



Seattle Snakers



CSE/EE 371 Final Project

Spring Quarter 2018.



Lab 5 Report

CSE/EE 371

Spring Quarter 2018.

June 6th, 2018.

Team Members:

Nguyen Lai

Uday Mahajan

Haytham Shaban

Samson Waddell

I. ABSTRACT

Lab 5 incorporates concepts thoroughly established in our digital circuits class to develop a system that utilizes concepts such as combinational logic, sequential logic, and overall state machines for operation management and data evaluation. In this lab we will be designing and building the hardware communication system to play multiplayer Snake Game controlled by an Android application via Adafruit Bluetooth shield and Intel Altera FPGA for the *Future Track* of the UW CSE/EE 371 course.

II. TABLE OF CONTENTS

ABSTRACT	3
TABLE OF CONTENTS	3
INTRODUCTION	4
PROJECT CONCEPT	4
BACKGROUND INFORMATION	5
RISK ANALYSIS	6
SYSTEM TESTING	8
IMPLEMENTATION	10
RESULTS	21
FUTURE WORK	24
CONCLUSION	25
APPENDIX	25
INDIVIDUAL CONTRIBUTIONS	26

III. INTRODUCTION

In this lab we will be designing and building the hardware communication system to play modified version of multiplayer classing Snake Game controlled by an Android application via Adafruit Bluetooth shield and Intel Altera FPGA.

We will continue to work with the Quartus II and implement a simple microprocessor system that utilizes the Altera FPGA that will incorporate an interface to support a serial-parallel-serial network. We will write a verilog program consisting of various modules to make a functional Snake game projected on a monitor display via FPGA. The game will be controlled via an Android Phone app via both the players. The Android phone communicates with the FPGA wirelessly via Bluetooth using an additional component Adafruit Bluefruit 52 wired to FPGA. The single Android app will allow two the players to control their game, third player will play using FPGA buttons and/or keys.

IV. PROJECT CONCEPT

The group's final project for "Digital Circuits and Design" incorporates aspects of digital circuitry, hardware description, and wireless communications to create an immersive multiplayer game utilizing multiple devices. "Snake" is a classic arcade game where the player maneuvers a "snake" in four directions throughout the screen in order to collect as many apples as possible while avoiding possible collision with its own body. The team's general goal was to create a version of this game that allows multiple people to play the game together, using wireless controllers separate from the DE1_SoC, with the game being displayed on a VGA monitor. Each player would control a different snake and try to collect the apples while eliminating the other players until only one is left. The parallelism of the FPGA is harnessed in the multiple snakes being controlled and updated at once, with separate modules instantiated and being updated concurrently for each snake. Snakes move and change location in parallel without having to wait for something else to move.

We succeeded in this goal, ending up with a version of "Snake" involving 3 players. In this case because we only had one bluetooth module, two snakes shared a stream for serial communication (not quite fully parallel), but hooking up multiple bluetooth modules so that 3 or 4 or more different players are using different phones to control different snakes at the same time would be a simple extension of our project expanding its use of FPGA parallelism (not done here because of budget constraints). The Background Information section includes the basic rules of "Snake" that our version follows, while the following list includes some of the more specific characteristics of our finished project:

- Two players can control the movement of their respective snakes using an Android application (the application handles two sets of input controls, so the two players share one phone as a controller). The third player uses the buttons on the FPGA.
- Each snake has the ability to activate "Ghost Mode" (via button in the app, or a switch for third player):
 - Ghost Mode allows the snake to collide with and go through other snakes without dying. However, snakes in "Ghost Mode" can still die if they hit themselves.
 - Ghost Mode has an active time of 3 seconds.

- Ghost Mode has a cooldown time of 10 seconds..
 - Players can only use Ghost Mode 3 times during a single instance of the game.
- Resetting the game is controlled with a switch. Upon reset, a start/title screen is displayed until the start switch is flipped. When the game ends, a game over screen is displayed.

V. BACKGROUND INFORMATION

“Snake” is a pretty basic arcade game. In this case, there are three players controlling snakes, each trying to avoid the other snakes while eating the randomly spawned apples that make the snake longer. More specific rules include:

- Each player starts in a corner of the “arena” (game area displayed on the monitor), with a length of 4 blocks.
- At the start of the game, an apple is at the center of the arena, and after being consumed a new one is spawned at a random location somewhere else.
- Eating an apple increases a snake’s length by 1 block, and adds 1 to their score
- Each snake can move in one of four directions at a time (left, right, up, or down), relative to the arena (i.e., to go in a circle, a snake would have to move left, then down, up, and right)
- If a snake hits any part of their own body, or the body of another snake, they die.
- If snakes collide head on both snakes die.
- Dead snakes disappear from the arena.
- The game ends when there is only one snake left alive.

These basic constraints define the general way the game works, and were the rules we referred to in developing the logic for the initial version of the game.

The Adafruit Feather nRF52 Bluefruit is a all-in-one Bluetooth Low Energy Board with the Nordic nRF52832 native Bluetooth chip. The Feather is basically an Arduino with built-in Bluetooth capability. However, the module is only used for communication purposes. Adafruit provided Arduino sketches for the basic functionalities of the module:

- UART Chat: basic chat application between the phone and the Feather. Both the phone and the Feather can send and receive strings.
- Controller: a basic controller that has a directional pad and 4 other buttons on the phone. Each button pressed send a number as a byte to the Feather.
- Color picker: A sample utility provided by the manufacturer to learn and get basic understanding of the application and UART communication with the Adafruit Feather device.

There are many other example sketches provided by Adafruit, however, due to the nature of our project, the team mainly focused on the “Controller” sketch. In the “Controller” sketch, when a button is pressed, the phone sends 4 bytes to the Feather in this order: [!][B][5][1]:

- The first byte is set to be “!” by default.
- The second byte represents the mode that the phone application is in:

- “A” represents UART Chat
- “B” represents the Controller
- “C” represents the Color Picker
- The third byte represents which button is pressed by its number representation.
- The last byte represents whether or not the button was pressed or released:
 - 0 is released
 - 1 is pressed

The Arduino sketch then interprets the bytes sent by the Phone Application then print out to the Serial Monitor the corresponding values. In our version of the sketch, instead of printing to the Serial Monitor, we modified it so that it send out the number representation via the Serial Output (TX Port):

- Player 1:
 - UP: 4
 - DOWN: 1
 - RIGHT: 2
 - LEFT: 3
- Player 2:
 - UP: 5
 - DOWN: 8
 - RIGHT: 7
 - LEFT: 6

VI. RISK ANALYSIS

Risks/Challenges we were able to Overcome:

- Communication Protocol between Android Application and the Adafruit Feather nRF52:
 - Initially, the team had no prior experience regarding Bluetooth Communication, Android development and learned about the same using the open source code available on the Adafruit’s website.
 - The open-source code had some bugs that did not allow sending inputs to the Bluetooth module. These were fixed by modifying the protocol when the bluetooth module was receive commands.
- Communication Protocol between Adafruit Feather nRF52 and the FPGA:
 - At first, the bytes being transmitted by the feather bluetooth were shown to be completely incorrect.
 - The team later discovered that this was due to the *Serial.println()* command of the Feather. With only one UART on the device, the “write” and “print” commands perform very similar actions in the sense that the serial console and transmission line share a common bus. This causes a mix of data being transmitted that produces incorrect output for both the console printing, and the bluetooth module transmitting to the FPGA.

- After removing the unnecessary *Serial.print()* and *Serial.println()* commands, the signals sent by the Feather are sent correctly by byte along UART: correct binary representation of the button is sent.
- Converting Pixels for displaying Start and Game Over screens:
 - It was difficult and tedious to convert the pixel arts to an array of 0's and 1's for displaying SNAKE and GAME OVER on the VGA display via Verilog. So, a Python program was written to solve the problem. The implementation on Verilog also required that the images be 90 degrees clockwise from its original orientation due to weird behavior from the VGA display. "Figure 1" shows the images input to the Python program, and "Figure 2" shows the array assignments the program created.

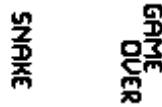


Figure 1: Images sent to python script

```
startScreen[17][TITLE_WIDTH-1:0] = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b0, 1'b0, 1'b1, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0};
startScreen[18][TITLE_WIDTH-1:0] = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b1, 1'b1, 1'b1, 1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0};
startScreen[19][TITLE_WIDTH-1:0] = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0};
startScreen[20][TITLE_WIDTH-1:0] = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0};
startScreen[21][TITLE_WIDTH-1:0] = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0,
1'b0, 1'b0, 1'b0};
```

Figure 2: Instance of generated output used to assign pixels (slightly modified for formatting and naming)

- Logic Handling Output of UART Receiver:
 - Initially, the “result” buffer was output all the time. Therefore, inaccurate results maybe output during the sampling of consecutive signals. As a result, incorrect signals were sent to the FPGA.
 - This issue was fixed by only changing the output when the sampling is finished.

Challenges we were not able to Overcome:

While we did end up with a complete, working version of the game, there were some extra features we had hoped to include but did not successfully figure out in time:

- Allowing snakes to change speeds: We attempted to add an ability where a snake could move at twice the speed for a short amount of time. We tried to do this through varying clock division controlled by user input, and while we could make a snake go faster, the methods we had time to implement did not work with the logic used to display a full snake body (snakeBody module), so the snake broke apart or disappeared. This failure was never resolved so we instead focused on the ghost ability.
- Implementing a “Tron” Mode: We did create basic logic to add a feature where the game could be switched to Tron, where the snake always leaves a trail that never disappears, instead of getting longer with apples. However, it could be switched on and off at any point in a game, which would break the game. We were unable to successfully implement the mode so that Tron could only be switched on or off before the game started, without disturbing logic determining how the snake is displayed (snakeBody module), so we scrapped the feature.

VII. SYSTEM TESTING

Testing our system at large mainly involved ensuring that expected output was shown at each stage when interfacing between peripherals. The first step was ensuring that commands from the phone application were being interpreted correctly by the bluetooth module. The phone application was also tested on various screens in portrait mode. The UART class was also tested using trial and error to ensure that bluetooth receiver was receiving the text properly as the source code provided by the manufacturers had some error initially. This would be as simple as printing out the stored value within the implementation of the receiving protocol for the module. This output is shown in serial console, as in “Figure 3” below.

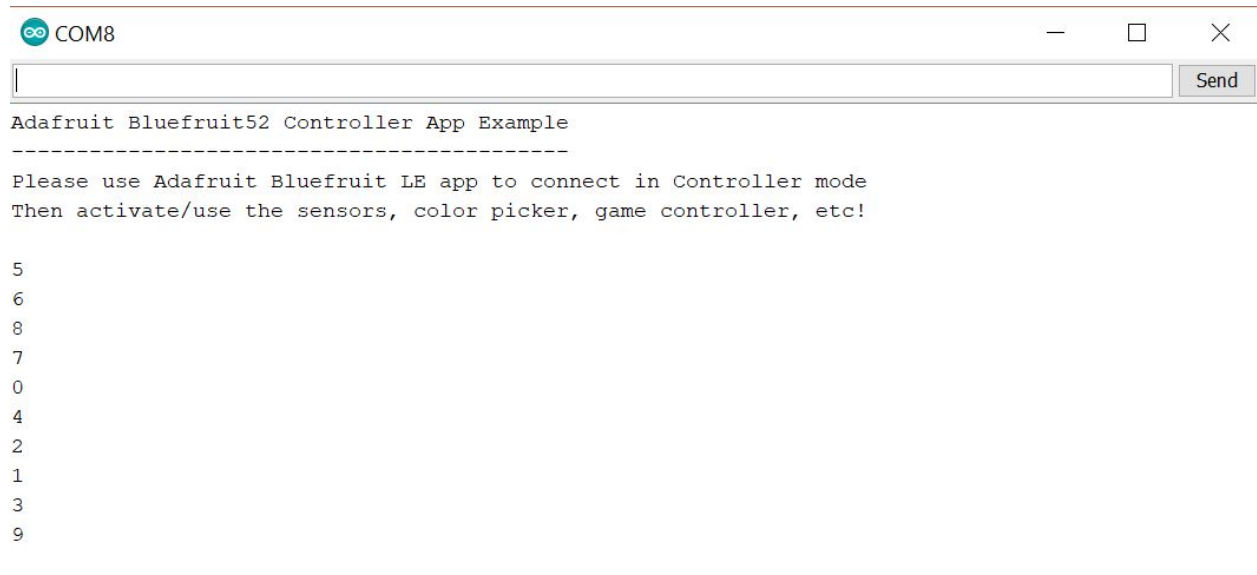


Figure 3: Serial Console output of Bluetooth Module Interpreting Smartphone Application Commands

The next step was ensuring proper transmission along the wire between the bluetooth module and the FPGA. Seeing as the bluetooth module outputs bytes of information via UART communication protocol, it was important to ensure that the FPGA was capable of correctly interpreting the byte stream as a digital signal. In order to ensure that signals from the bluetooth module were expected output going into the FPGA, the group took note of the signal waveform by displaying it on an oscilloscope. Then, we connected the bytes output by the UART receiver module to the LEDRs on the DE1_SoC to see if they showed up correctly. After setting up the logic to interpret the output of the UART receiver module, we connected those signals



Figure 4: Instances of Serial Transmission from BLE (Signals to FPGA for ‘3’ in data field)

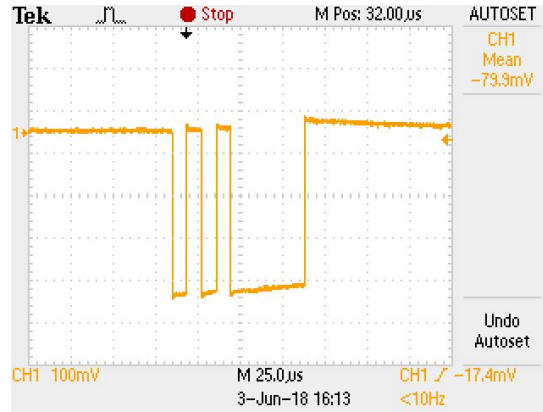


Figure 5: Instances of Serial Transmission from BLE (Signals to FPGA for ‘5’ in data field)

NOTE: The digital signal output shows a slight increase in amplitude as time increases. A faulty probe is the attributing factor most likely causing this deviation, however the signal does not affect the accuracy of command interpretation.

In developing and testing modules controlling game mechanics, testbenches and ModelSim was used only sparingly (examples can be found in the Implementation Section). This was a result of a lot of the essential game logic being developed in parallel to ensure proper integration and functionality, as well as the complexity of certain aspects, such as the numerous large arrays keeping track of the snakes’ movement or the video_driver for the VGA display, which made ModelSim less helpful for debugging. Previous experience using a VGA display with the video_driver module meant that there were little to no problems getting the game to show up on the monitor (regardless of whether the game logic was correct). Thus, testing of the game and overall system integration was done by repeatedly running and playing the game on the FPGA. Awkward gameplay, incorrect inputs, and thorough testing of different movements were used to check potential edge cases and find any bugs.

VIII. IMPLEMENTATION

Overall System

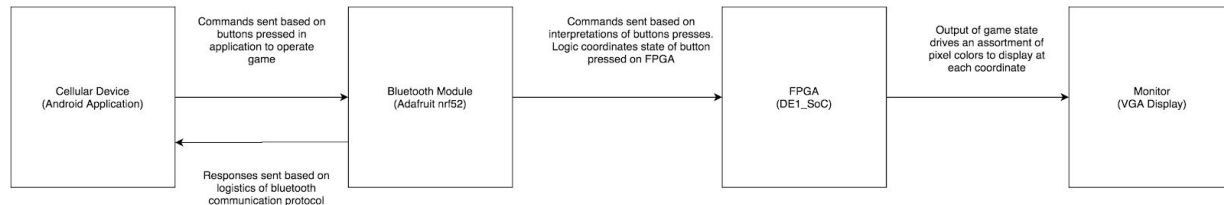


Figure 6: Top Level Block Diagram for all system components

The system consists of four devices: smartphone, bluetooth module, FPGA, and VGA display. The smartphone outputs commands based on button pressed within the application. These bytes are sent to the bluetooth module where they are interpreted and redirected to the FPGA via UART communication protocol sent along a transmission line. After the FPGA has completed its necessary logic computations and assignments, it will update the frame of pixels on the VGA display via the system's internal driver. These four systems are summarized below:

- Phone Application:
 - The application was implemented using Android Studio with Java language.
 - A game controller screen was designed for controlling movements of the snakes.
 - The game controller consists of four buttons for each player to control Up, Left, Right and Down movements and one additional button for both players to activate “Ghost Mode”. The game pad of Player 1 (Red) is flipped for convenience.
 - Each button sends an integer Android tag using UART buffer once the phone and bluetooth module successfully establish a connection and the button is pressed.
 - Every tag is unique ranges from 0-9 and helps to distinguish between the different button presses as per the communication protocol.
- Bluetooth Module:
 - Arduino controller module built on top of existing hardware implementation of the module.
 - Establishes serial communication with cell phone at a data transfer speed of 115200 baud (commonly used for Bluetooth applications)
 - Library functions perform operations such as powering up the transmission line, permitting using of the UART, and establishing a protocol for the service to for transmitting and receiving
- FPGA:
 - Modules implemented using hardware description (Verilog) to implement overall game mechanics and communication modules
 - Modules were developed in order to hold conditions for certain behaviors of the game, and drive peripheral output for the VGA display
 - Operates at 50 megahertz with uart and VGA display modules, while other modules that directly correlate with updating frames, snake states, etc. operated at only 12 hertz
- VGA Display:
 - Continuously updates frames with the pixel assortments set by the logic of the game
 - Updates at very fast rate, but is bottlenecked by the speed of the game to the users
 - Uses RGB indices to display a variety of colors that depict particular objects in the game

DE1_SoC

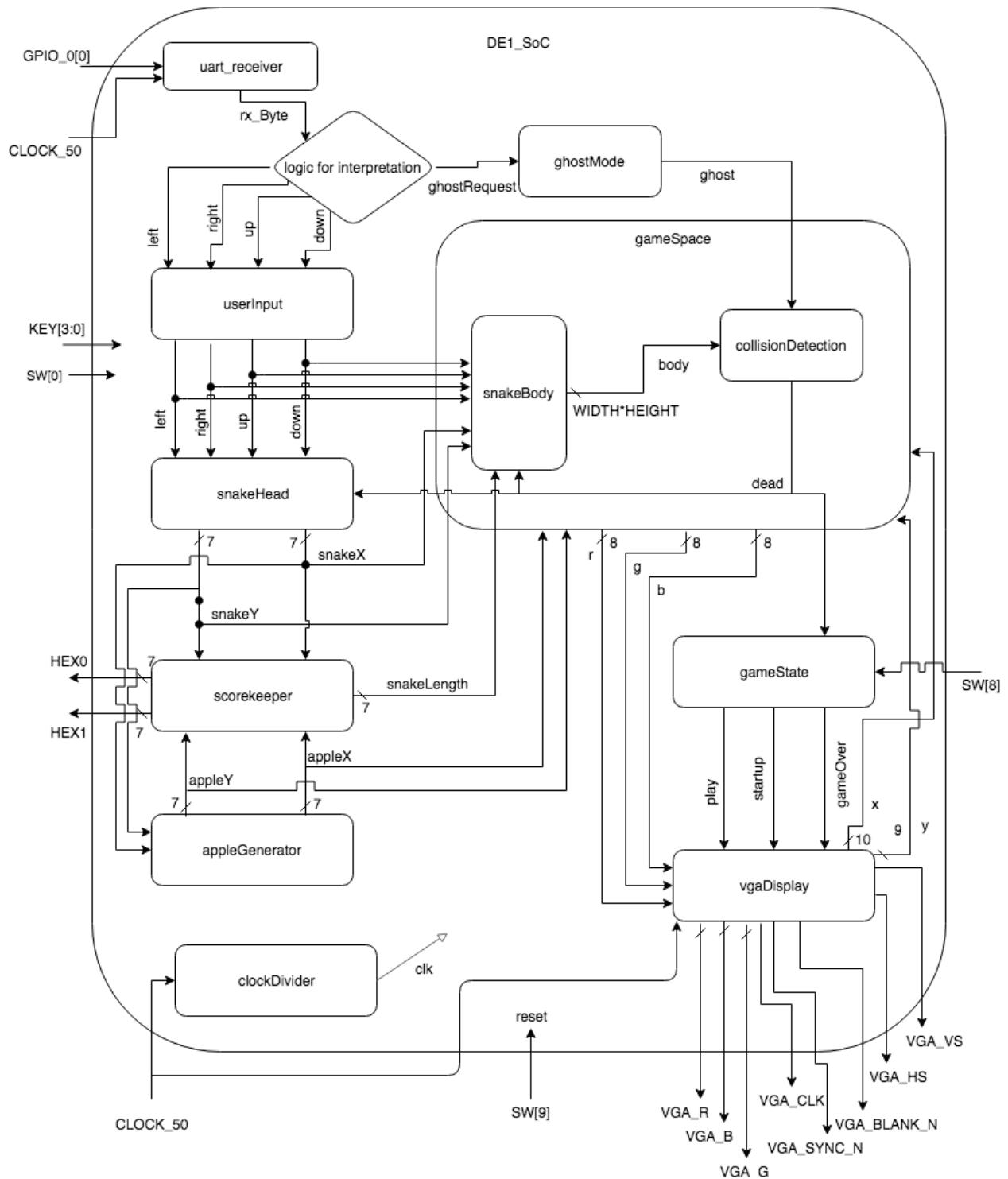


Figure 7: Block Diagram for Top Level Module on FPGA (showing modules for just one snake)

DE1_Soc is the top level module for all the verilog of the project. It handles the input (via GPIO) from the bluetooth module, with logic that reads the byte output of the uart receiver and determines which button was pressed on the app. These signals move on to userInput and other modules controlling the game logic, which influence the VGA outputs of the system. “Figure 7” above shows how the various modules fit together in DE1_SoC. Note that the figure only shows modules needed for one snake. There are actually 3 instantiations of the modules userInput, ghostMode, snakeHead, snakeBody, and scorekeeper, for each snake. Two copies of the userInput and ghostMode modules take inputs from the uart_receiver, while the third uses KEY[3:0] for controlling the snake, and SW[0] for ghost mode. CLOCK_50 is passed into the uart_receiver and vgaDisplay modules, while the rest of the modules get the divided clock from clockDivider. Reset is assigned to SW[9].

The DE1_SoC module also has logic for a grace period for each snake. This is implemented with basic two-state state machines and up-counters. The grace period remains true for the 4 clock cycles after the snake starts to move, during which the snake can’t die, to avoid accidental immediate deaths, which was more of a problem in an earlier implementation of the game.

Phone Application

The phone application was designed in Android studio to send tags to the Adafruit Feather nR52. The java classes that control the snake game controller are MainActivity.java, activity_pad.xml, PadActivity.java. The activity_pad.xml takes care of the layout view and controller of the game pad and also assigns the Android Tags for the buttons to distinguish their presses and the PadActivity.java contains the instructions for the controls. The layout of the app was hard to implement as buttons would not align properly on the screen. The MainActivity.java is used to display the set of initial options and connect to the Adafruit Feather nR52. The PadActivity.java extends the UartInterfaceActivity.java provided as open source by Adafruit Industries and uses an ArrayList representing the buffer to transmit the tags of the buttons to the bluetooth receiver based on touch listener. The rest of the classes also imported via the open source code provided by Adafruit Industries were just left as it is to avoid build issues when creating the APK and are not shown or used in the final version of the application.

UART Receiver:

The UART Receiver module was designed to interpret the UART signals sent from the Adafruit Feather nRF52. The UART Receiver is a simple finite state machine. The module stays in “idle” state until receiving a start bit (negative edge). The module samples the UART signals with the rate of 434 clock cycles per bit. Upon receiving the start bit, the module samples the input signal until the middle point of the bit. If the signal is still low as expected, the module has successfully recognized the start bit and the module should be prepared for receiving the data (switching to the “receiving data” state). During the “receiving data” state, the module samples the UART signals up until the middle point of each bit, the data is then sent to a “result” buffer. After finishing receiving 8-bit of data, the module will go into the “end bit” state which operates similarly to the “start bit” state. After receiving the end bit, the “result”

buffer will be output and the module will go into a “restart” state which prevents timing issues and provides a clock cycle delay for stability. “Figure 8” below illustrates how the UART Receiver operates.

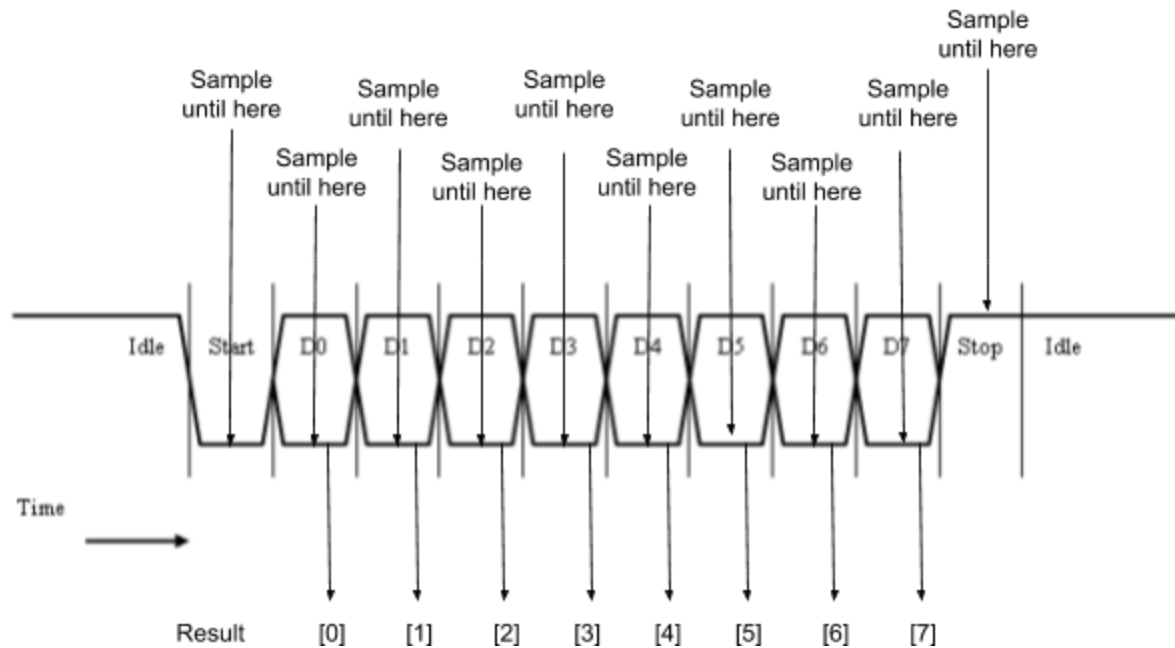


Figure 8: Operation of the UART Receiver

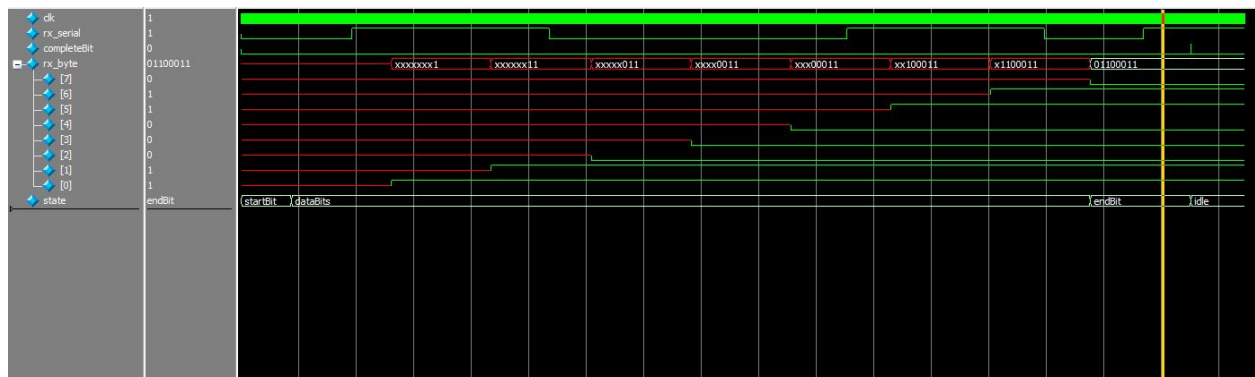


Figure 9: ModelSim simulation of the UART Receiver.

- The module was tested for its ability to interpret UART signals. In the simulation, a byte was serialized and sent to the UART Receiver. The module behaved as expected and was able to interpret the byte sent.
- As can be seen from “Figure 9” above, the module was able to interpret the byte “99”. The “result” buffer was updated from the least significant bit to the most significant bit after each bit is read. The “restart” state can’t be seen here because it’s only a clock cycle. Upon finishing sampling the UART signal, the module will output 1 to the complete bit.

userInput

userInput takes in and interprets user commands, and outputs the directions that go into the snake modules and control the movement of the snake. The inputs are either KEYS, or signals interpreted from the uart_receiver output. The module has two DFFs to deal with metastability with KEY inputs, while the outputs are driven by a state machine. At reset, the module is in an initial state in which all outputs are zero, and the user must input/press down to leave that state and go to a second, not_pressed state. From here, any direction can be input except down (to deal with how snake is initially spawned). There's a different state for each output direction (up, down, left, right), and only one output direction can be true at a time. The user cannot flip directions (go up from the down state, left from the right state, etc.), and the output direction doesn't change if two opposing directions are pressed at the same time. The modelSim waveform in "Figure 10" below shows this implementation.

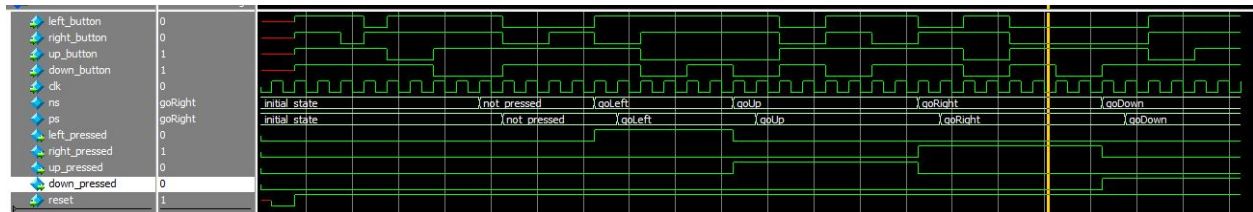


Figure 10: modelSim results for userInput module, with port names changed for clarity

snakeHead

The snakeHead module takes in the directions from userInput, and tracks the x and y coordinates of the snake head on the 32 x 32 block arena. "Figure 11" below shows a simulation of how this works (with only a 16x16 display). HEIGHT and WIDTH are parameters that determine the size of the arena, in blocks of pixels (provided video_driver module creates display with this resolution). STARTX and STARTY determine where the snake head spawns at the start of the game or reset. Two signals in the waveform below are not actually used in the final version of the module: the parameter PLAYER specified the starting coordinates in an earlier version of the module, while moveclk was part of the attempt at allowing different snake speeds.

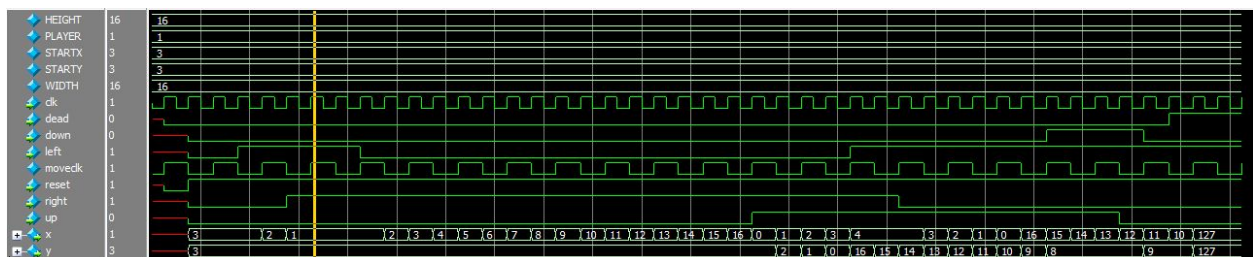


Figure 11: Modelsim Results for snake head state

The way this behavior was implemented was pretty straight forward. The snake starts at the specified coordinates, and the saved x and y values are incremented or decremented based on the direction inputs to move the snake head around. If the x value is at the maximum specified by WIDTH and the right input is still true (incrementing value), x goes back to 0, and from 0 going left x will loop around to the maximum (coordinates go from 0 to 31). The y coordinate behaves the same, returning to 0 from the max value when going down, or jumping to the max from 0 when going up. When the snake dies, the x and y coordinates are set to 127 (7'b1111111), outside of the arena.

snakeBody

This module is responsible for keeping track of where the entire body of the snake, and making sure it stays the same length as it moves around, except for increasing in length by 1 block when it eats an apple. It takes in the x and y coordinates of the snake head and the apple, the signal specifying if the snake is dead, and the user-controlled directions output by the userInput module. It uses parameters to specify the width and height of the game arena, as well as the starting x and y of the snake head (should be set to the same as STARTX and STARTY of the respective snakeHead module). The output is an array the size of arena/display, where a 1 represents the snake body occupying that coordinate's block, and a 0 means the area is empty.

To correctly fill and update this body array, the module keeps track of x and y coordinates for a tail, which is the block after the last piece of the snake body. The module also uses an additional array that has two bits for every space on the arena, which stores the direction (up = 00, right = 01, left = 10, down = 11) that the snake head was moving as it left that spot. Combinational logic is used to assign the current direction specified by the user, and to pull out the direction stored in the directions array at the coordinates of the tail.

Then an always_ff block uses these and the inputs to update the two arrays and move the snake body. At reset, the body array is set to all 0, except for the starting x and y of the snake head, and two segments below that. The block after those is assigned to the initial tailx and taily. The directions array is initially set to all 2'b00 (up). As the snake head moves around, the body array's value at the head's location is set to 1, while the location of the tail is set to 0. The current direction assigned in combinational logic is put into the directions array at the index of the head x and y. The direction pulled from the array at the tail location in comb. logic is used to move the tail x and y, in the same way that the direction inputs control the head movements in the snakeHead module. If the snake x and y and apple x and y are equal, meaning the snake has consumed an apple, then the tail location does not update, staying at the same spot for one clock cycle to allow the snake to increase in length by 1. When the snake dies, all logic regarding the tail is ignored and the entire body array is set to 0.

appleGenerator

The implementation of the apple generator used preconditions revolving around the locations of the snakes during each clock cycle at game speed. A 10-bit LFSR was used to create a randomly generated set of bits that would be used to dictate the new set of coordinates for the apple. Half of the bits equated to the coordinate of the apple along the x-axis, while the other half equated to the coordinate of the y-axis.

During the initial state of the game (reset), the apple's coordinates are set to preset values in order to prevent a potential unfair advantage to a particular player during the early stages of gameplay.

The coordinates of the apple are compared with the coordinates of the heads of all snakes in order to evaluate if a player has successfully "consumed the apple". If this condition is satisfied by any of the three players, then the apple's coordinates are reset and generated in a new location based on the current instance of bits from the random number generator.

The logic block for this module has no terminating condition besides the reset case (as this an ongoing execution during the span of the game).

scoreKeeper

This module is responsible for updating the state of score for all players. The module utilizes the "counterToNine" in order to increment the value of count and properly account for carryovers to produce double digit values. The module also uses "twoseg_7", a submodule responsible for displaying the values on the HEX displays of the FPGA.

This module uses a continuous assignment to calculate the score based on the length of the snake, a value that is updating independently and passed in. This value dictates the score of the player by simply subtracting the length of the snake at the start of the game (e.g. length of 3). This works because of the effect of collecting apples incrementing the length of a snake, indirectly producing a status update that is utilized here as well.

The only logic block in this module is a combinational logic block that essentially acts a boolean on whether the score is allowed to increment in the first place. If the game ends due to a victory, the state is set back to false, or "incr_no", to set the score back to 0 for all players, and prevent it from being incremented again until the game has been set back to a state where it isn't "over" anymore.

collisionDetection

This module keeps track of whether all of the snakes are alive or dead. It takes in the location of each snake's head, as well as the body array representing each snake, from the snakeBody modules. Each snake gets a two-state state machine: alive or dead (defined in basic case statements, with sequential logic for updating state). Once reset, every snake is in the alive state. The module looks at each of the body arrays and head locations, and if a snake's head ever reaches a point occupied by its own body, or another snake's head or body (based on whether a body array is 1 at that head's coordinate), the snake is set to dead, and the "dead" output for that snake is set to 1. There are exceptions for this death if the snake's collision happens while that snake's ghost signal is true and the collision is not with itself, or if graceperiod is true in any situation. Each snake is being checked at the same time in combinational logic, so multiple collisions can occur at once, and if two snakes collide head on, then both snakes will die.

gameSpace

This module handles tracking where the full snakes are, checking if they collide, and outputting the color values to the vgaDisplay based on this. The module takes in x and y coordinates, snake length, four directions from userInput, a signal for the grace period, and a signal for the ghost mode for each snake (3 different versions of each of these inputs). It also takes in the x and y of the apple, and the x and y from the vgaDisplay (the coordinate the display is requesting color values for). Besides the clk for the module, it also has inputs for clks for two of the snakes, and the collisionDetection module, to specify them from the top level module, which were used in trying to implement different snake speeds but is currently just accepting the same clk as the other modules.

As shown in Figure 7, gameSpace includes instantiations of snakeBody for each snake, as well as the collisionDetection module. It takes the body array modules from snakeBody and passes them to collisionDetection. Meanwhile, combinational logic in gameSpace looks at the apple coordinates, and each of the body arrays at the x and y values passed in (checking what occupies the space at (x,y)), and based on those a certain set of r, g, b values are output from this module to vgaDisplay. It also outputs the dead signals for each snake that come from collisionDetection.

gameState

The inputs are reset, clk, start, and dead signals for each snake. The outputs are signals for startup, play, and gameOver. The module uses a basic state machine to track the state of the game between starting, playing, and finished. At reset it's in the starting state, and remains there until start is true, at which point it switches to playing. It stays in playing until only one or fewer dead signals is false (meaning one or zero snakes are alive). The parameter PLAYERS determines how many dead signals to check. This is set to 3 in our project so that it leaves the playing state when 2 dead signals are true, but if PLAYERS was set to 1, then it would leave this state as soon as dead1 is true. From the playing state it goes to the finished state, where it stays until the game is reset. The output startup is only true in the starting state, play is only

true in the playing state, and gameOver is only true in the finished state. These signals are used to by other modules to know what screen needs to be displayed.

ghostMode

This module is responsible for signalling the other game modules to activate the ghost ability, where a snake can pass through other snakes. While collisionDetection is where ghost mode actually affects snake behavior, this module is what outputs the ghost signal for a limited amount of time based on user input and counters that manage how long the output is true, how long until it can be activated again, and how many times it can be used. Parameters are used to set the number of uses, the cool down time, the time the mode remains active, and to inform the module of the clock frequency used so it can properly use those times.

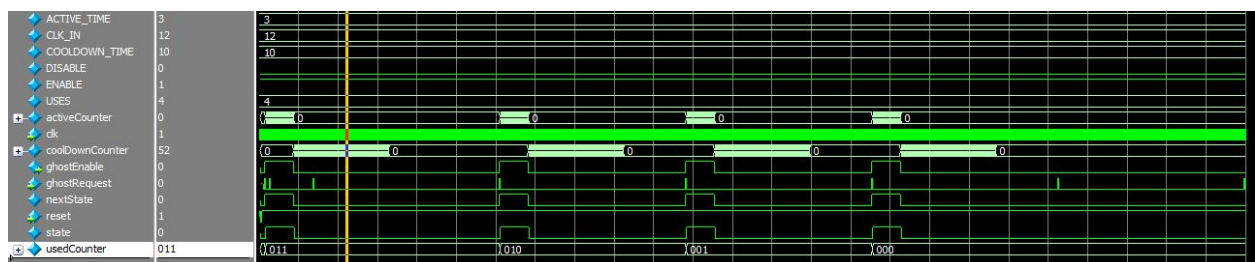


Figure 12: Modelsim Results for ghost mode (overall simulation)

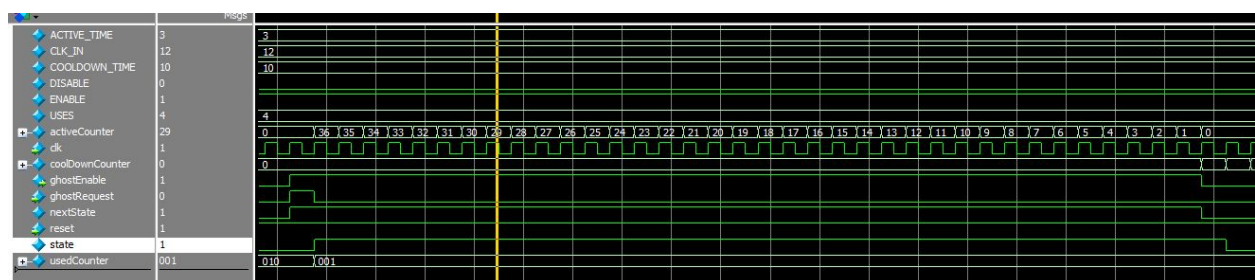


Figure 13: Modelsim Results for ghost mode (emphasis on decrementing behavior)

As shown in simulation in Figures 12 and 13, the module successfully activates a decrementing counter depicting the duration of the active state for a snake's powerup. The module accounts for decrementing the amount of uses for the powerup, and the reactive initiation of the cooldown timer. When the active timer reaches zero, the user cannot activate the powerup again until the subsequent cooldown timer has reached zero as well (assuming that the user still has uses, and that the use counter hasn't reached zero).

vgaDisplay

The vgaDisplay module is responsible for outputting the necessary VGA signals to send out of the FPGA to a VGA monitor that displays the game, or the start/end screens. These outputs include VGA_R,

VGA_B, VGA_G, VGA_BLANK_N, VGA_SYNC_N, VGA_CLK, VGA_HS, and VGA_VS, and the module also outputs the x and y values that it currently wants the r, g, and b values for (for the game display). The module takes in the same reset as everything else, CLOCK_50 (not the divided clock), r, g, and b values for the given x and y, and signals specifying that the game has started (startup), that the game is currently underway (play), and that the game has ended (gameOver). Parameters can be set for the WIDTH and HEIGHT of the game arena/display, as in other modules. As “Figure 14” below shows, the module uses two different instantiations of the video_driver module provided to us to display 3 different screens it can switch between.

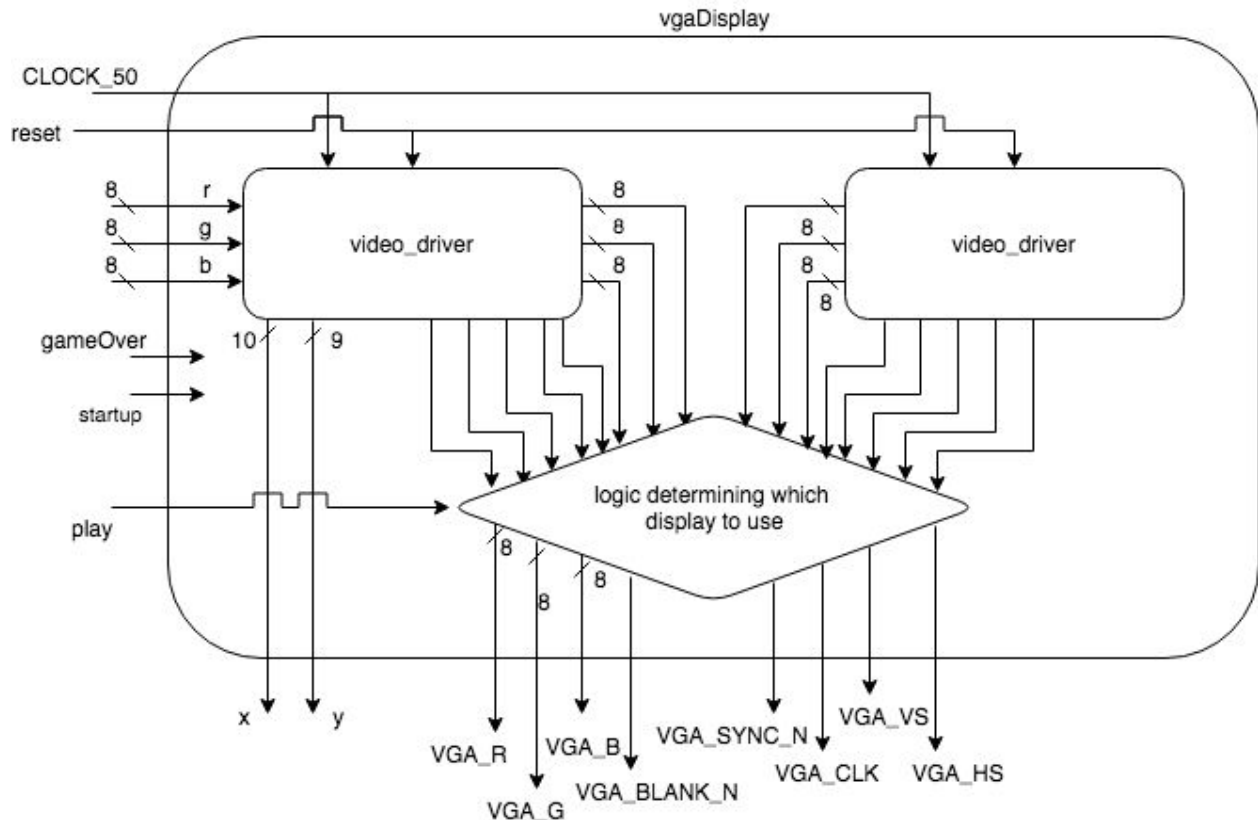


Figure 14: Block Diagram of the vgaDisplay module

One of the video_drivers is responsible for the display used while actually playing the game, and is set to the same width and height as the upper level vgaDisplay and other game-related modules. This is the driver that outputs the x and y values that vgaDisplay outputs, and takes the r, g, b values that get input. The other video_driver produces a 64x64 display that is used for a start and gameover screen. Arrays are defined for those two screens and the r, g, and b values passed in to the respective video_driver are one of two options based on a 0 or 1 in that array. At reset, the title screen video_driver gets the inputs for the start screen (one of two states in a state machine), and holds this state until the gameOver input becomes true at which point it switches to the gameover/end screen. Both video_driver modules are always running in parallel, but the outputs of the vgaDisplay module are taken from one driver at a time, based on a simple if statement. If the play input is true (the case from when the start switch has been triggered to

when the game ends), the vgaDisplay outputs are based on the game screen video_driver, and if play is false, the title screen video_driver's outputs are used.

IX. RESULTS

Overall Game Description:

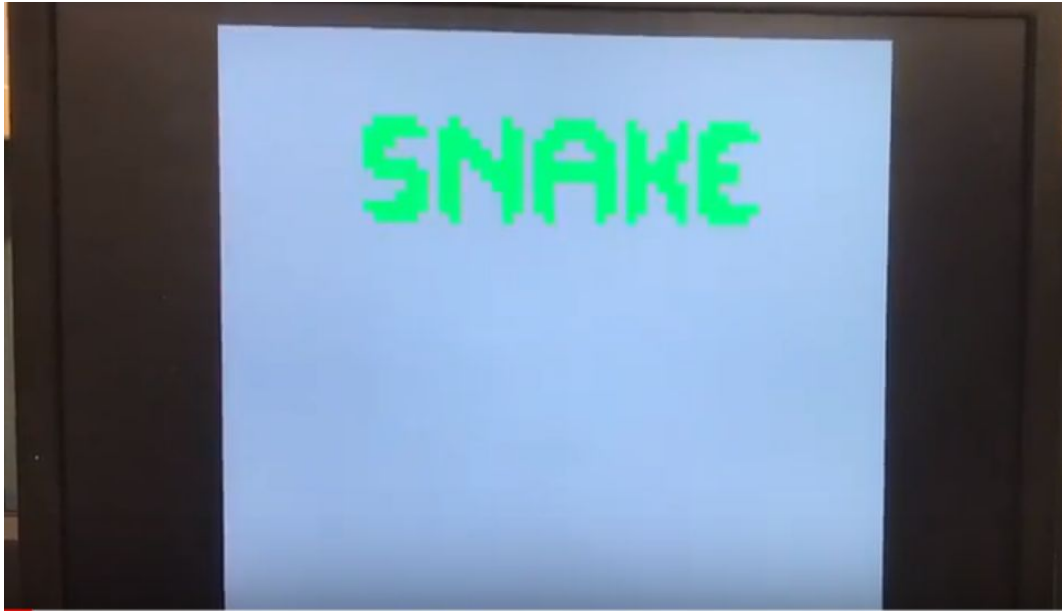


Figure 15: Start Screen for the game

The FPGA is reset to its initial state, shown in Figure 15 above by toggling a switch and bringing the game to a state where it simply shows the start screen. This screen remains displayed on the monitor until another switch denoted as “start” is toggled high, after which the initial state of gameplay is set, as shown in Figure 16. Players can see their snakes on the screens, but can not move in any direction until they have initialized play by pressing their respective “down” buttons. Once they have done so, players can begin moving in any direction besides the one opposite to their current direction of movement (first move cannot be down). In other words, Players can utilize the four directional pad and buttons to rotate their snakes by 90 degrees depending on the direction that they are moving in.



Figure 16: Game Arena at Start of Game

Players will now aim to collect apples, and avoid elimination in order to remain in play. Collecting apples will increment the score of the player, although the primary objective of the game is to stay alive by avoiding other snakes. Players can be eliminated if their individual head pixel makes contact with any body pixel on the screen. If desired, they can use a power up called “ghost mode” that grants them invulnerability to death by other snakes (not themselves) for a small period of time. Players must wait for a period of cooldown time before using the power up after its duration, and the power up can only be used a limited number of times.

When only one player remains in play, the game will switch to its “game over” state where a screen displaying the words “game over” will appear (Figure 17). The game can only be restarted when the “reset” switch is toggled once more.



Figure 17: End Screen for the game

Phone Application:

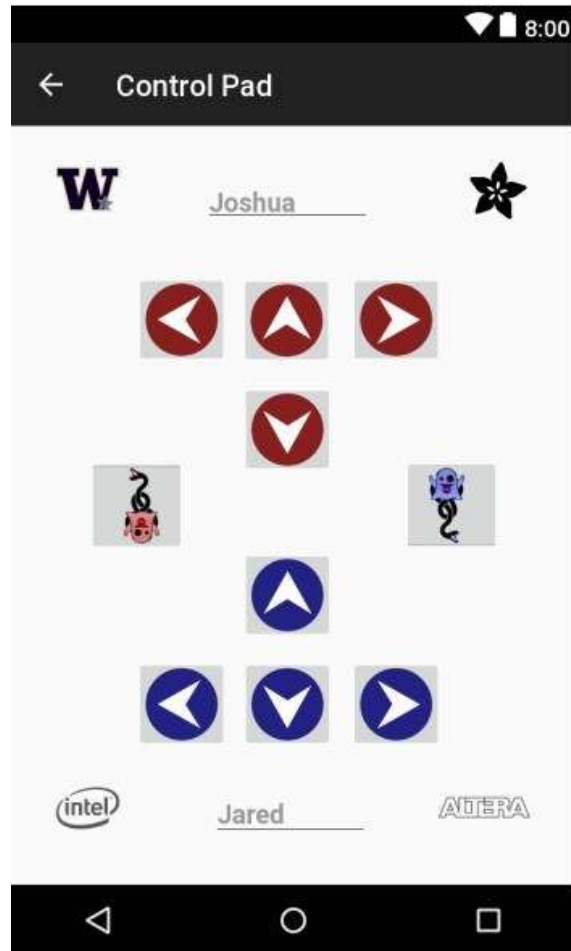


Figure 18: Android Application - Game Controller

As shown in Figure 18, we ended up with a properly functioning app with a very simple layout. It provides controls for 2 players (who can add their names if desired) to move the snake or activate the ghost ability. Also:

- Evaluated all buttons to validate whether there was a valid response, and whether it was produced in a timely manner.
- The user interface provided functionality needed to take full control of their individual snakes.
- Buttons on the screen were responsive and promptly issued commands.
- The Android app was difficult to implement across various available screens of different sizes.
- Successfully established connection between the smartphone host and bluetooth module.

Bluetooth Module:

- Successfully received command output from Android application via Bluetooth after fixing the initial bug in the open source code obtained from the manufacturer's website.
- Interpretation correctly maps out button presses to command protocol for game mechanics modules on the FPGA to understand
- Successfully outputs command bytes based on command protocol to the FPGA via UART communication.

VGA Display:

- A 32x32 pixel display for the game, as specified, and a 64x64 pixel display for the start and end screens to show text in slightly more detail.
- Proper connection and communication from the driver on the FPGA ensured that pixel assignments were instantiated in the correct coordinates and with the correct color, as shown in the previous figures showing what is displayed.
- Display seamlessly updates frames with the correct states based on the expected frame rate.

X. FUTURE WORK

The most straightforward expansion of this project would be adding new players. Initially we designed for 4 players, but cut back because we only had one Adafruit Feather nRF52. The parallelism of the FPGA, along with how our game logic is set up, makes it fairly simple to add new players in Verilog, simply by instantiating more versions of the relevant snake modules, and adding cases or if statements for collision detection, assigning colors, generating apples, etc. (mostly a matter of copying and pasting with slight edits) so that each snake is incorporated. At that point it's just a matter of obtaining additional bluetooth modules and having phones to use them with. The placements of the snake at the start of the game, and the size of the game arena are easy to change to allow more players on the screen. Also, obtaining multiple bluetooth modules could allow for each player to use their own device/controller, rather than having two player share one phone (which we did to avoid the cost of an additional Adafruit Feather nRF52).

Other improvements or additions to this project could include:

- Improve graphics on VGA display (e.g., using sprites and so on to make game more detailed)
- Allow user profiles and saving scores in Android Application with high score option.
- Add additional powers that the snakes can use when particular conditions are met, such as firing projectiles in exchange for losing length, or gaining increased speed for a limited amount of time.
- Add new modes to the game, such as Tron (snake heads leave "body" trails that don't disappear, instead of eating apples to grow), or add other new kinds of obstacles to avoid.
- Have an iOS based app for Apple iPhone™ and Apple iPad™ users and allow cross-platform usage.

XI. CONCLUSION

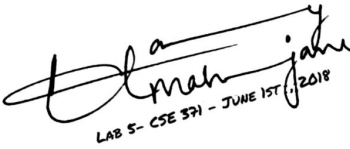
This project has given the group a greater understanding of the FPGA's potential as a platform within a system of multiple devices. Progressing from projects that utilized its computing power as a standalone entity, the team has incorporated multiple devices and implementation techniques to produce a project of much greater capability. This implementation of wireless multiplayer snake has proven to be very proficient as a game overall, with compatibility for expansion at a level that surpasses traditional implementation on CPU-based platforms. With that being said, the team has grown to appreciate this device's capability to compute in parallel, and plan to leverage this towards potentially expanding the project, and applying the device to other projects in the future.

XII. APPENDIX

All code and/or relevant documents are included in the ZIP folder submitted with this laboratory report as suggested.

XIII. INDIVIDUAL CONTRIBUTIONS

We hereby state that report and contents of the lab files are solely our own work.

<p><u>Nguyen Lai</u></p> <p><i>Nguyen Lai</i></p> <hr/>	<p>Set up bluetooth module and worked on its integration with the FPGA and helped with setting up the communication protocol between phone application, bluetooth module and FPGA and helped with system integration.</p>
<p><u>Uday Mahajan</u></p> <p> LAB 5- CSE 371 - JUNE 1ST, 2018</p> <hr/>	<p>Created and designed the phone application with communication protocol used to interface phone application vs bluetooth module to play the game and helped with system integration.</p>
<p><u>Haytham Shaban</u></p> <p><i>Haytham Shaban</i></p> <hr/>	<p>Worked on integrating bluetooth module with the FPGA, peripheral game modules, and system integration.</p>
<p><u>Samson Waddell</u></p> <p><i>Samson Waddell</i></p> <hr/>	<p>Created the main game logic (snake movement, collision, etc) and logic for outputting to the VGA display, and helped with system integration.</p>

Thank you for a great quarter and have a great summer and beyond.