HW 5

Objective

You are going to extend/modify your previous snake game to implement ***levels*** and ***obstacles***. In particular you will learn to implement and utilize ***case-statement-based state-machines*** and learn to ***serialize calculations***.

Due Dates

- Due Friday Nov 10: top-level system diagram and top-level game management state-machine description. (5 pts)
- Due Tuesday Nov 14: Project Submission.
- Due Wed Nov 15: Report Submission (only updates to the report folder are allowed).

Description and Requirements

- The obstacles should be drawn in magenta, but otherwise are like the fence â€“ the game should stop once the snake (head) overlays an obstacle.
- The game field should be initiated with no obstacles in the field of play for Level 0, and proceed just as in HW4 until the 5th apple is eaten.
- Every time the player eats 5 apples within a level, a new level should be generated with 10 more obstacles than the previous level and play should restart on that level.
  - Level 0 has no obstacles, Level 1 has ten obstacles, Level 2 has twenty obstacles, and so on...
  - The snake size is reinitialized to 1 to start each level
- The level score, and the level should be displayed on the LCD display. The display should be as follows:

  ```
  Lyy
  ```

  with (yy) denoting the current level.
- When the game ends, append an *E* as follows:

  ```
  LyyE
  ```

Design and Implementation Requirements

- In this assignment you are required to create a simple ***case-statement-based state machine*** to control and manage levels of the game.
- You need to modify your last HW as needed to accept control signals and output status signals to interface with your state machine (This means some of you need to finish HW4 in order to complete HW5.)
- You need to be able to restart the entire game (reset the ball and go back to A0L00) when RESET is pressed.
- In this assignment you are required to create a simple ***case-statement-based state machine*** within a self-contained module that internally generates, stores, and manages obstacles, and it also computes collisions given the appropriate input information about the snake.
  - Unlike the previous HW for which you may have checked for snake-head to body collisions with *parallel* hardware, you should instead *serially* iterate over the obstacles checking one obstacle for collision in a given clock cycle. There are enough spare clock cycles between the game's frame updates to support serializing this. Overall serialization could require less hardware. Try to support as many obstacles as you can and include a discussion in your report.

Additional Notes

I will be providing the following module for you to write to the LCD display.

```
module LCDDriver ( output reg [3:0] sf_d_11_8, //for LCD, see table 5-1 page 42 in user guide
                   output reg lcd_e,          //for LCD, see table 5-1 page 42 in user guide
```

```
            output reg lcd_rs,          //for LCD, see table 5-1 page 42 in user guide
            output reg lcd_rw,          //for LCD, see table 5-1 page 42 in user guide
            output reg ready,           //high when module is ready for write signal
            input wire start_write,     //pulse 1 clk period to begin LCD write
            input wire [6:0] dis_pos,   //used during write strobe, fig 5-3 pg 43 in UG
            input wire [7:0] dis_char,  //used during write strobe, fig 5-4 pg 43 in UG
            input wire rst,             //must set rst for one clk cycle to init. module
            input clk
);
```

Design Approach

The design approach and structure is up to you.

- I would recommend creating the obstacle management and multi-cycle serialized collision detection module first.
- Then separately build a state machine that can control the display to ensure you understand that.
- Last, you'll need to implement the top-level game management state-machine. Here is one terse and likely incomplete outline of states that you may follow:
  - Initial State - send signals to modules to initialize
  - Wait for LCDDriver ready (move one when LCD is ready)
  - Send 'L' to the LCDDriver (move on when LCD is ready again)
  - Get ready for Level
  - Send level digits (finish each when LCD is ready again)
  - Initialize for level Signal any other modules, like the obstacles module to initialize obstacles and wait for ready
  - Send signals to Start Level
  - Wait for the outcome from the level
  - Update internal score and update display if required (wait for LCDDriver to finish) and the go back to starting a new level with updated number of obstacles
  - Finish Game (display an X)

What to be sure to turn in

- Submit the whole CLEANED ISE project folder(source files used to generate it) and instead of submitting only Verilog(.v) files (YOUR COMMENTING OF CODE WILL BE GRADED).
- Submit bit files in a separate directory
- Create and hand in one multiple Verilog testbench modules that test your design
- Create a report that briefly explains your design and your testing You must have one testbench for each module.
- Include the output of your Verilog testbench(s) in your report (THIS IS EXPLICITLY GRADED) with additional explanation about each testbench as needed to convince someone that each part of your design works and your simulation-based testing of each module is sufficient.

Provided Files

- https://eclipse.umbc.edu/robucci/cmpe415/attachments/LCDDriver.v

-