Michail Ouroutzoglou

2014030063

Report on the second milestone.

In this milestone, I implemented:

* Input handling.
* Snake movement.
* Snake growth\Score keeping.
* Drawing the snake.
* Collision detection with the brick walls.

We are going to analyze each of these one at a time, starting with:

* Input handling.

The input is taken from the four push buttons placed on the bottom left corner of the FPGA. The signal is then taken through a debounce unit, where the signal’s glitches are removed

(by glitches we mean the fluctuation of the signal until it stabilizes). Then, we forward the debounced signal to the game component where it is handled for the snaked movement.

The debounce unit works by remembering the previous state of the input signal, and starts counting cycles if it changes. So, for example when it changes from “0” to “1”, it counts 2^k cycles, and if it’s stable to “1” throughout all these 2^k cycles it then the output of the unit becomes “1”, otherwise it remains “0”. There is also, extra logic in the debounce unit, that outputs “1” for one cycle right after the 2^k cycles, and also outputs “1” for one cycle after 2^k from when the button is released. This is really useful for our case, since we want to get the input for one cycle, that is to prevent the snake from turning around uncontrollably.

* Snake movement.

After getting the debounced inputs (four of them: up, down, left, right), we use them in a simple FSM, where depending where the snake is looking, the input is either valid or invalid, and changes direction accordingly. What that means is that, if it is looking up or down, we can’t accept as input the buttons up or down, the same applies for left and right. That’s how we control the movement of the head. The rest of the body must follow the head’s movement. The way we solved this problem was by having a register for each part of the body and each time the head changes, the values are propagated from top to bottom.

* Snake growth\Score keeping.

We set up a counter that increases its value every time the snake is about to move in the position where an apple is located. This is the score, but it also is the length of the snake’s body. Since we already store the position for every part of the snake, we just need to use this counter as a way to know up to which depth should we search for when drawing the snake (or when it bit itself, but that’s in the next milestone).

* Drawing the snake.

In every frame, the game component is given (X, Y) coordinates and is asked by the HDMI component to return an RGB triplet. Thus, the game component must first determine what is to be drawn on that (X, Y) pixel. The way we solved it is by checking each and every part of the snake (up to the depth that the score counter limits us) and if any part of the snake is on that pixel then we return the appropriate RGB value.

* Collision detection with the brick walls.

This is simply solved by checking if the next position of the head is on the borders of the grid. If it is, then, for the time being, the snake just loops around the grid. Later, we can show a game over screen when the collision is detected.

Some thoughts on the choices we have made for the above design:

It is a fact the way we store the snake locations is not optimal both in terms of size and in terms of speed (when it comes to determining where to draw the snake). A better approach would have been to have a “map” and store “1” wherever the snake is located and “0” otherwise. Nevertheless, this simple approach, is enabling us to implement other parts (milestone 3) of the project with more ease, so we decided to stick with it.

The end.