Project Report:   
FPGA Video Game

**Author: Dale Andreoli**

**Date: 20 December 2021**

**Class: ECEN 397**

Contents

[Purpose 3](#_Toc90927891)

[Methods and Procedure 4](#_Toc90927892)

[Video Output 4](#_Toc90927893)

[Audio Output 4](#_Toc90927894)

[Controller Input 4](#_Toc90927895)

[Game Design 5](#_Toc90927896)

[Results 6](#_Toc90927897)

[Video Output 7](#_Toc90927898)

[Video Output – Testbench 10](#_Toc90927899)

[Renderer 10](#_Toc90927900)

[Controller 11](#_Toc90927901)

[Paddle 11](#_Toc90927902)

[Ball 12](#_Toc90927903)

[Collisions 13](#_Toc90927904)

[Sound 13](#_Toc90927905)

[Bricks 13](#_Toc90927906)

[Game State 14](#_Toc90927907)

[Conclusions 15](#_Toc90927908)

[References 16](#_Toc90927909)

# Purpose

This project demonstrates knowledge and skills expected of a senior-level Computer Engineering student by using a Field Programmable Gate Array to implement a video game. The project uses an FPGA to display video, output sound and take input from a game controller while also handling the game logic.

Many components of this project required significant research and testing to implement. Fortunately, each component was well documented and with proper research, I was able to successfully bring them all together into the final working device.

As a Computer Engineering student, gaining further knowledge and skills relating to FPGAs has been valuable as some of my potential career paths involve hardware description languages. This project provided new challenges to overcome as I researched and implemented its various components. This has increased my competence with hardware description languages and demonstrated an ability to learn new skills through research.

# Methods and Procedure

This project uses a Basys 3 Artix-7 FPGA to handle input, output and game logic. This FPGA has an on-board VGA output. The video was output to a computer monitor at 1080p resolution at 60Hz refresh rate. The audio was output from a buzzer that makes a beep when given a high signal from one of the Pmod headers on the FPGA. The game is controlled by a Nintendo Entertainment System controller which connects to several of the Pmod headers on the FPGA to send button input data. Game logic is handled by hardware designed in Verilog then programmed on to the FPGA.

## Video Output

I did some research into other video output methods such as HDMI and DVI. They would have been possible but HDMI was a bit more complicated to implement than VGA. DVI also was less convenient since my FPGA already has a VGA out connector built in.

Implementing this component was aided by documentation of timing values for this output specification. Understanding the function of a pixel clock and the various sync regions used in VGA output was necessary to get this component working. A testbench for this module was invaluable as I had to debug several errors along the way. Research, development and debugging for the video output component took about 14 hours. My plan to test the VGA module was to assign each pixel a color (they all had to be the same color since I did not have a rendering module yet). I could then hook the device to my monitor and verify that it was outputting the proper color.

## Audio Output

There were many options I found in my research for handling audio output. My favorite was a little soundboard that you can upload audio files onto then send a signal to various pins on the board to select a sound to be played. I couldn’t get my hands on one for this project, but it would be a cool upgrade to make in the future.

I went with a buzzer that makes a beep when sent a signal. I would have liked something more versatile, but this is what I already had in my parts drawer. Research and testing for the audio module only took about 7 hours total. To test this module, my plan was to send the buzzer a signal and verify that it made a beep.

## Controller Input

I researched a few different controllers that would have worked for this project. Most of them used USB, which would have worked just fine. I went with the NES controller that had a lot of documentation and seemed very straightforward. It also went with the retro feel of a more primitive style of video game.

The NES controller sends all of its button values through a single data pin, which can be read with the help of a clock signal and latch signal that go into the controller. This required research to get the proper timing for the signals (bertiusgames). An official vintage NES controller actually requires 5 V to operate, where my FPGA has mostly 3.3 V pins. Fortunately, my controller is a modern off brand that supports 3.3 V operating voltage. Research and testing for the controller input took around 13 hours. My test plan for the controller was to assign each button value to a built-in LED on the FPGA. Then I could press each button and make sure the proper LED activates.

## Game Design

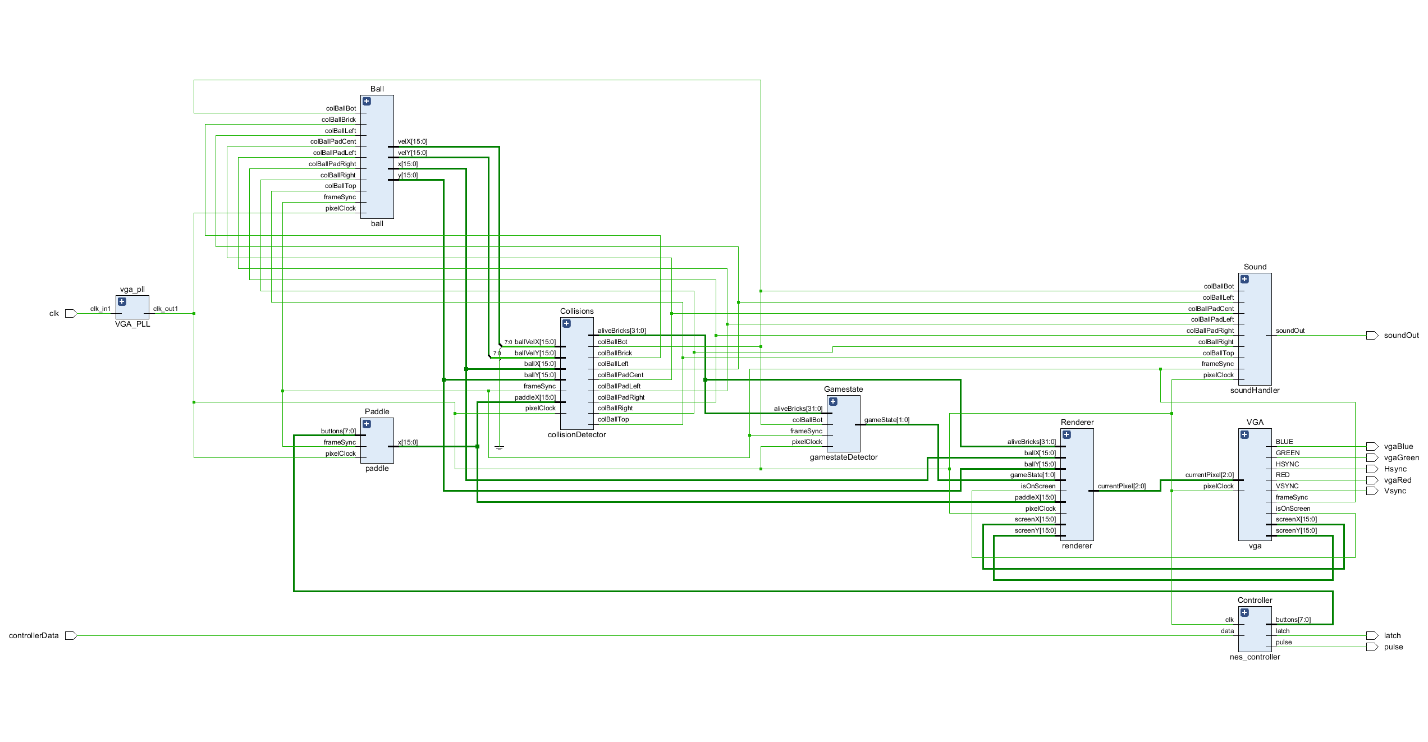
The remainder of the project was to bring each of the hardware components together and design the game. I looked to simple games of the past as I researched a viable game to design for this project. The game I settled on is a variant of Brick Break. The player controls a paddle that moves along the bottom edge of the screen. The player must move the paddle to hit a ball that bounces around the screen. An array of bricks is laid out. When the ball collides with a brick, the brick is destroyed and the ball bounces away. If the player misses the ball, the player loses and game is over. If all the bricks are destroyed, the player wins and the game is over.

As planned, this phase of the project took by far the most time. Combining all the hardware modules while designing the game logic led to a lot of debugging and troubleshooting. Some features needed to be added to the VGA module to fit in with the rest of the project. Drawing shapes on the screen and handling collisions had a lot of room for error and debugging. Designing, developing and debugging the game was done over the course of several weeks and took around 70 hours. The test plan for the project as a whole is to program the FPGA and make sure the game plays as designed.

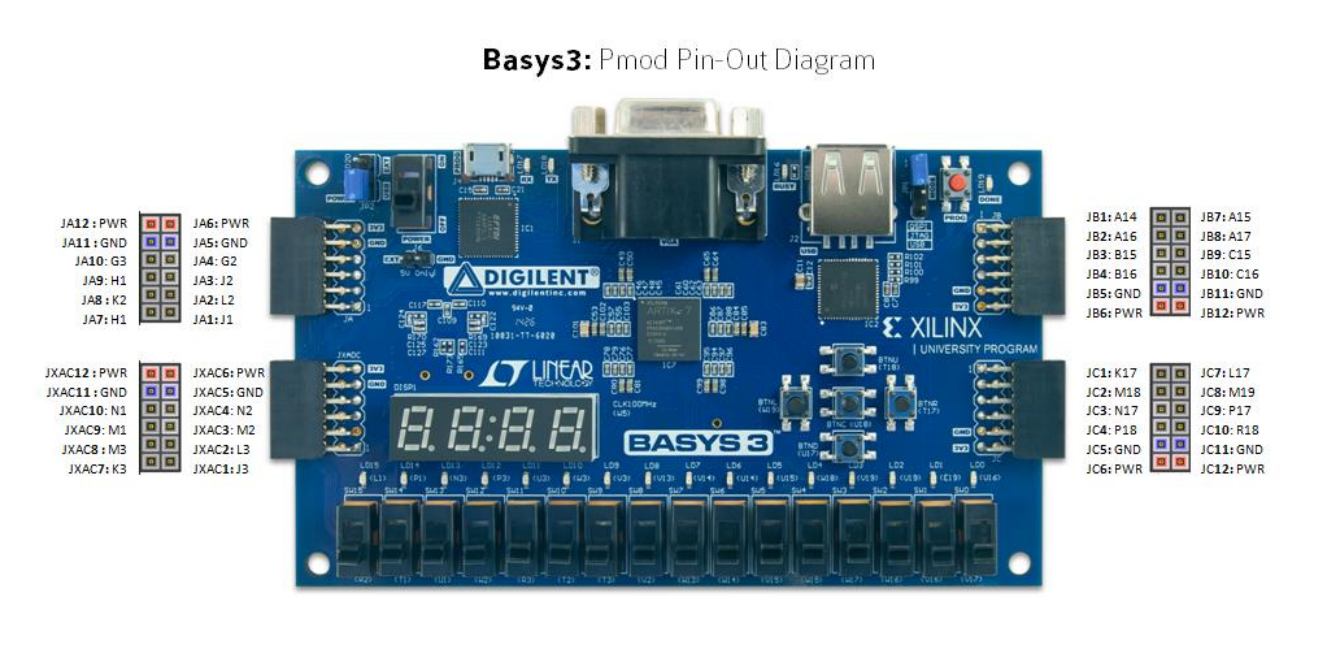
# Results

The following image shows the final register transfer level schematic of this project. For a better view, it is included with this document as Game\_RTL\_schematic.png. This schematic neatly shows how each of the following modules are connected.

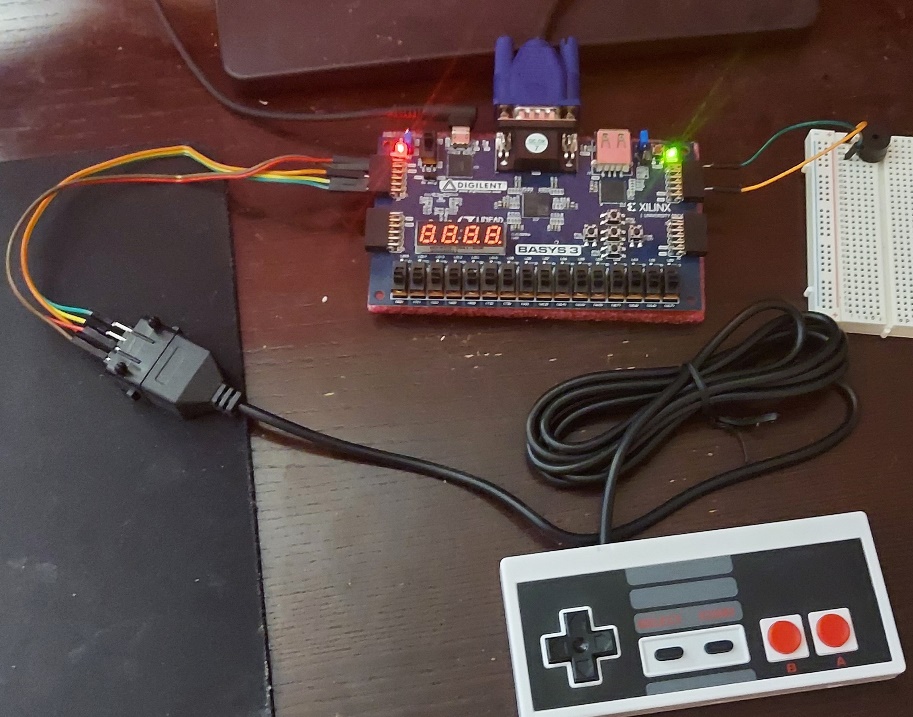
A video demonstration of the game working in full can be seen [here](https://youtu.be/BgsAh8YejHQ).



The following Pin Out Diagram provided with the documentation for the FPGA was helpful when connecting the hardware up all together.



With all the hardware connected, the final product looks like this:

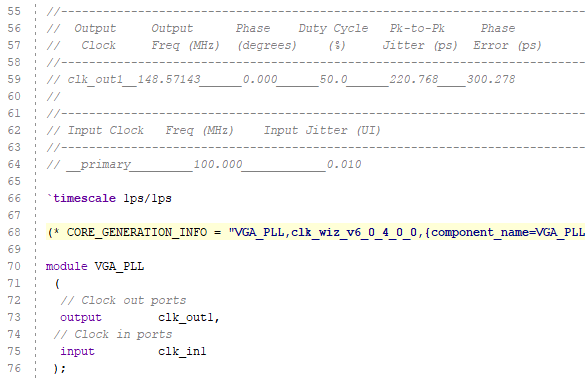


The following sections describe the function of each Verilog module and provide relevant documentation. There will also be some snippets of the Verilog code that are the most crucial to the function of that module.

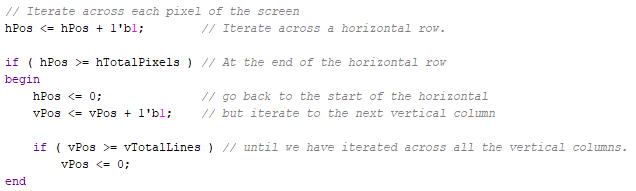
## Video Output

In order to output to VGA, specific timing requirements need to be met. The screen I used has a refresh rate of 60Hz. This means that each pixel needs to be updated 60 times per second. Since each pixel is assigned its color one at a time, a “pixel clock” is used where each pulse updates a pixel. This clock needs to be fast enough to update each pixel 60 times per second in my case. VGA timing value documentation (Green) confirms the pixel clock for 1080p@60Hz to be a clock speed of 148.5 MHz.

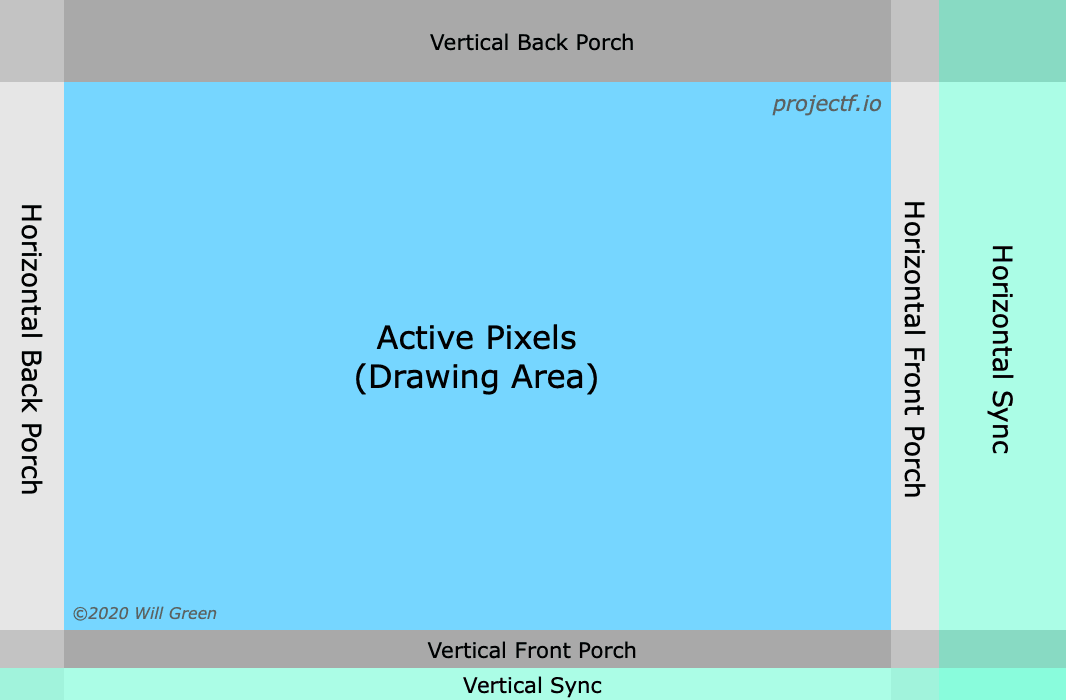
This clock rate is faster than the built-in clock on my FPGA is able to provide. In order to obtain a clock rate of this speed, a phase locked loop must be used. This converts the base clock speed to exactly what is needed for the pixel clock. Vivado has a clocking wizard to automatically create a PLL with the specified input and output clock speeds.



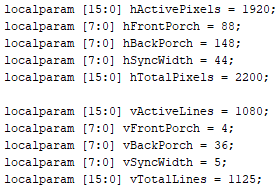
With a proper pixel clock, the VGA module can be implemented. The following code iterates across each pixel of the screen with each pulse of the pixel clock.



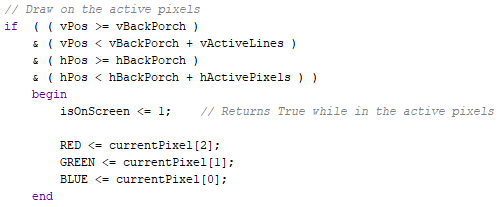
VGA uses several different regions to help sync what is eventually displayed. The following image from the VGA documentation (Green) shows these different regions.



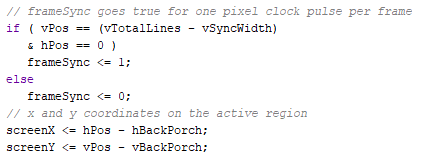
The exact size of each region is different for each resolution. The following parameters used in the VGA module were established in the documentation (Green).



With these parameters established, as we iterate across the display, we can draw our desired color on the current pixel if it is in the Active region.

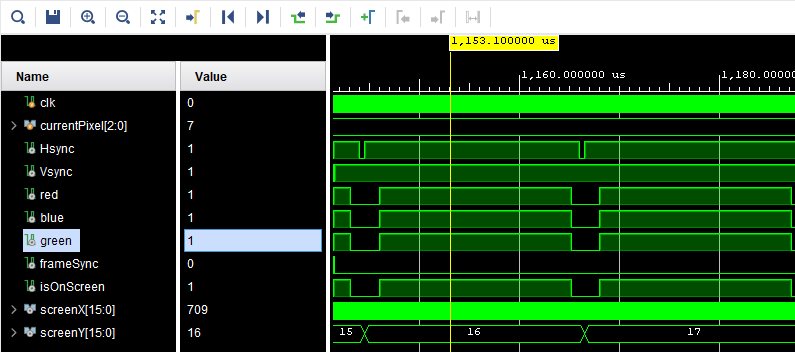


The VGA module also provides important information to other modules of the game. The frameSync signal helps other modules update once per 60Hz frame. The screenX and screenY give useful coordinates to other modules without them needing to worry about the sync regions.



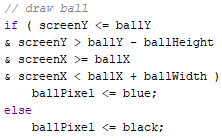
### Video Output – Testbench

Creating a testbench was very helpful to getting the video output to work. The timing diagram made it easy to see the values of each signal and determine where things went wrong in the code.

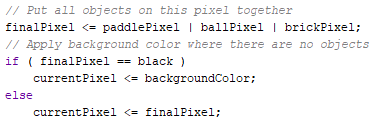


## Renderer

The VGA module must receive as an input the desired color for the current pixel of each pulse of the pixel clock. The Renderer module is given the position of each game object, and uses that information to tell the VGA module what color to make the current pixel. The following code is an example of how a game object is handled by the renderer.

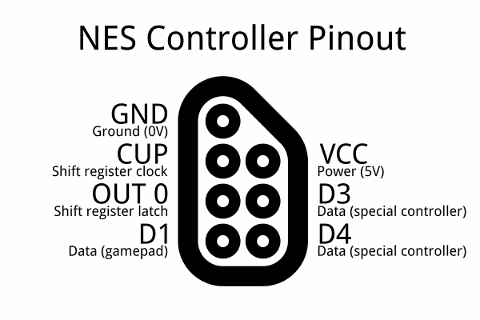


After each game object is considered, the output pixel can be determined. If two objects overlap a given pixel, their colors will be combined. If there is no object at a pixel, the output pixel is assigned the background color.

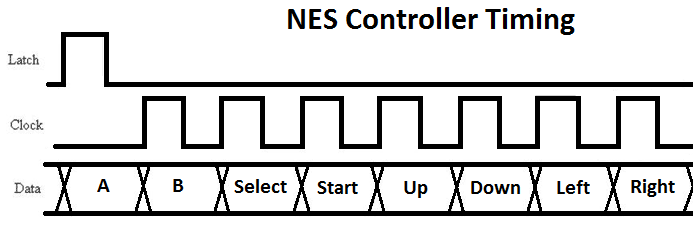


## Controller

The NES controller sends all of its button values through a single data pin, which can be read with the help of a clock signal and latch signal that are sent into the controller. The following diagram was used to hook up each pin to the FPGA (pinoutguide).



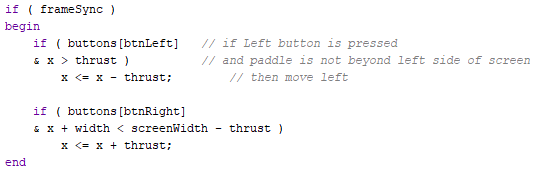
The controller sends button data one data per clock pulse, with a latch signal use to sync to the A button data signal. This process can be seen in the following timing diagram (bertiusgames).



To implement this timing for the controller, we set Latch high which makes the controller send the A button data. Then we set Latch low, and send the controller 7 pulses to its Clock input. This will cause it to return the next 7 button values through its Data output. A common way to implement this is with a state machine by using a case statement for each step of communicating with the controller.

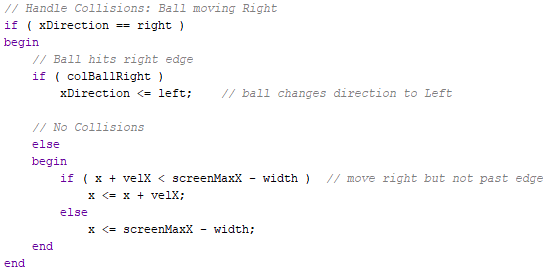
## Paddle

The player uses the left and right buttons to move the paddle along the bottom of the screen. The Paddle module receives the buttons values from the controller as inputs. This module holds, updates and sends the paddle’s x coordinate as an output. The following code shows how it changes the paddles x coordinate each frame according to what button is pressed.

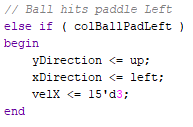


## Ball

The ball moves around the screen bounces off the other objects. This module holds the ball’s position, speed and direction values. It takes collision values as inputs, then handles the collision by changing the ball’s position, direction and/or speed. It gives the ball’s new position and speed values as outputs. The following code shows how the ball handles some potential collisions:

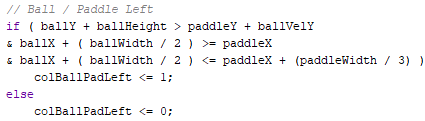


An important feature that improves playability of the game is that the if the ball hits the left side of the paddle, it will move toward the left. Same goes for the right side making the ball move toward the right. This gives the player a lot more control and can partly be seen in the following collision handling event:



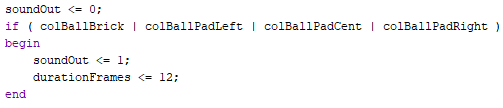
## Collisions

The collision detector module determines if the ball has hit another object. It takes as inputs the position of the various game objects. It compares their positions to see if they are colliding, then sends that information out as outputs. The collision logic is pretty straightforward:



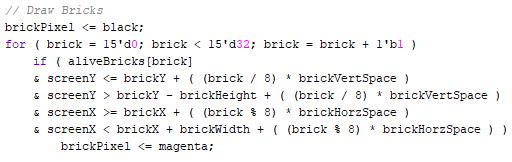
## Sound

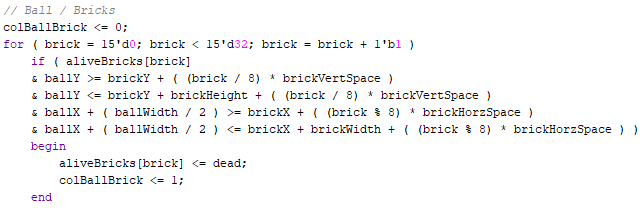
The sound module receives information from the collision handler, which allows it to determine when a signal should be sent to the buzzer to make a beep.



## Bricks

Since the game features an array of thirty-two bricks, the best way to implement the bricks was by using a for loop. In Verilog, for loops work a bit differently than in typical programming language. They are essentially unrolled and each iteration executes in parallel rather than looped through procedurally. They are only really useful as a shortcut to write a segment of code over and over. The bricks are implemented by the following for loops:

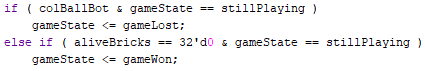




A potential improvement to the project would be to design a module that could handle the bricks, similar to the other game objects.

## Game State

The game state module determines if the game is over, and passes that information to the renderer. If the game is over, the renderer will display a red screen or a green screen depending on whether the player lost or won the game. The game state module takes the ball/screen bottom collision signal as an input, as well as the alive status of the bricks. If the ball hit the bottom of the screen, it sends a “game over – lost” signal. If all the bricks are destroyed, it sends a “game over – won” signal.



# Conclusions

This project fills the purpose for which it was designed. The game is fun to play and works as designed. This project could easily be modified to make other games. Many of the modules would require little to no changes to be used in a different game. One could just design new game objects and game events while keeping the modules that run the hardware. A nice improvement would be to upgrade the renderer to more easily display letters, numbers and shapes beyond just rectangles.

# References

Digilent, (2016). *Basys 3™ FPGA Board Reference Manual.* <https://digilent.com/reference/_media/basys3:basys3_rm.pdf>

Element14, (2019). *Retro FPGA game engine.* <https://community.element14.com/technologies/fpga-group/f/forum/39328/retro-fpga-game-engine>

Green, W., (2020). *Video Timings: VGA, SVGA, 720p, 1080p.* <https://projectf.io/posts/video-timings-vga-720p-1080p/>

Pinout guide, (2017). *Nintendo NES and SNES joystick controller pinout.* <https://pinoutguide.com/Game/snescontroller_pinout.shtml>

Szymansky, R., (2018). *NES / SNES USB Adapter.* <https://bertiusgames.com/projects/gamepad.html>