 7-segment display components on the FPGA board share seven cathodes, one for each segment (shared by that segment across all digits), as shown in the figure below. And the eight independent anodes act as "digit select," akin to mux select. So, to display two numbers separately over the two 7-segment display components, our 7-segment display IP would alternate between each number, charging their corresponding anodes at different times. That way, our system charges the anode of each of the eight digits at 1 kHz.

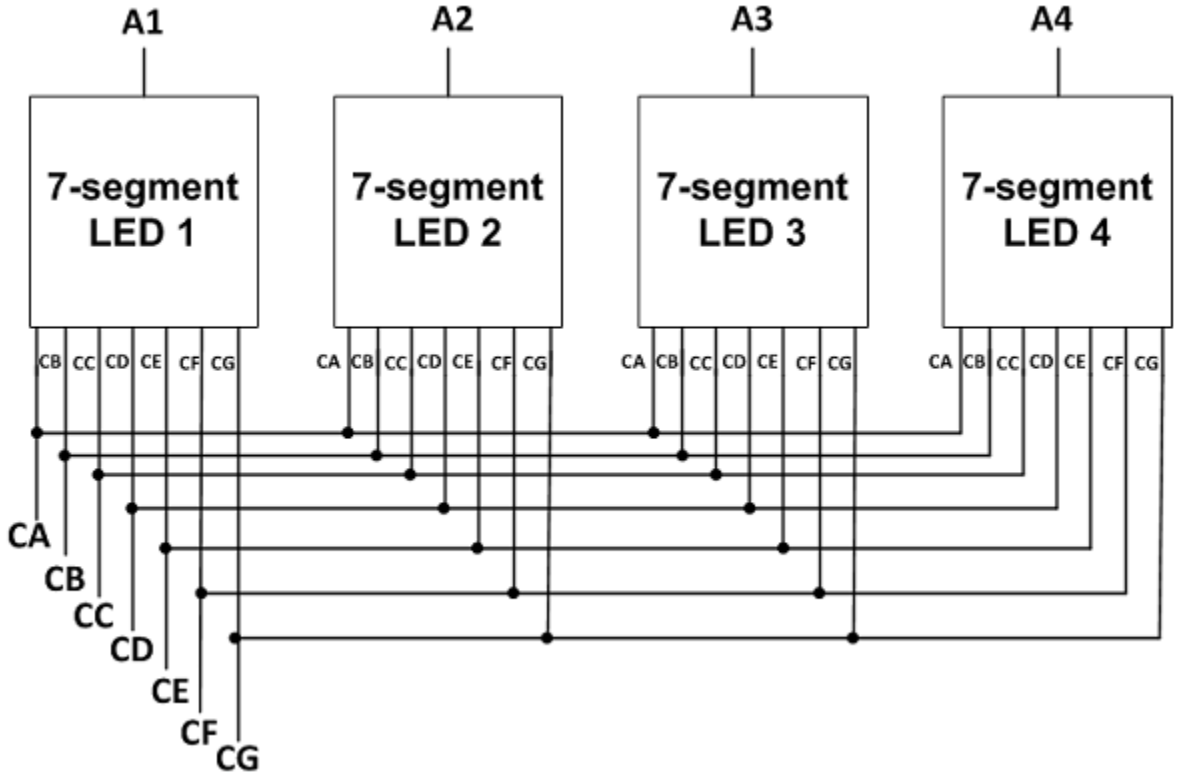

Figure 5: The anodes' and cathodes' connections in 7-segment display components.

Our 7-segment display IP receives the two numbers to display over AXI, with each number stored in one control register. The maximum number it can display correctly is 9999 for both numbers. It also trims any non-significant digits at the beginning of each displayed number. So, instead of displaying "0000" and "0003," it will display "0" and "3," with the higher digits left blank.

Last but not least, the 7-segment display on our FPGA board will show "you LOSE" if the game is over.

4.8 Bluetooth IP

The Bluetooth module has a manufacturer IP with AXI interfaces available for use here [3]. This was used in our block design along with the standard Uartlite IP provided in Vivado. The Uartlite IP's baud rate had to be modified to properly decode the data from the Bluetooth Pmod. For the Bluetooth Pmod we used, the appropriate baud rate is 115200.

We used our Bluetooth module to receive commands from a smartphone, and the commands will control the playing and switching of the game's background music. We will discuss more on that below.

4.9 Music Module

The music module performs pulse width modulation (PWM) on an incoming byte of data and is a semi-custom IP. It accepts a value sent from MicroBlaze via AXI and modulates it at 100MHz. Every 44.1kHz [see Appendix A], a counter is incremented. The counter is polled by MicroBlaze to determine when to send a new byte of data to be played. The PWM code came from [4], and the clock divider came from [5]. The module is capable of playing any song/sound effect, as well as two sounds at the same time (e.g. a song and a sound effect of the game).

We attempted to make MicroBlaze work with the music module using interrupts instead of polling, so that the C code of our game could run in parallel with playing the game's song/sound effect. We used a timer interrupt that triggers at 44.1 kHz to invoke the code required for playing the sounds, and it did not end up working. Should we have more time, we would examine the stack of our code to determine the cause of the problem.

For the music module to work, the music files also had to be converted to a raw format and stored in DDR, as there was not enough BRAM to store the files. The MicroBlaze, after sending a song byte to the music module, will move to the next byte of the song and send that to the music module next. The music files were C header files containing an array of bytes. This [6] hex reader was used to generate the files in the correct format. To convert a song to the correct format:

- Highlight the bytes to be included
- Right-click
- Click "Export selected bytes as code snippet"

4.10 Customization of MicroBlaze

Last but not least, we customized our MicroBlaze to suit it better with our system. We enabled the hardware implementation of the square root function inside MicroBlaze to speed up

the code of our game, for it uses the square root function in its physics calculation. To save BRAM space, we put the Branch Target Cache into distributed RAM, and we dedicated only 1KB to the Data Cache of the DDR2 RAM.

Since we know that our system accesses the DDR2 RAM (containing sounds) completely sequentially, we also set the Cache Length of MicroBlaze to 16 (the maximum value available).

5.0 Description of the Design Tree

The GitHub repo is located here. The file tree description is available in the README.

6.0 Tips and Tricks

For future students, we would like to share the three lessons we learned in doing this project:

- Dedicate at least two weeks to integrating your project when planning your milestones. We could not get our modules integrated in the final week, which was the greatest setback we met in the project. Our TA also told us halfway through the project that integration would be hard, and that prophecy was fulfilled.
- Do not create a complicated application in C unless you are planning to use high-level synthesis. This course does not grade your C code, and it only grades your hardware. We created a game with a complicated physics model and 34 kilobytes of code, which was a big waste of time in hindsight.
- Pay attention to your BRAM usage if you are doing a project related to images or displays. You can put one frame buffer or image of size 640*480*12 (bits) into BRAM, but that will consume most of the BRAM blocks on the FPGA and could make placing other modules (including MicroBlaze) difficult. To avoid that, you can consider putting your frame buffer into DDR2 instead (and you must use DDR2 if you need multiple buffering).

References

[1] Commanderfranz, “How to use a joystick with an FPGA,” *Instructables*, 01-Oct-2017. [Online]. Available: https://www.instructables.com/How-to-Use-the-PmodJSTK-With-the-Basys3-FPGA/.

[2] Digilent, “VGA display controller,” *VGA Display Controller - Digilent Reference*. [Online]. Available: https://digilent.com/reference/learn/programmable-logic/tutorials/vga-display-congroller/start.

[3] Digilent, “Digilent/Vivado-Library,” *GitHub*, 12-Nov-2021. [Online]. Available: https://github.com/Digilent/vivado-library.

[4] E. F. Y. Bureau, “Implementation of a simple PWM generator for microcontroller using Verilog,” *Electronics For You*, 03-Jun-2022. [Online]. Available: https://www.electronicsforu.com/electronics-projects/pwm-generator-microcontroller-verilog.

[5] “Verilog code for clock divider on FPGA,” *FPGA Projects, Verilog Projects, VHDL Projects - FPGA4student.com*. [Online]. Available: https://www.fpga4student.com/2017/08/verilog-code-for-clock-divider-on-fpga.html.

[6] J. Duttke, “Browser-based online and offline hex editing,” *HexEd.it*. [Online]. Available: https://hexed.it/.

[7] Real Digital, “Binary to BCD and BCD to Binary,” *realdigital.org*, n.d. [Online]. Available: https://www.realdigital.org/doc/6dae6583570fd816d1d675b93578203d.

Appendix A

Due to the (possibly) irregular output of the clock divider, an additional value was sent from MicroBlaze to the music module. The music module accepts the value as the *divisor,* which allows the song to be played faster or slower than 44.1kHz, by dividing the clock input by a different, user-specified number. This also allowed us to experiment with the sounds generated and tinker with the output so it sounded as close to the original as possible (which was especially important when playing songs that had originally been sampled at rates other than 44.1kHz).