**FROGGER, THE GAME**

**(*for the Spartan 3E-500 development board)***

ECE 525.442 FPGA Microprocessor Design

**Erik Lee**

**Emily Kahn**

**Edward Jones**

*Abstract*

*FPGA design has been the forefront of modern circuit design as the need for increased reusability is required in the marketplace. Given the need for reusability and an increased demand for lower cost, FPGAs are easily used for scalable and changeable designs to accommodate different needs in any environment. With that in mind, FPGAs are also very useful in creating a multi-function chip where costs can be driven lower to avoid placing multiple components on a circuit card assembly. We explore the robustness of the Spartan 3E FPGA by implementing a game design where multiple functions are used and explored. Frogger, the game we have chosen to implement has several features to its design that display how multiple functions such as VGA output, user input, and edge-detection circuitry can all be assimilated together to form one design on one chip. We provide in this paper a means to explore the creation of this game, our testing methodology, and end with our conclusions about further expansion to development.*

1. INTRODUCTION

With the culmination of modern technology increasingly demanding a need for design reuse, FPGA design has lead the forefront in areas such as military radar applications, automobile drive customizability, vending machine applications, and even video game development. As a boon to developers that continue creating new and innovating applications requiring multiple features, FPGA designers have begun to create multiple code segments in VHDL which emulate common integrated chip functionality. This functionality within a code structure allows for designers to create highly robust and modifiable firmware which can be scaled very easily with ongoing advancement in current technology. FPGA designers can now take segments of code (dubbed as CORE IP) and reuse these structures constantly to suit their design needs. Under this premise, our group decided to use this same approach and combine several different snippets of VHDL to create the game known as “Frogger.”

Frogger, a modern game of 1981, consisted of a frog leaping through dangerous crossroads and rivers in order to advance to the end of the screen. This process may be trivial to the end user, but in the design world many challenges to get all the pieces to flow together arose. From the development of the background, to the obstacles the frog may or may not encounter, to the point system where a computer must track the position of the frog through any terrain it may cross.

What has been discovered by us, is that over the course of our understand of FPGA and VHDL digital design, is that these functions which the Frogger game requires, can all be implemented seamlessly in the Spartan 3E-500 development board, which we will continue to discuss below.

1. BACKGROUND

The premise for our game is to utilize multiple instantiations of CORE and created VHDL files to create the Frogger game. In defining the functionality of the game, a list of requirements was defined from the inception of our design to facilitate operational structure. Each VHDL file created and/or used contained its own specific function for overall implementation of the design. The main flow of operation according to each VHDL file is as follows:

1. Initialize VGA Controller and output parameters
2. Create initial background display of color map
3. Generate moving objects to traverse map for frog to avoid
4. Form frog block and reset position
5. Detect frog motion
6. Update collision detection state machine with current frog position
7. Output score

Each operational item has its own set of VHDL file(s) which combine to form the Frogger game. While most of the design process used key ideas from previous implementations of the game (such as motion, collision of objects, and scoring structure), we added the implementation of a keyboard to further enhance the functionality. As most of the design was created by our group, the keyboard CORE VHDL file helped to facilitate the overall functionality of the game allowing for a greater user experience. We expand on each operation from the above section in detail delineating how each file within the program directory functionally gives purpose to our game.

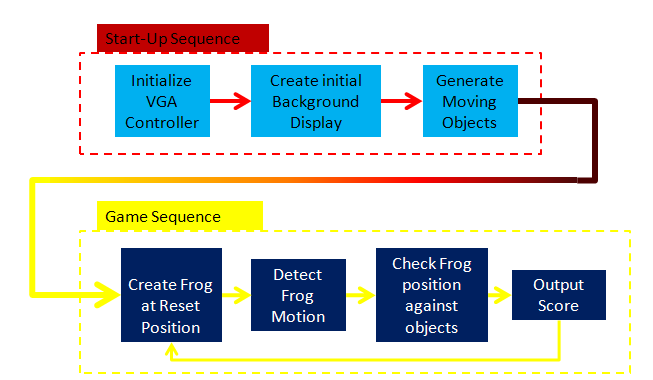


Figure 1 - Game Design/Implementation Flow

1. DESIGN

The design of the Frogger game as mentioned previously is broken down into subsections with each subsection corresponding to a specific function written in VHDL. At the beginning of the game, the FPGA will initialize all game parameters allowing the game to start. The Start-Up sequence involves three VHDL files as described below:

**Start-Up Sequence Files:**

* *vgaSyncGenerator.vhd*
* *backGroundGenerator.vhd*
* *objects.vhd*

*vgaSyncGenerator.vhd*

The vgaSyncGenerator.vhd file implements the onboard VGA controller utilizing both a horizontal/vertical sync bits as well as horizontal/vertical counters to display a 640x480 resolution on a screen. By definition, the VGA standard first scans the first pixel row (horizontal direction) creating the initial horizontal resolution, then proceeds to rescan a second row 1 pixel below the initial scan until the resolution is drawn. The VGA standard uses an 8-bit color vector (three red color signals, three green color signals, and two blue) to create color to be shown on the screen. As the screen is scanning, these colors will get assigned depending on whether or not a signal to the corresponding bit of choice is being sent on the VGA port.

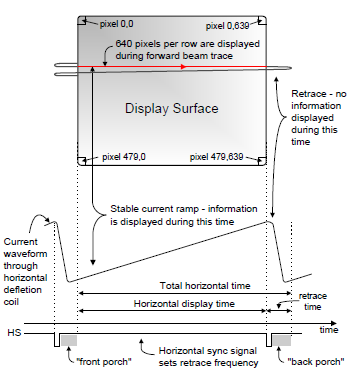


Figure 2 - NEXYS VGA implementation from manual

To do this on the our FPGA, the vgaSyncGenerator.vhd file uses a horizontal counting signal to traverse across the horizontal portion of a display which then gets resets to its initial position incrementing a vertical counter which does the same. The speed at which this is done is created from subdividing the onboard 50MHz clock with the FPGA to 25MHz, as required for standard VGA resolution in order to sync the output to the screen.

*backGroundGenerator.vhd*

Expanding on the vgaSyncGenerator.vhd file, the backGroundGenerator.vhd file creates the color map of the display using the counting signals (vectors) created from the horizontal and vertical scans. This file contains the entity which draws specific colors for the street, water, and grass areas of the Frogger game. Each “pixel” of the generator VGA output gets assigned one of five different colors, depending on pre-defined selection as chosen by the design team for the map layout.

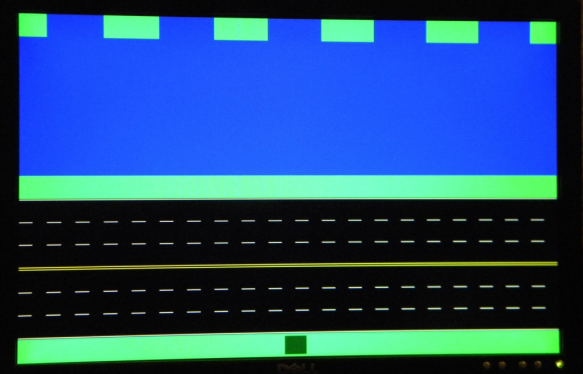
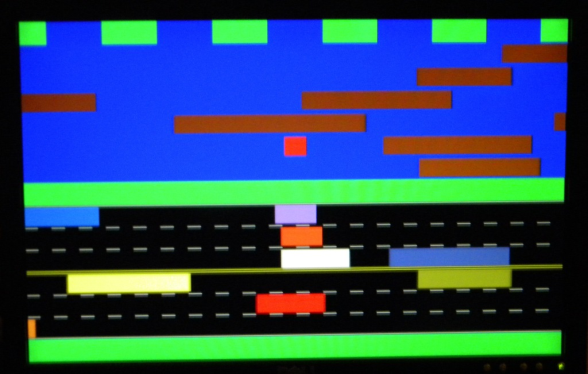


Figure 3 - Initial Background Creation (frog included)

The black and yellow areas of the background map indicate the “street” portion of the display, whereas the blue area represents the “water” portion. Both the green and black areas represent safe areas for the frog to travel, which will be modified when the implementation of the objects.vhd begins.

*Objects.vhd*

The objects.vhd file is what is instantiated to create both the hazard objects (ie: “cars”) that go along the “street” portion of the map, and the “logs” portion of the map which exist in the “water” section.



**Figure 4 - Objects created on screen**

**(red block indicates a dead frog**

This entity within this file houses the speed at which the objects move across the screen, and provides a frog position point. The level design is two-fold. The lower portion of the screen is only hazardous when the moving objects collide with the frog, and the upper portion of the screen is only hazardous when the frog is –not- part of the moving objects. As the frog moves up the screen, the row position is saved to pass to subsequent files in the game sequence to detect whether or not the frog is “safe.”

**Game Sequence Files:**

* *frogGenerator.vhd*
* *frogLocation.vhd*
* *collisionDetection.vhd*
* *score.vhd*

*frogGenerator.vhd*

For the frog to be generated, we implemented a square block that would move according to input from the buttons on the FPGA 3E development board or a keyboard. The entity contains a four-position (Two X coordinates, and two Y coordinates) block which is checked against the scanning horizontal and vertical counting vectors from the vgaSyncGenerator.vhd file. As the user presses command from either device, the frog will traverse the obstacle course in one of four possible directions. As the frog is moving, its current position is checked against the objects.vhd file ensuring that its current position is not part of a predetermined “danger zone” resulting in a death (Water in upper port of the screen, or car collision on lower portion of the screen). As long as the user input does not collide with the objects on the screen, the frog will successfully progress towards the goal. If collision is determined from the collisionDetection.vhd file, it will result in a reset, as well as successfully reaching the goal.

*frogLocation.vhd*

The location and position of the frog is defined in the frogLocationl.vhd file. This file unlike the frogGenerator.vhd file, stores the position of the frog against the generated objects.vhd file. This file tracks the frog movement as well as both the upper portion of the screen and lower portion of the screen to determine with the collisionDetection.vhd file whether or not the frog is safe.

*collisionDetection.vhd*

The major game operation of whether or not the frog survives based on the user input is encountered in the collisionDetection.vhd file. As the frog moves from one end of the screen to the other, his position is set into a Mealy State machine. The state machine determines five different states:

* Frog is on the grass
* Frog is on the road
* Frog is in the river
* Frog is dead
* Frog success (or win state)

The state machine gets information from all of the game files defined thus far and places the frog into a state. Determining whether the state is safe or hazardous is done as information changes from the user input and game files.

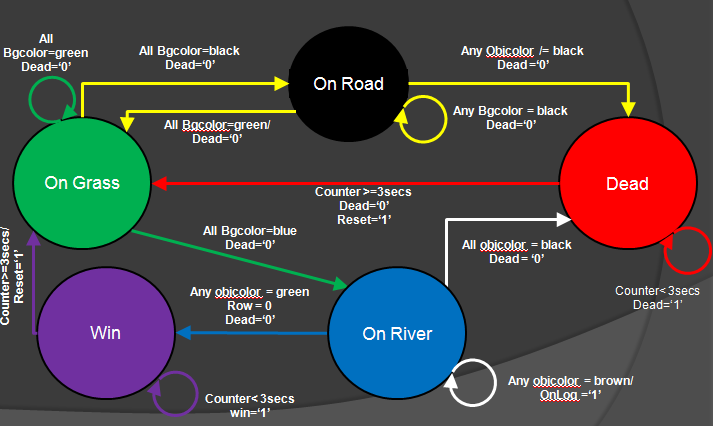


Figure 5 - Mealy State Machine for Frog

<Text further describing the state machine>

*Score.vhd*

For the final portion of the game sequence, a score tracker is implemented using the seven segment display on the Spartan 3E board. As the frog reaches the end of the course without fault, a score is added to a running total and display after the frog resets back to the starting position. Depending on the level of difficulty set by the user input, each successful attempt across the course is awarded 10, 20, and 30 points depending on the increasing level of difficulty.

*Miscellaneous Files*

* Debounce.vhd
* froggerKeyboard.vhd

Lastly for the game sequence, two extra files are used to improve the user experience. The debounce.vhd file is used to ensure that only one movement is detected when pressing the movement buttons on the Spartan 3E development board. Without the debounce.vhd file, the frog may jump too quickly in one direction resulting in a poor game experience.

A second method of user input is the froggerKeyboard.vhd file. This file is a CORE IP file that we implemented to recognize all keyboard key presses to be read by the FPGA. The keyboard implementation is somewhat straight forward. The board clock is fed into the keyboard module. As the clock from the FPGA is ticking, the module is scanning a sequence of hex values during a key press. Since our game only requires four directions, the keyboard module only need recognize four specific hexadecimal values. Since the module itself is CORE, we simply feed the direction value into a component instantiation at the top module for the direction of the frog.

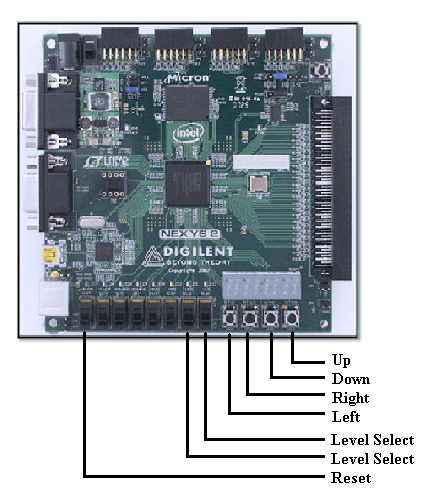


Figure 6 - On board game user input

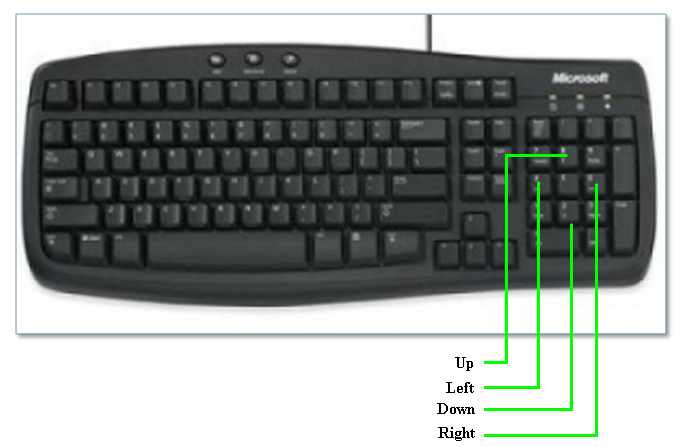


Figure 7 - Keyboard game user input

1. IMPLEMENTATION & VERIFICATION

The top module for our design is described by 15 inputs and 10 outputs (19 total to include the seven segment display and anodes). The keyboard clock, board clock, slider switches, and push buttons all serve as inputs to the design. Outputs include both horizontal and vertical sync, VGA color, and the anodes/encoded character to drive the on-board seven segment display.

Verification of design input is done with a series of test scenarios we created. Rather than using a test bench, a series of operations were conducted after synthesis to verify that the frog and background objects integrated as suspected.

Testing the design we implemented resulted in creating several different modifications to the VHDL code sequence. Initially, the cars and logs shared identical sizes. This was done to ensure that the objects file was generating correctly. Furthermore, the frog collision aspect of the game required its own testing scenario to verify that both the objects, frog, and background created worked in tandem with each other.

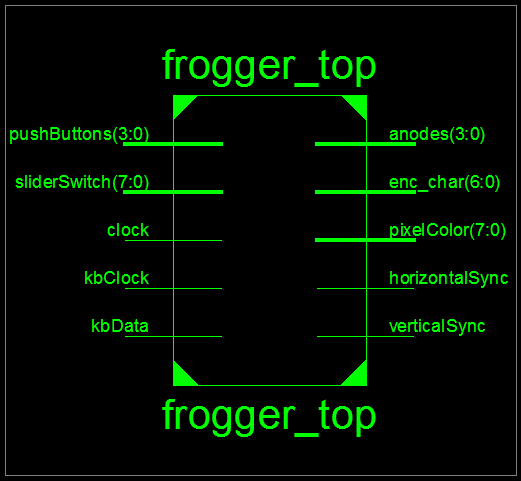


Figure 8 - Top module I/O

1. RESULTS & ANALYSIS

Below are the results from the FPGA Resource and Synthesis reports as created from Xilinx ISE. For our design to be synthesized, close to 1/3 of the allowable FPGA design blocks were used. From the report, 32% of configured “slices” made up the FPGA. From a resource standpoint, the lion’s share of the design stems from comparators and Flip Flops. Most of our coding structure used seems to model this, and our results are expected.



Figure 9 - FPGA Resource Allocation

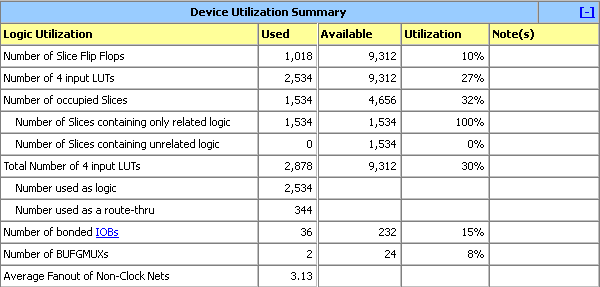


Figure 10 - FPGA Device Utilization

1. CONCLUSION

The development of the Frogger game allowed us to explore many of the robust features that FPGA designs allows for. During development of the game, it became clear that multiple instantiations integrating game functionality provided a greater design experience. The experience is shaped by allowing highly customizable code to custom fit any functions we could think of in order to make the Frogger game work. From component declaration, multi-file design integration, and I/O port configuration, many features of FPGA were taken advantage of.

Many of the design challenges came from event driven outcomes, such as integration of the state machine to fully capture game operation, and user input from both the keyboard as well as the on-board buttons. As we continued to create the coding sequence, a few minor bugs crept up that we were able to iron out (edge detection challenges). It is always part of design where following a pre-determined path gets harder to implement changes as one gets closer towards completion. However, given the nature of how capable the Spartan 3E board was able to create our game functionality with relatively lower resource utilization, it is safe to say that our game provided the full gambit of FPGA design from any vantage point.

1. REFERENCES
2. Xilinx Nexys2 Reference Manual
3. FPGA Resource Guide, <http://www.digilentinc.com/showcase/contests/designcontest.cfm?contestid=8>
4. Keyboard Implementation & Application <http://www.pyroelectro.com/tutorials/ps2_keyboard_interface/theory_ps2.html>