**FROGGER, THE GAME**

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

ECE 525.442 FPGA Microprocessor Design

**Erik Lee**

**Emily Kan**

**Edward Jones**

*Abstract*

*Field-programmable gate array (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, lowering costs and avoiding placing of 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 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 understanding 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 create 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 maps
3. Generate moving objects to traverse the map for the frog to avoid
4. Form frog block and reset position
5. Detect frog motion
6. Update collision detection state machine with current frog position and status
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.

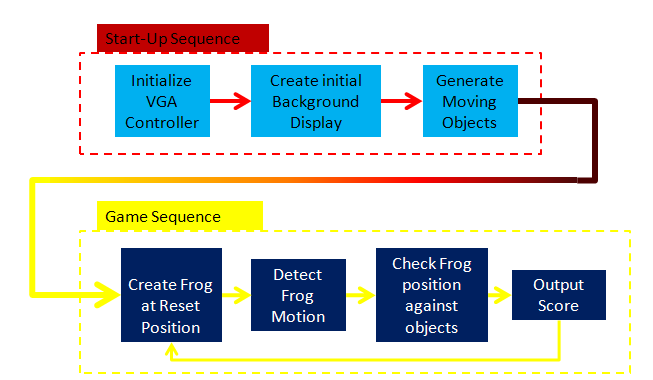


Figure 1 - Game Design/Implementation Flow

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*
* *randomGenerator.vhd*

*vgaSyncGenerator.vhd*

The vgaSyncGenerator.vhd file contains the vgaSyncGenerator entity which 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 asserted depending on whether or not a signal to the corresponding bit of choice is being sent on the VGA port.

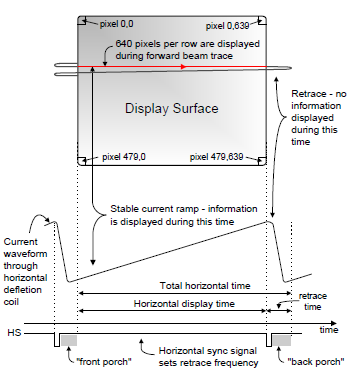


Figure 2 - NEXYS VGA implementation from manual

To do this on the FPGA, the vgaSyncGenerator entity 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.

The horizontal counter is used to identify a particular pixel in a column of the VGA monitor while the vertical counter is used to identify a particular row of pixels. Using both counters as xy-coordinates of a single pixel as inputs to the following entities enables us to assign a particular background color, object color, or the frog color to develop the visible grass, roads, rivers, cars, logs, and a frog which make up the game.

*backGroundGenerator.vhd*

Expanding upon the function of the vgaSyncGenerator entity, 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 8-bit colors, depending on pre-defined selection as chosen by the design team for the map layout.

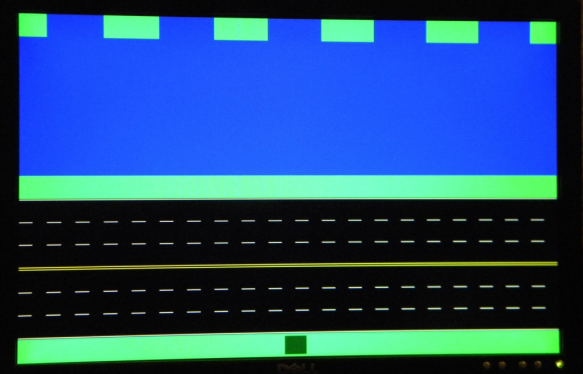
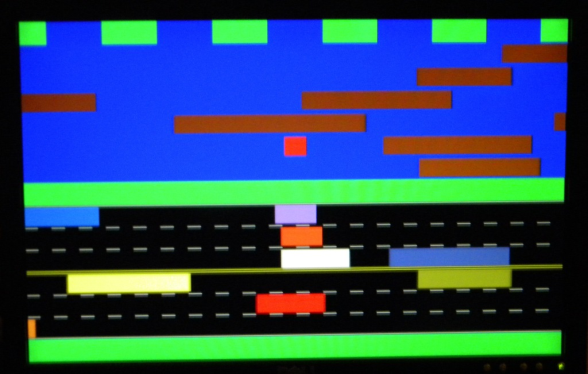


Figure 3 - Initial Background Creation (frog included)

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 and objectGenerator.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 “river” section.



**Figure 4 - Objects created on screen**

**(red block indicates a dead frog**

The entity within this file houses the object location, the speed at which the objects move across the screen, the direction the object moves, its size and its color. If the horizontal and vertical counters of the VGA driver do match the position of the object, the 8-bit object vector color is assigned all a non-zero color vector else the 8-bit color vector is assigned all zeros. Notice that brown is exclusively assigned to logs. All car objects are also non-zero. The road portion of the screen is only hazardous when the moving objects collide with the frog, and the river portion of the screen is only hazardous when the frog is not intersected at least partially by one of the moving logs. The file also takes in the current row position of the frog to prevent the user from changing the speed of the objects and consequently the point value unless the frog is at the start position.

*randomGenerator.vhd*

The randomGenerator.vhd file is used to generate random 12 bit numbers. Twelve bits was chosen because it supplied enough random bits to generate the necessary values. It is a simple linear feedback shift register (LFSR) with taps at bits 12, 6, 4, and 1. The using these taps ensures all 4096 numbers are used before the numbers repeat. Also tapped in the LFSR is a truly random input from the user. Every time the user presses the forward button the enable signal gets XOR’d in the shift register.

The random stream of bits output by the randomGenerator module generates influences a few features of the objectGenerator. Every time an object goes off screen it reaches a point where the features get randomly changed. For the logs and the cars the size changes within certain values. For the cars only the color also gets randomly generated. Also one other item gets changed. All the objects are based off the left edge. When the objects are off the screen they also get a randomly generated jump value. This prevents the objects left edges from repeating the same pattern through every rotation across the screen.

**Game Sequence Files:**

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

*frogGenerator.vhd and frogLocation.vhd*

For the frog to be generated, we implemented a dark green square block that would move according to input from the buttons on the FPGA 3E development board or the keyboard. The frogGenerator component asserts the frog’s color (dark green) for a pixel region 25 pixels wide, and 25 pixels long. If the frog is alive, the frog is green. If the frog is hit by a car, or if it falls completely into the water, the frog’s color alternates between red and white in the 25x25 pixel region where the collision occurs. The frogLocation entity controls the frog movement across the screen. It outputs the current xy-position of the frog’s four corners (two X coordinates, and two Y coordinates) block which is checked against the scanning horizontal and vertical counters from the vgaSyncGenerator component. The four corners are the boundaries of the frog. When the horizontal and vertical pixel counters of the vgaSyncGenerator are within the bounds of these four corner pixels, the green color of the frog is asserted and displayed. As the user presses a command from either device, the frog will traverse the obstacle course in one of four possible directions, updating the position of the four corners and the result is that a different grid of 25x25 pixels will be displayed green. The xy-position of the four corners of the frog are also used as inputs to the collisionDetection component to determine whether its current state is safe or not.

*collisionDetection.vhd*

After developing the layout of the game background, the objects, and the frog, we had to implement the rules of the game, by dictating how the frog would interact with the objects if both should intersect or not intersect. The rule portion of the game is modeled by a Mealy state machine contained within the collisionDetection entity. Since we use the Mealy model for our finite state machine, our outputs depend on the current state and the inputs. Our model consists of five different states:

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

The inputs to the machine are the object colors, the background colors at the four corners of the frog, and a timer counter. The outputs are: 1) a dead flag to indicate that the frog has either died by a collision with a car or by falling completely into the river; 2) an on-a-log flag to indicate that the frog has successfully jumped onto a log; 3) a reset flag to indicate that the frog must be returned to the beginning; 4) a win flag to indicate that the frog has reached the end of the game safely. A diagram of the Mealy state machine model used for the game is shown in the figure below.

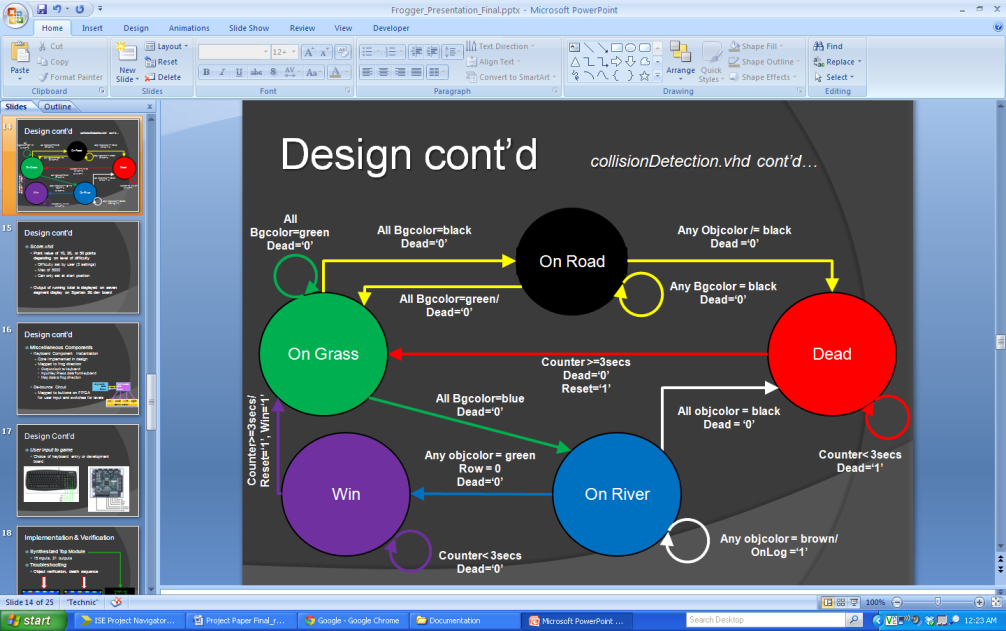


Figure 5 - Mealy State Machine for Frog

Each circle in the diagram represents a different state in which the frog can be. The arrows represent the transition from one state to another. Each transition arrow is described by the current state (the origin of the arrow), the input, and the corresponding output (indicated after the forward slash).

As described earlier, there is an object 8-bit color vector, a background 8-bit color vector, and a frog 8-bit color vector. These color vectors are assigned to particular pixels, yet only one of the three is asserted to the VGA monitor. The object color vector and the background color vector assigned to the four corners of the frog are input into the state machine to determine which state the frog to remain or transition into.

If all the four corner pixels are assigned a background color vector that is equivalent to the grass color (green), then the frog is in the “On Grass” and the frog is considered safe (i.e. dead flag is assigned a logic ‘0’). While in the “On Grass” state, if all four corner pixels are assigned a background color vector equal to black (vector is all zeros), then the frog transitions into the “On Road” state; similarly, if all corners have a background color equal to blue, the frog transitions into the “On River” state.

The frog can transition into the “Dead” state if it is currently in the “On Road” state and the object color vector assigned to any corners of the frog are non-black, i.e. an object is located at the same location as the frog, indicating a collision with a car – and that the frog has become roadkill. The other possible condition under which the frog can transition to the “Dead” state is when it is in the “On River” state and the object color vectors for all four corner pixels are black, i.e. no object is located at the same location as the frog, which indicates it has fallen completely into the river – and has drowned. The output of the dead flag is given a logic value of ‘1’ while it is in the “Dead” state, and only transitions back to the “On Grass” state when the maximum timer value of 3 seconds is reached by the counter.

Using similar logic described for the game thus far, the frog reaches the “Win” state only from the “On River” state and if it has reached the grass region at the top of the VGA display. Like the timed “Dead” state, the frog remains in the “Win” state as long as the timer counter is less than the maximum timer value of three seconds.

The four outputs of the state machine affect the appearance of the frog and its position in the game. This is ultimately what defines the visible actions of the frog in the game. The logic value of the “Dead” flag is determined by the current state of the frog (On Road or On River) and the input (object colors, background colors, and timer counter value). When the “Dead” flag has a logic value of ‘1’, the frogGenerator entity asserts red and white for the frog color to indicate that it has died. The logic value of the “reset” flag depends on the frog being in the “Dead” or the “Win” state and if the timer has reached 3 seconds. When the reset flag is set to a logic value of ‘1’ then the frog is reset to the start position. Until the timer reaches 3 seconds, the frog should remain at its current position either flashing dead or remaining green. The “on-a-log” flag operates similarly to the “dead” logic signal, in that it depends on the frog being in the “On River” state and that the frog has successfully landed on part of a log. This signal is used in the frogLocation entity to move the frog at the same speed of the log it has jumped onto. The “Win” flag depends on the frog being in the “Win” state and the timer reaching the 3 seconds. This flag, when equal to logic ‘1’, is used in the score entity to update the total score of the user.

*Score.vhd*

For the final portion of the game sequence, a score keeper 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 displayed 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, or 50 points depending on the increasing level of difficulty. The score keeping uses the VHDL code from a third-party for the binary coded decimal function used to convert the hexadecimal score to a decimal representation. The total score is incremented each time the “win” flag gets a logic value of ‘1’. The maximum possible score over the course of playing the game is 5000 points, at which point the score no longer increments and must be reset by toggling sliderswitch SW7.

*Miscellaneous Files*

* Debounce.vhd
* froggerKeyboard.vhd

Lastly for the game sequence, two 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 an inaccurate response between a key press and frog jump and ultimately 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 needs to 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.

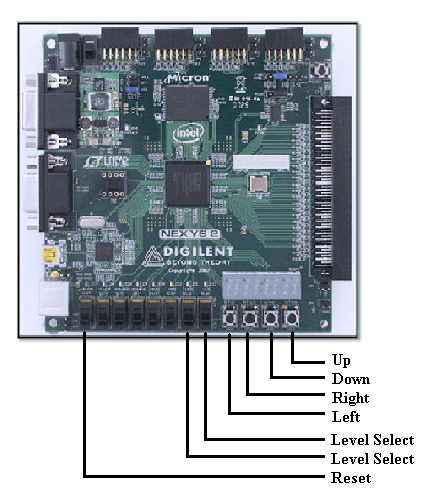


Figure 6 - On board game user input

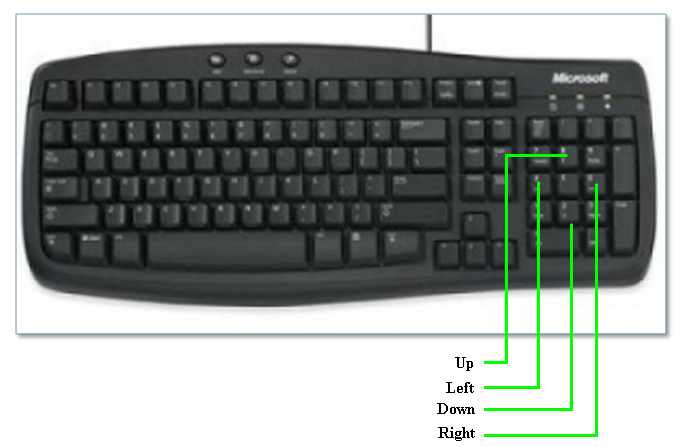


Figure 7 - Keyboard game user input

1. IMPLEMENTATION & VERIFICATION

The top module for our design is described by 15 inputs and 21 outputs. The keyboard clock, board clock, slider switches, and push buttons all serve as inputs to the design. Outputs include both horizontal and vertical sync signals, an 8-bit VGA color signal, and the anodes and encoded characters 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 expected.

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 component was generating correctly. Furthermore, the frog collision aspect of the game required its own testing scenario to verify that frog behaved correctly when interacting with the objects and background created.

Development of the game did not come without its challenges. The first was restricting the frog movement to the boundaries of the 640x480 pixel region displayed on the monitor. During the early stages of the game design, the frog was allowed to move beyond the edges of the screen and wrap around to the opposite edge of the screen. This would result in the user surviving on a log on a loop and never fall off into the water or even jump directly from the start to the finish. To resolve this problem, the frog’s horizontal and vertical location counters could not reset or wrap to the beginning (appearing at the opposite end of the screen).

Troubleshooting was required for the first design of the rules implementation which was first designed as having only two states – dead and undead. It operated on a similar principle as the state machine described for the collisionDetection entity. The frog was able to move across the road and grass portions of the game correctly, dying and flashing when colliding with a car, and remaining green when successfully avoiding them. However, the frog would immediately flag the “dead” flag and return to the start whenever it jumped onto the first row of logs. The cause is believed to be that the state machine was evaluating the inputs to the state machine a cycle to early. To resolve this, we redesigned the rules model, expanding the “undead” state into the “On Grass”, “On Road” and “On River” states. The object and background color vectors (inputs to model) should be evaluated in one cycle to determine where the frog is in the game. The inputs should be evaluated on the next clock cycle to then determine if the frog should transition into the “dead” state or remain safe in its current state. The initial design did not distinguish between the frog arriving at a particular region of the game (road vs. river) and the frog colliding with an object.

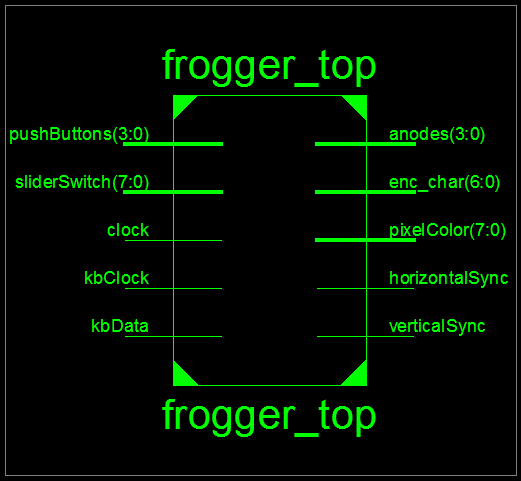


Figure 8 - Top module I/O

1. RESULTS & ANALYSIS

Below are the results from the FPGA Resource and Synthesis reports as created from Xilinx ISE. For our synthesized design, close to forty percent of the allowable FPGA design blocks were used. From the report, 39% of configured “slices” made up the FPGA. From a resource standpoint, the lion’s share of the design stems from comparators and Flip Flops (DFFs). Most of our coding structure used seems to model this, and our results are expected. Since we used a very modular design for the game, we were able to do a more granular analysis of where the most resources of the FPGA were being dedicated based upon the entities instantiated.

The following is a breakdown of FPGA resources used for each entity instantiated as well as for the overall game.

* **VGA drivers**
  + 2 Counters ( pixel count)
  + 35 DFFs
  + 2 Adders / Subtractors
  + 8 Comparators
* **Frogger Keyboard driver**
  + 1 Counter
  + 20 DFFs
  + 1 XOR
* **Background Generator**
  + 8 DFFs
* **Object Generator**
  + 60 DFFs
  + 7 Adder / Subtractor
  + 9 Comparators
* **Random Generator**
  + 1 Counter
  + 1 DFF
  + 1 Comparator
  + 1 Xor
* **Frog Generator**
  + 1 Counter
  + 8 DFFs
  + 5 Comparators
* **Frog Location**
  + 1 ROM (frog Row location)
  + 42 DFFs
  + 3 Adders / Subtractors
  + 3 Comparators
* **Finite State Machine**
  + 5 states, 61 transitions, 22 inputs, 3 outputs
  + 1 Counter
  + 64 DFFs
  + 8 Comparators
* **Score Keeper**
  + 1 ROM
  + 38 DFFs
  + 1 Adder/Subtractor
  + 1 Comparator

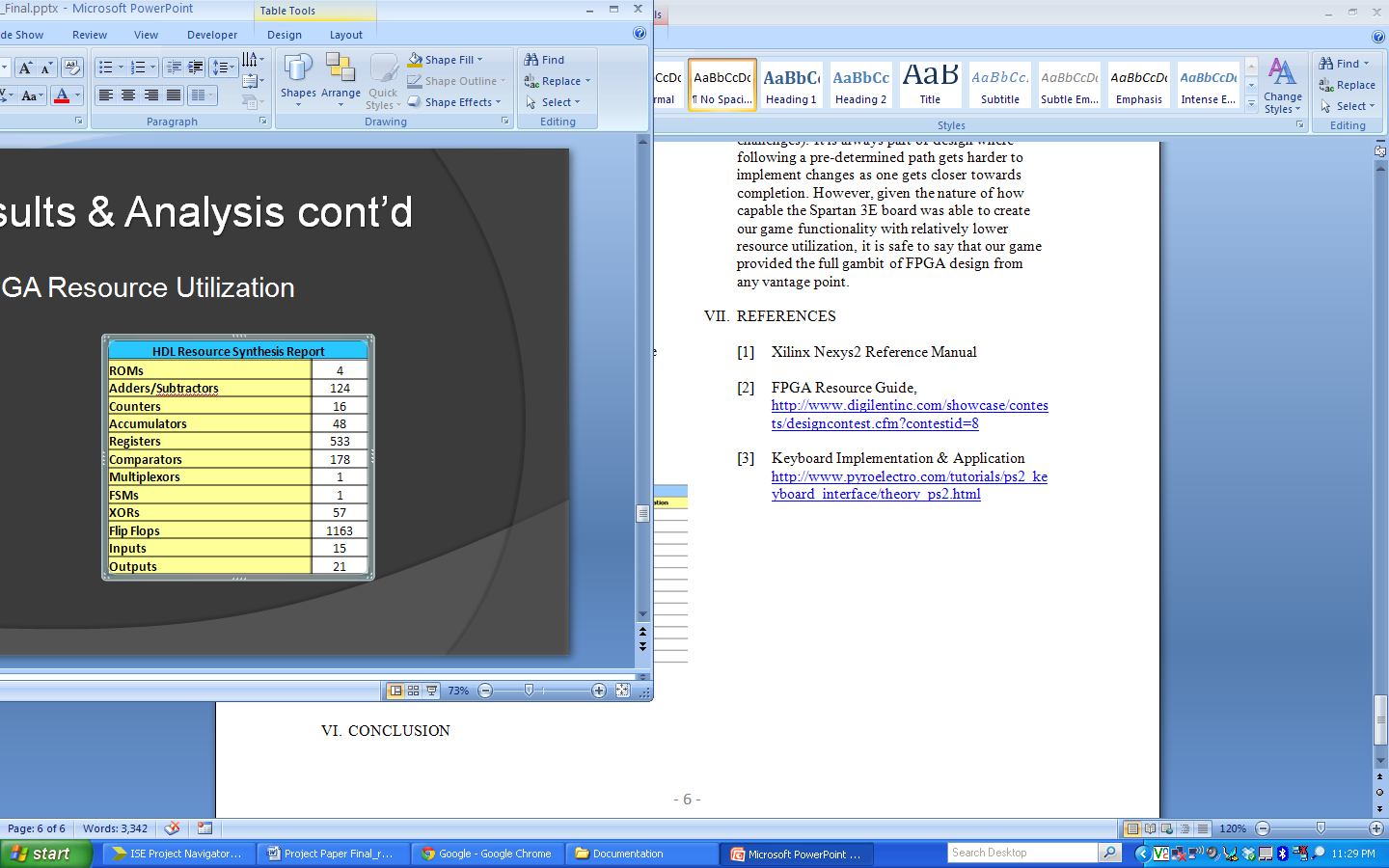


Figure 9 - FPGA Resource Allocation

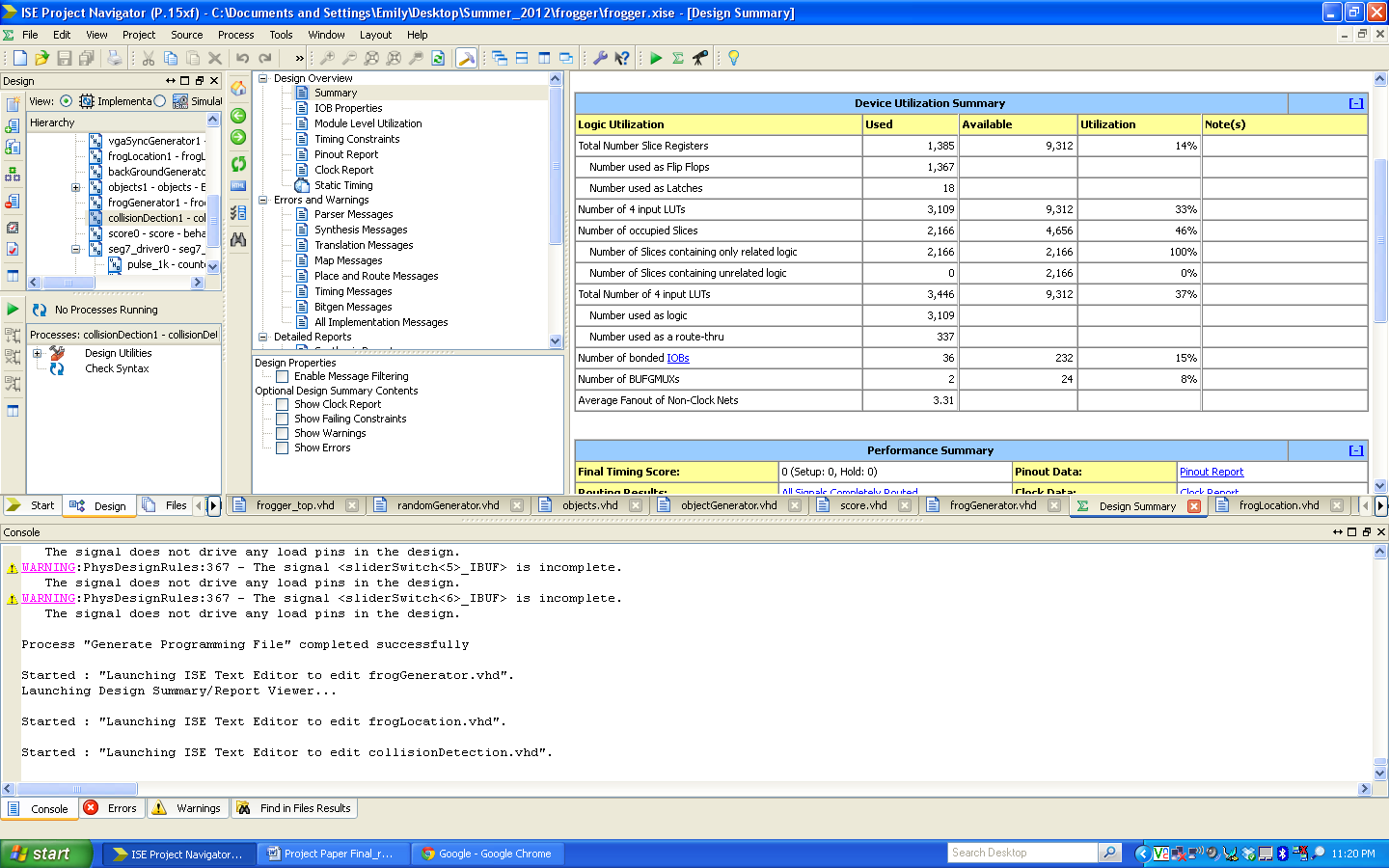


Figure 10 - FPGA Device Utilization

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 more efficient approach to designing a game. The experience is shaped by allowing highly customizable code to custom fit any behavior we desired to have in the Frogger game. 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. Future improvements to the game could include reducing the device utilization for certain synthesized components or using programmed .bit images from SRAM to use as the static background of the game as others have done. This could improve the images of the frogs, cars, and logs by adding more details. Levels of complexity could also be added to the game such as additional frog lives or sound effects such as the frog jumping or splashing in the river. 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>