Verilog Snake Game

Sai Praneeth Maddi – 16110145 sai.praneeth@iitgn.ac.in

Jayakrishna Sahit – 16110104 jayakrishna.sahit@iitgn.ac.in

K. S.V. R Rithwik – 16110083 kukunuri.sai@iitgn.ac.in

Chakka Snehith – 16110039 chakka.snehith@iitgn.ac.in

*Abstract*

Our Project involves using **Verilog Coding** along with **Xilinx Spartan-3e FPGA,** with the help of **Xilinx ISE**, in order to create and implement the classic Snake Game which we displayed/output with the help of an 8X8 LED Matrix. The Game involves a virtual snake and the object of the game is to be able to eat as much food as possible, before the timer runs out with the additional constraint that the snake cannot overlap with itself ie, if the snake collides with itself, the score you have attained is the score of the game irrespective of the timer. We implemented the following in **three basic steps – 1. Creating the Virtual Snake and displaying it; 2. Generating the Random Food; 3. Adding Constraints** to the game which include the increase of the score upon eating the food, the timer which ends the game after the time given is over and finally the crash logic, which ends the game if the snake crashes into itself. The estimated **time for completion of this project was about 8 weeks**.

(Note that the organization of the body of the paper is at the authors’ discretion; the only required sections are Introduction, Methods and Procedures, Results, Conclusion, and References. Acknowledgements and Appendices are encouraged but optional.)

*Index Terms*— Behavioural Coding, Binary Output File, Clock, Field Programmable Gate Array(FPGA), LED Matrix, Mapping File, Seven Segment Output, Shift Registers, Verilog, Verilog Module, Xilinx ISE

*Introduction*

For the following project, we decided upon implementing a game in order to understand the complex inner circuit mechanism and the logic behind the earliest games which were developed in order to be able to connect it to how games are presently using computer software, to expand their capabilities. After extensive research across various games, we finally decided on going with the snake game, as it was a game which was logically complex however easy to display so that we could emphasize more on the logic and hence more on the hardware implementation part.

In order to implement this game on a **Field Programmable Gate Array(FPGA**), we used Verilog coding, the language in which we had prior knowledge in and had a firm grip with. In order to code this and be able to deploy it, we used **Xilinx ISE**, the best software in the market for the deployment of Verilog onto an FPGA. The FPGA we used was the **Xilinx Spartan-3e FPGA.** And lastly, we had decided to use an LED Matrix display for displaying the output.

In order to implement the snake game, we divided the project into three phases as mentioned earlier. These are:

* Creating the Virtual Snake
* Generating Random food
* Adding the constraints to the game which include
  + Score increase on Eating logic (Eating Logic)
  + The limited time after which the game will reset/end logic (Timer Logic)
  + The rule that the snake must not overlap with itself at any point of time (Crash Logic)

However, before all of these, the initial step we had to do was

* Figuring out the display on the LED Matrix and the Seven Segment Display (BCD display)

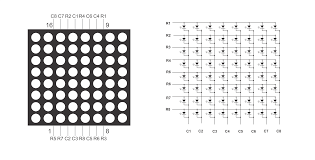
Through deep understanding and proper implementation of all four steps, we were successful in being able to create the classical Snake Game using an FPGA and Verilog Coding.

*Procedure*

In order to go about creating the game, our initial step was about displaying of the snake using the **LED matrix** and also the score using a **Seven Segment output**.

**Display Using the LED Matrix**

The LED Matrix we had decided to be used for the project is an 8X8 matrix. The pin configuration of this matrix is as the following.

image source - http://www.circuitstoday.com/interfacing-8x8-led-matrix-with-arduino

The LED Matrix has 16 pins in total, 8 anodes and 8 cathodes shorted accordingly. Individually controlling each and every LED on the Matrix is not possible, due to the way it is manufactured, however, each row or column can be set individually. Hence in order to display on the LED matrix, each row or column must be displayed faster than the human eye can perceive the change, several times a second so that the human eye is fooled into believing that all of them are glowing at the same time, which on the contrary, are not.

In order to do this, we assigned a 2D matrix register of the size 8X8 with row numbers decreasing from up to down and column numbers decreasing from left to right, so that each individual cell of the register represents each LED on the matrix. That way whenever we want an LED to glow we assign it a 1, and if we don’t want it to glow we assign it a 0.

*For example, we assign*

*reg [7:0] display [7:0]; now if we want the LED on the fourth column(from right) and third row(from bottom) to glow we say,*

*a[2][3] = 1; ( since assignment starts from 0 )*

And now in order to display this matrix, we display each row or column multiple times in a second so that it seems that all the rows are lit at the same time. In order to this, we assign another 1D register which represents each of the cathodes of the LED Matrix. We then run a for loop for each of the columns, where for the respected column, the respected cathode is turned to 0 so that it's in forward bias, while the others are turned to 1 so that they don’t glow and put into reverse bias. The anodes are given the configuration of the respected column in the 2D matrix register.

*Reg [7:0] column;*

*Integer i;*

*Intial begin column = 8’b11111111; end*

*for ( i = 0; i < 8 ; i = i + 1) begin*

*column[i] = 0;*

*output = display[i]*

*end*

This way with the help of the edge of the clock, we can display each row for about a millisecond, which is enough for the human eye to be tricked.

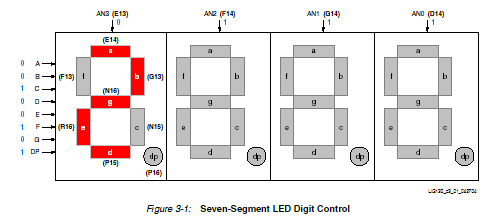
The Seven Segment Display too works in the same as the LED Matrix works. Its pin configuration is as the following,

Image source - http://ollintec.com/SistemasDigitales/libros/FPGA%20Piano%20in%20VHDL.htm

Each of Seven Segment output has common Cathodes and different anodes, so we do the same thing in order to display all four of the seven segment displays. However since all of the four numbers are linked to different parts of the program, we do not use a 2D matrix register, instead, we directly code them as separate 1D linear registers. These Registers will be decided in a later part of the project.

**Creating the Virtual Snake**

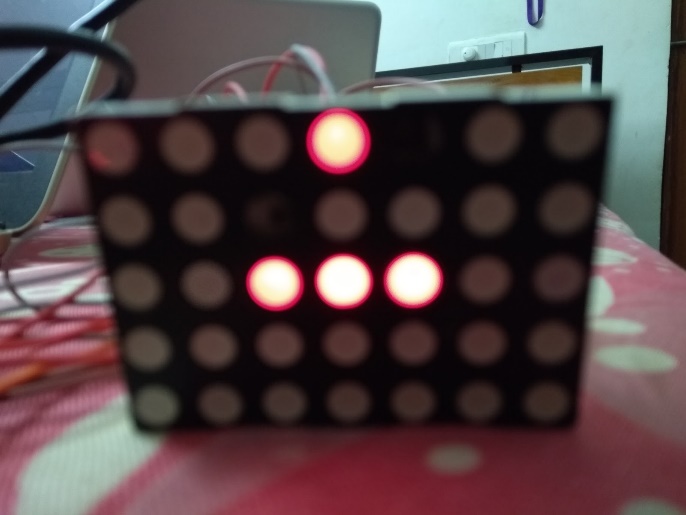
The Snake we had decided to make would have the following features

* The Snake would be a represented by a combination of LEDs on the LED Matrix, all in a single line, either horizontally or vertically, depending on the game setting at that moment. It can be seen in the following picture.
* The Snake is to be controlled using four input buttons present on the FPGA, which help in the direction change function of the snake with a fixed button configuration ie, Up, Down, Left, Right would be fixed.
* It is also to be noted that as the snake cannot overlap itself, the movement of the snake cannot be reversed, for example, if the snake is moving in the upward direction, then the downward button becomes non-functional through the period that the snake is traveling in the upward direction.

For coding the snake we use Verilog coding for it, and **Xilinx ISE** to write it down, simulate it if necessary and to create a bit file so that it can be dumped onto the FPGA. In Verilog, the main type of coding we will be using is **behavioral coding**, which allows us to logically be able to create our snake, rather than use hardware components independently to compile the code. The code will be powered the standard 50Mhz clock on the FPGA.

As the snake we will be creating will have a fixed length of 3 units, we will assign a 2D matrix register, the number of rows being 2, which correspond to the X coordinate and the Y coordinate and the number of columns corresponding to the length of the snake.

*reg [1:0] snake [n-1:0], where n is the length of the snake(3).*



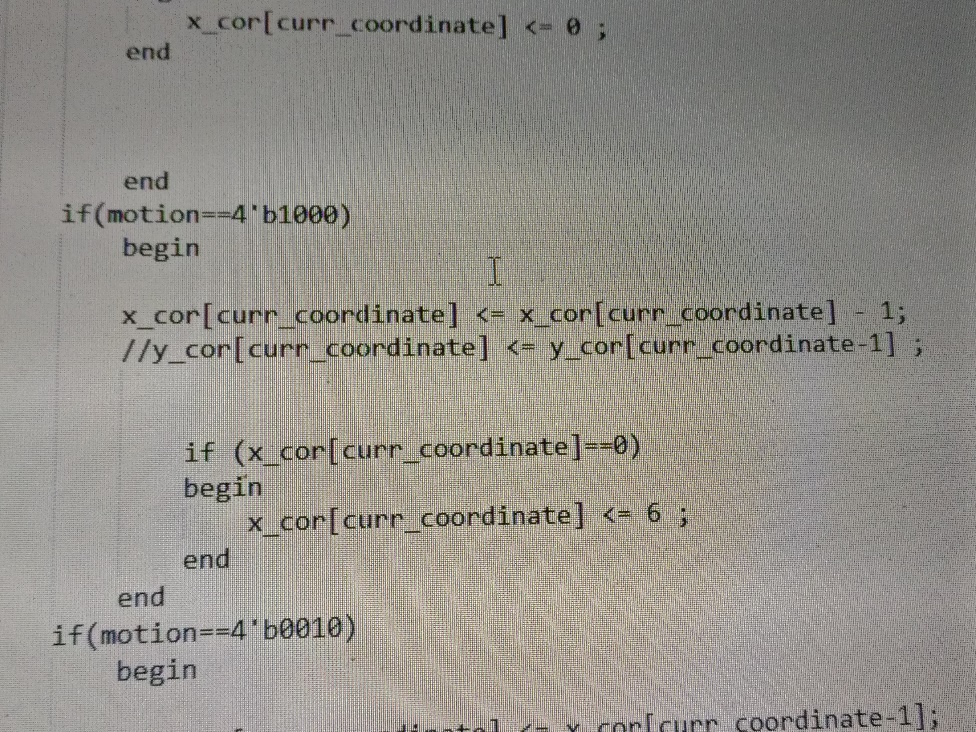
Each of these registers will hold the coordinates of all the parts of the snake at any point in time. These are then copied respectively into the LED Matrix display logic and printed.

The next task is the movement of the snake. In order to do this, we define four states of snake movement using a 2bit register (because 2 bits can represent four states ), each for up, down, left and right. We then each state separately as the following

**Up(2’b00)** – each of the Y coordinates is incremented by 1 (as the Y coordinates decrease from top to bottom and we take care that after it reaches the maximum value of 7, it becomes 0.

**Down(2’b01)** – each of the Y coordinates is decremented by 1 and we take care that after the value reaches the minimum value of 0, it becomes 7.

**Left(2’b10)** – each of the X coordinates is incremented by 1 (as the X coordinate decreases from left to right) and we take care that after it reaches the maximum value of 7, it becomes 0.

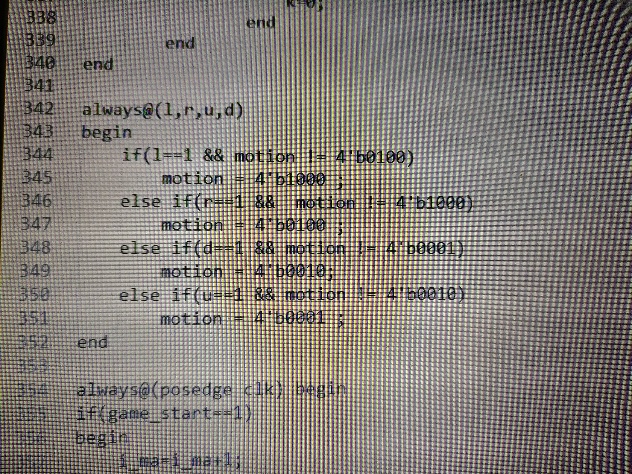
**Right(2’b11)** – each of the X coordinates is decremented by 1 and we take care that after the value reaches the minimum value of 0, it becomes 7.

We then map the input buttons to the states, which will be done in the later stage of the project.

We now we look at the third and the final condition, that prevents the user from giving an input which will make the snake overlap with itself. This happens when the user inputs down when the snake is moving up and the vice-versa and when the snake is left and the user inputs right and the vice-versa. For this, we had an additional when changing states.

*For example -*

*If (input = 2’b00(Up) && state != 2’b01(Down))*



And have the following steps we have now finally completed creating the virtual snake and the necessary features in it with it. We now move on to creating a program to generate random food.

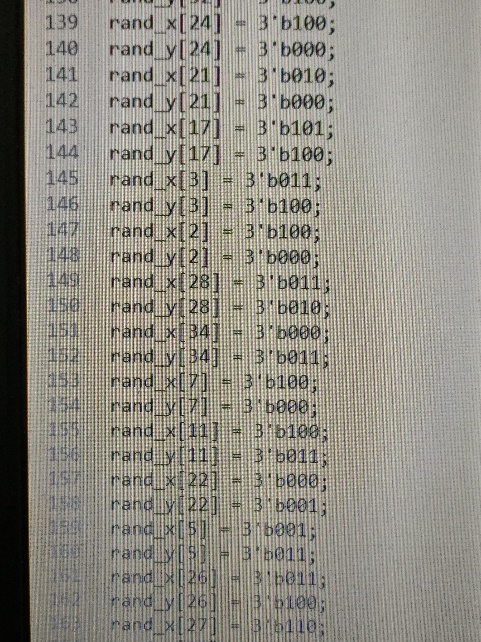
**Generating Random Food**

Due to the inability to create a truly random function on hardware, we hence use our own random function in order to place the food at a random place after every iteration. In order to do to this, we create 64 variables, each corresponding to each position of the LED Matrix. We then randomly map these to a time variable.

For example,

*T34 = matrix[2][4];*

*T35 = matrix[6][0]; and so on…*

Based on this, we combine it with the time logic to provide the food, which will be talked about in the Time logic part of the report.

**Adding Constraints to the Game**

We now move to adding the final three constraints of the game. They are:

* Creating the Score logic - making sure the score increases after the user is successfully able to make the snake eat the food.
* Adding the timer – to reset the game after the time for playing the game is over ie, creating a time limit for the game.
* The Crash Logic – making the game end after the user mistakenly makes the snake overlap with itself during the course of the game, causing the game to end.

To create the **Scoring logic** of the game, we basically need to see if the head of the snake coincides with the food coordinates on the LED Matrix as this will signify that the snake has eaten the food. We hence use a simple if condition for this

We first we need to define the head coordinates for the snake, which is simply given by

*head\_x = snake[0][0];*

*head\_y = snake[0]1];*

we then implement the if condition by the following code.

*If ( food\_x == head\_x && food\_y && head\_y)*

*score = score + 1 // simple representation*

The score in order to be displayed on the Seven Segment Display, the score value is divided into two parts 1. The Ten’s Digit 2. The Unit’s Digit. Each of these parts is then a 4bit register to count all values from 0 to 9. Hence we define the following

*reg [3:0] score\_tens; and*

*reg [3:0] score\_units;*

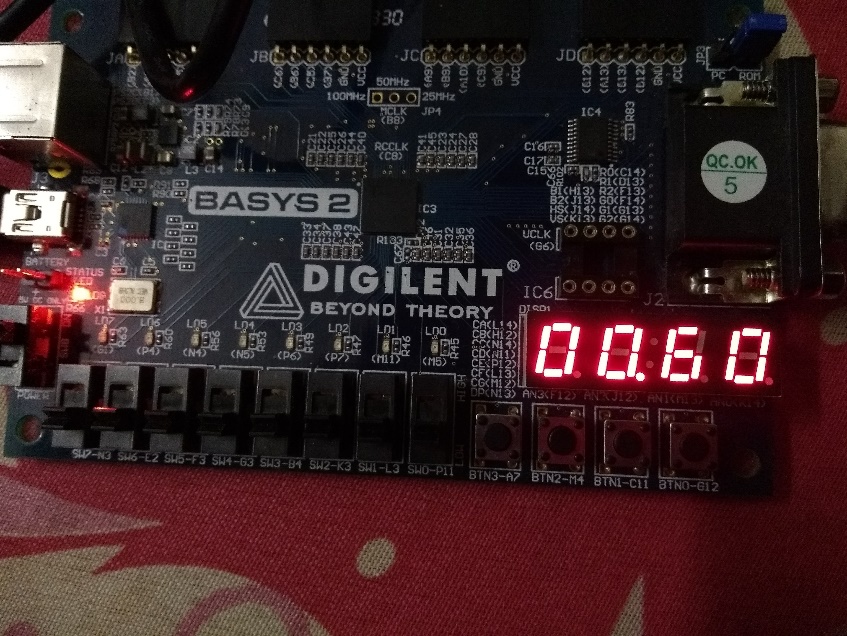
Now we add the **Timer Logic** which is also a simple step, by simply setting the timer to initialize to a value, let's say 60 seconds and then counting down till it reaches 0. After 0 the score then resets to 0, for the user to play the next game. Similar to how we divide score into 2 registers of 4bits each in order to display them, we divide the timer into two parts

*Reg [3:0] timer\_tens;*

*Reg [3:0] timer\_units;*

*And for the score to reset to 0, we define a simple logic for it*

*If ( timer == 0)*

 *score = 0; // simple representation of it*

However, we have another thing to code, which we do because of the fact that we chose the timer and score values to be represented by two registers of 4bits, each representing the tens and units digit of the variable respectively, we now must additionally write the increment /decrement logic of the following.

For this we hard code it, to make sure the bits change after the units digits reach 9 ie,

*For the increment logic (for the example of score logic)*

*after score\_units == 9*

*score\_units = 0; score\_tens = score\_tens + 1;*

*similarly for the decrement logic (for the example of time logic)*

*after timer\_units == 0*

*timer\_units = 9; timer\_tens = timer\_tens - 1;*

Now we add the **Time Logic** to the game and this we incorporate with the food logic. In order to incorporate the food logic into the separate, we create a separate food counter or the actual timer of the game, which tracks the time from when the game has initialized. We make sure that this timer resets after every 64 ticks, the number of positions presents on the LED logic. We then map the food counter/actual counter to the random function of the food and code it such that the time at which the previous food is eaten determines where the next food will be placed.

*For example, if the food is initially at matrix[2][3], and the snake eats this food at actual\_time = T45, the next food then appears at the place which is mapped to variable T45.*

This position is naturally updated to the display matrix and printed every cycle.

We Finally create the **Additional Crash Logic.** This naturally works for snakes which have a length more than 4, where the snake when it collides with itself, must end the game as it is an invalid move. We simply do this by checking the coordinates of all the snake parts and check if all of them are unique. If any two of them happen to be the same, it will indicate that the snake has overlapped with itself and hence has crashed, thus ending the game. We do this through a simple search and compare function.

*for ( i = 0; i < length of snake; i = i + 1)*

*check uniqueness of snake[i];*

if at all the snake has crashed the score and the timer are then reset back to their original values.

The logic development has been completed and now the next part of the procedure would be to dump the file onto the FPGA, which would be explained briefly in the next section.

**Dumping the Code on the FPGA**

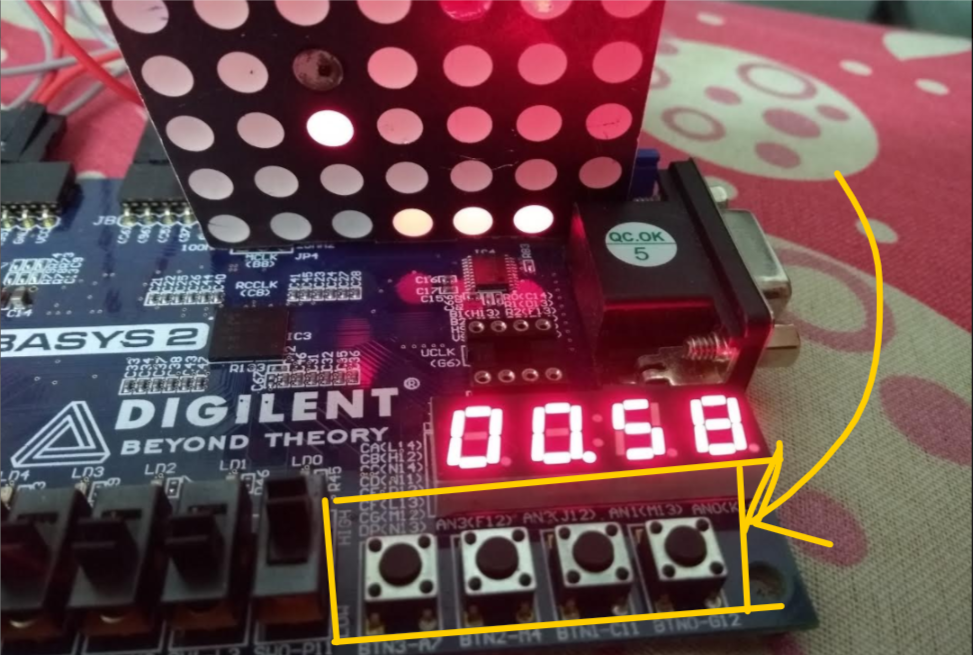
After the following logics have all been implemented using Verilog coding, we now need to dump the file onto the FPGA, to which we need to create two additional files, one being the Mapping file to link the outputs of the FPGA and finally the bit file, which the software automatically creates but it is the main file which will be dumped onto the FPGA.

The Mapping File is the file which writes the mapping of the input and the output files. This is used a special code for each input/output pin.

NET “(input/output variable name)” LOC = “(the pin you want it to assign it to)”

This should be done for the LED Matrix and the Seven Segment Displays. The Seven Segment displays which are totally four in number have to be divided into 2 parts, one for the score counter and the second for the timer.

And finally, after this has been created we created the bit file which is necessary for implementing the following program onto the FPGA.



*Results, Discussion, and Conclusion*

After the following implementation, we are able to play the snake game using the FPGA inputs and receive the correct corresponding outputs.

Overall this project majorly helped us understand well, the differences between a programming language and a hardware development language. It helped us understand what actually is feasible using hardware and how actually big programs such as games and under software actually work on a hardware-based level. The project did inculcate interest about how embedded systems work and how we can program better hardware based programs in the future.

We also understand a great deal of how electronic parts are manufactured in order to decrease the amount of hardware a component contains and that it is a common and optimized practice which is used throughout the industry. We understood the idea of hardware optimization and were able to apply it our program to an extent but most importantly learned a great deal about it.

*Appendix*

**Anode –** the positive end of the diode.

**Behavioral logic** – the code which involves conditional and logic based statements instead of direct hardware coding

**Cathode –** the negative end of the diode

**Forward Bias –** the condition of the diode where current flows through it, least resistance is offered through it.

**Mapping file** – the .ucf file which needs to be created in order to map the input and output pins

**Registers** – the combination of multiple flip-flops

**Reverse Bias –** the condition of the diode where almost zero current flows through it, it offers high resistance.

*Acknowledgment*

We would like to thank our Digital Systems professor Prof Joyce Mekie, for giving us full support with respect to this project and helping us through every stage of our project with our queries and doubts.

We would also like to thank our Teaching Assistant Mili Lavania for being a constant guide for us in this project and also being a direct link between us and our professor and for solving our queries whenever it wasn’t feasible to do so with our professor.

We also would like to thank the Electrical Lab Officials Palak Bagayi and Tushar Brahmbhatt for helping us with providing Lab equipment whenever asked for, without any hesitation so that our project could run smoothly.

And to all our friends who have helped during this process, even though they had their own projects to look after and to the other Teaching Assistants who have helped us in other small things in the process we all in the team would like to extend huge thanks.

*References*