**Project Name: FPGsnAke**

**Student Name: Michail Ouroutzoglou**

**Student Code: 2014030063**

**Introduction**

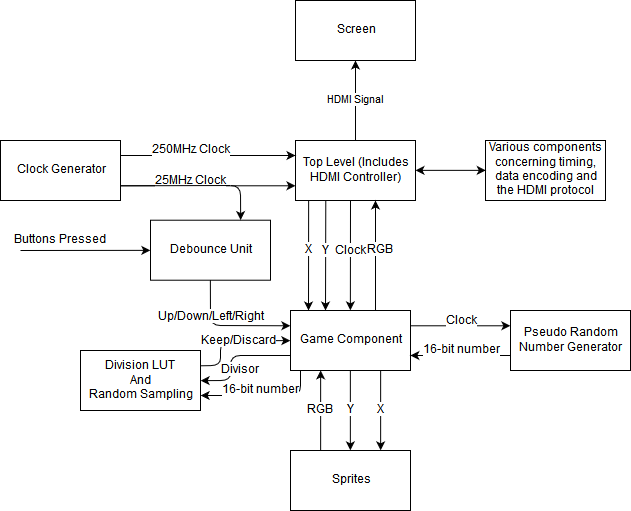
FPGsnAke is a snake game for the ATLYS FPGA that was developed as a project in the context of the subject “Embedded Systems” throughout the winter term of year of 2017.

The game is won by moving the snake towards the apples and accumulating points until the entire grid is full. Game over is reached when the snake either bites itself or collides with the brick wall.

The purpose of this project was to attain knowledge on the FPGA technology by recreating the snake game found in the older cell phones.

In order to implement the game, we had to teach ourselves from scratch the hardware description language named Verilog (so as to be compatible with the HDMI drivers). The code for the most part was written in the editor named VIM, and the platform used to simulate and create the binary files was ISE 13.4. Version Control (git) was also used (the code of this project is available [online](https://github.com/mouroutzoglou/FPGsnAke)). Finally, some external scripts written in python were used to simplify some processes. So, despite this project being focused on a game developed for an FPGA platform, much more diverse knowledge was acquired throughout the above process.

**Block Diagram**



The above is a block diagram that denotes all the major components of the project, as well as the basic signals that enable their communication.

The analysis of this project will now be broken down to the analysis of each individual component.

**Clock Generator**

This module is responsible for generating the needed clocks for the functionality of the rest of the circuit.

Depending on the video mode of our HDMI controller it can provide us with the correct clock to operate the HDMI driver.

Also, it is responsible for counting the horizontal ticks and vertical ticks to generate the HSYNC and VSYNC signals used in the HDMI driver.

This is one of the most fundamental components of the circuit.

It should be mentioned that the 25MHz clock is used to count 25 million cycles and activate various function once per second.

**HDMI Drivers**

Since the only available video output of Spartan-6 is an HDMI port, then we needed code to drive our data through that output.

The basic idea of the HDMI protocol will be shortly analyzed here.

Given input RGB (24 bits, 8 bits each color), VSYNC and HSYNC and some control signals, we encode our data through a TMDS encoder that outputs a 30-bit output (10 bits for each color). These bits are stored in 3 registers, from where we pop one bit from each register at a time and drive it to the HDMI port. It should be noted that the HDMI port uses differential encoding, meaning that for each signal there are two actual wires, one transmits the signal as it is, and the one negating it before transmitting it.

According to the above, one can realize that a single clock is not enough. As a matter of fact, a second clock, which is 10 times faster than the other, is utilized to drive the differential output, which must output 30 bits through 3 signals during one cycle of the slow clock. It should be noted that, the HDMI protocol demands that the fast clock is also transmitted through the differential output.

In this project the video screen mode used was VGA meaning 640x480 @ 60 Hz, this means that a clock @ 25 MHz (as stated in the Xilinx documentation found below) was needed to drive the pixels, and a fast clock @ 250 MHz to drive the actual output.

Xilinx provides documentation for HDMI Drivers [here](https://www.xilinx.com/support/documentation/application_notes/xapp495_S6TMDS_Video_Interface.pdf). In the Page 14 of the PDF linked, there is a link that can be used to download the HDMI drivers that Xilinx has already prepared for use. One can start playing with the code by opening the vtc\_demo.v file (in our project this file is named top\_level.v). You can change the output by writing a component that takes as input a (X, Y) coordinate and output an RGB value, and replacing the hdclrbar.v module instantiated in the vtc\_demo.v (this is what we did, by checking the comments one can see exactly what to do to achive the same results). The drivers offered are really good and are able to run the demo, straight out of the box without any modifications. They also provide the ability to transmit at various modes, and change them at real time execution which is really handy.

**Debounce Unit**

This module is responsible for stabilizing the input from the push buttons before it reaches the game component.

The above is accomplished by having two flip-flops, one of which stores the last stable state and the other stores the current input from a push button. When the two flip-flops output differ, a counter starts counting from 0 up to a certain predefined number of cycles, that should vary according to the speed of the clock. When the counter reaches the maximum predefined number, the flip-flop that stores the steady state gets assigned with the value of the other flip-flop.

When the current input becomes the same as the stable input, either due to the fluctuation of the push button or due to the stabilization, the counter resets back to zero.

What this accomplishes is that the debounce unit waits until the input is stabilized for a number of cycles before it gets propagated to the rest of the circuit.

Another feature of the debounce unit, is that it outputs a single cycle pulse whenever the push button gets pressed (stable ‘1’) and whenever the push button gets released (stable ‘0’).

This functionality was used to ensure that snake’s head turned only once when a button is pressed.

**Sprites**

For each image we want to project on the screen we need to have an individual file that stores the RGB value for its of its pixel. For example, the apple, the snake, the bricks, the numbers for the score are all a sprite.

In our case, we use sprites of 32x32 dimension. So, in order to represent the sprite in Verilog we create a big case statement that covers each combination for pixel input (X, Y) and assign the proper RGB color to the output wires.

Since this task can’t be performed by hand one by one, we implemented a flexible script in python that generates a Verilog file given an image and the dimensions of the output.

Thus, whenever we need to project a sprite we just give it as input a coordinate and it returns us an RGB value.

Pseudo Random Number Generator

In order to produce random number to use in our game component we used the method known as ‘Linear Feedback Shift Register’ ([LFSR](https://en.wikipedia.org/wiki/Linear-feedback_shift_register)).

The circuit is implemented by a shift register, some logic gates at the appropriate bits of the register and using the output of that logic as input to the shift register.



By choosing an appropriate logic and size of the shift register, one can make an FMS that goes through 2^K states (where K is the number of bits) before looping back to the initial state.

These 2^K states are pseudo random, meaning that if we know beforehand the state that we are in then we can calculate the next state.

But, if the time in which we are looking at the state is random, we can get more randomness out of our RNG.

Since this circuit is cheap and easy to make (FPGA cost-wise) we can get a big shift register so as to not have problems with the clock period synchronizing with the LFSR loop cycle giving us less random numbers.

**Division LUT and Random Sampling**

This component is used to perform the random sampling algorithm.

This algorithm takes as input a serial stream of items, without storing them, and keeps only one of them randomly and equiprobably.

This is performed by assigning a probability to keep each item equal to 1/i, where i = 1, 2, 3, …, N.

From there on, it can be proven that at the end of the algorithm the probability to choose an item is 1/N for every item.

Since we cannot generate random float numbers in the range of [0, 1] to carry out the random experiments we use the integers produced in the RNG module which are in the range of 0, 1, …, 2^16 – 1.

The Division LUT is a look up table that given a divisor M in the range of 1 to 169 (we check at most 169 items) returns the integer part of the division of (2^16-1) / M.

If the random number is smaller than the above division then we count as a success in the experiment.

Thus, we have successfully implemented the algorithm of random sampling in hardware level.