CSE 371A
Autumn 2019

Lab 5: Digital Signal Processing

Phuc To, Darren Lu

CSE 371A
Autumn 2019

Introduction

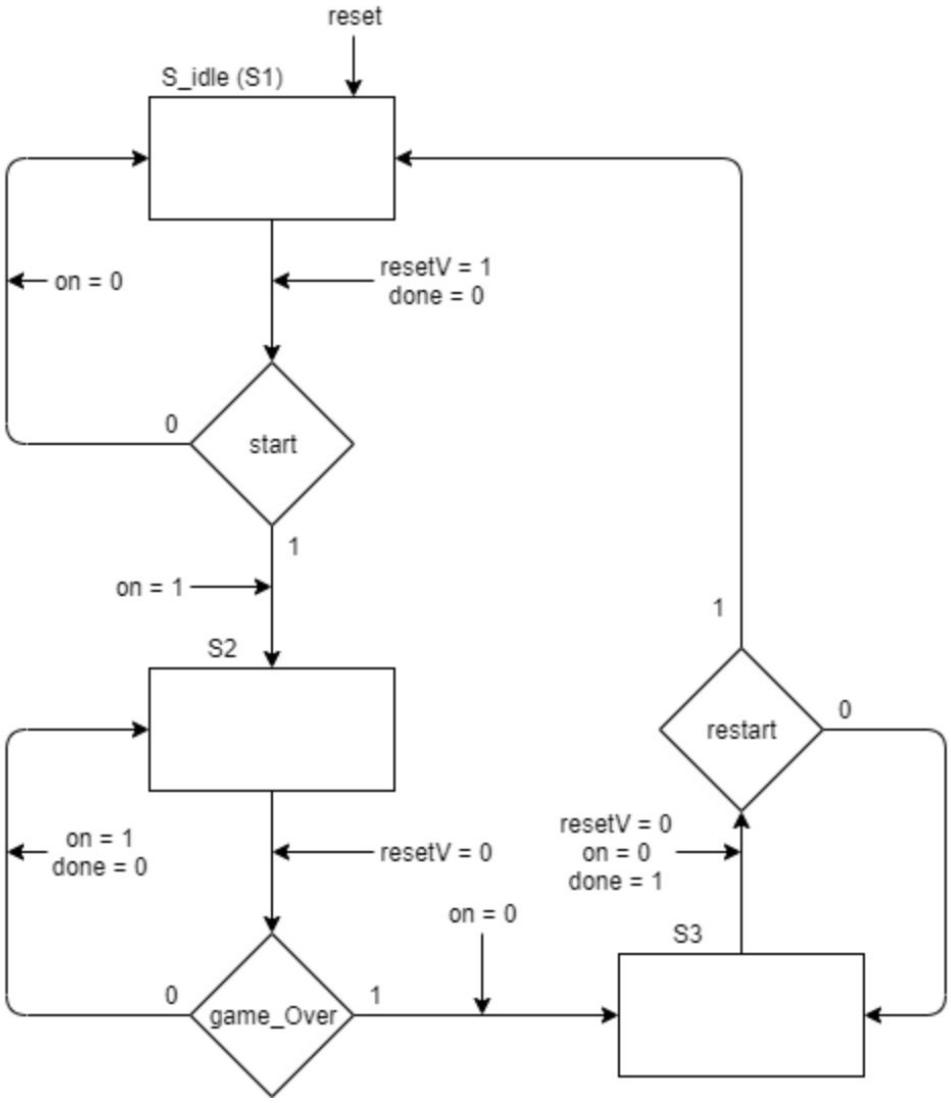
This is the culmination of EE 371 and SystemVerilog. We are challenged with building from a number of possible tasks. We chose to do a sidescroller call “Racing” in which a player controls the left and right position of a fast moving vehicle. Cars will be flowing down from the top of the screen and act as obstacles that the player needs to dodge. Although it might not seem that complex, many of the game logics needed to be thought of carefully.

Design Specifications

Our game works with the player utilizing the left and right arrow keys on the keyboard to move the player tank unit left or right. Tank obstacles will be generated randomly from the top and the player needs to dodge. All the animations are shown via the VGA to a monitor. Additionally, music is being played throughout the game to give a more grand feeling, and the player use “Enter” key to start, “Space” key to restart once they lose.

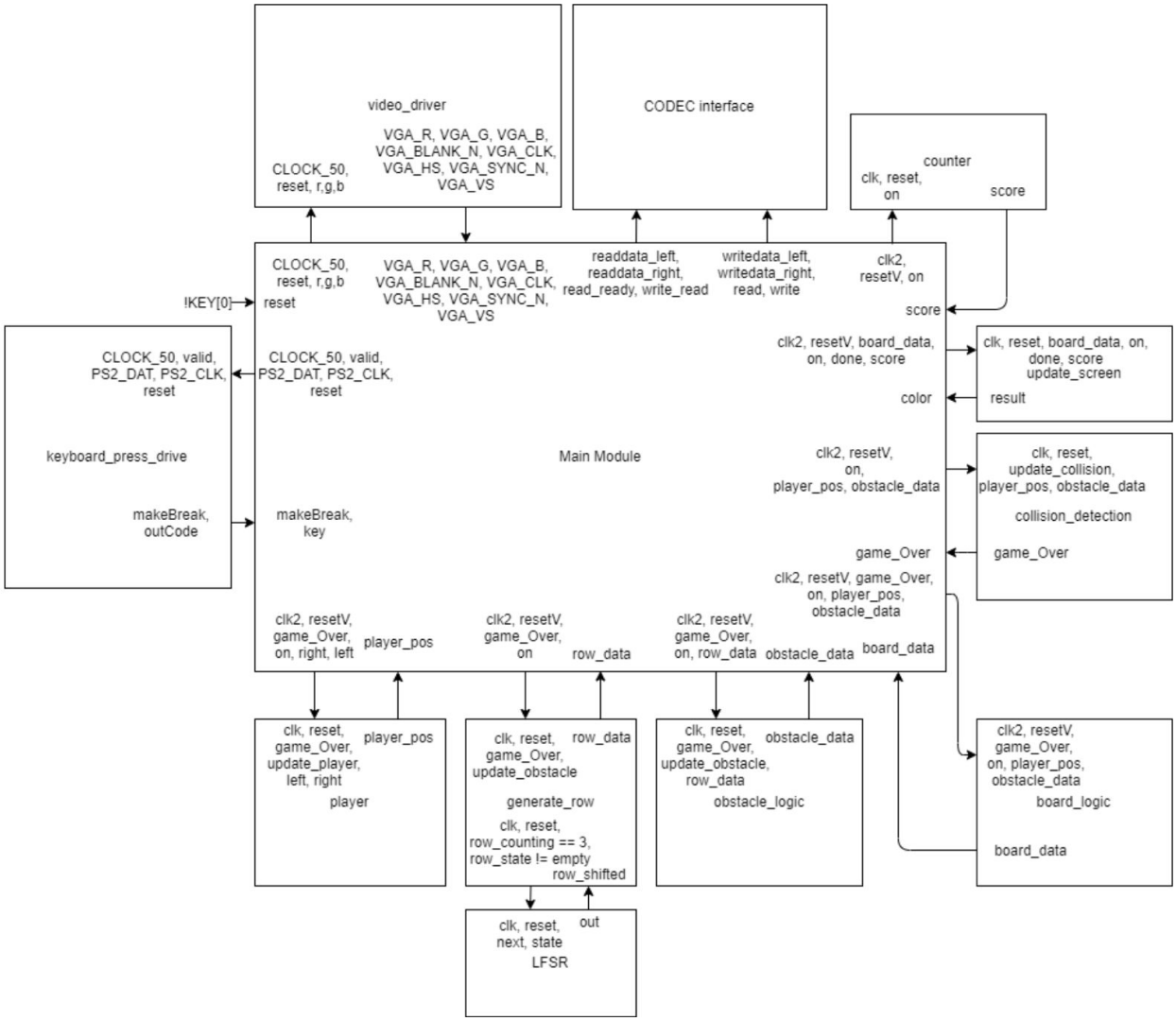
Procedure

ASMD Chart



CSE 371A
Autumn 2019

Block Diagram



CSE 371A
Autumn 2019

Modules Explanation:

There are two main parts of this project: Logic and System.

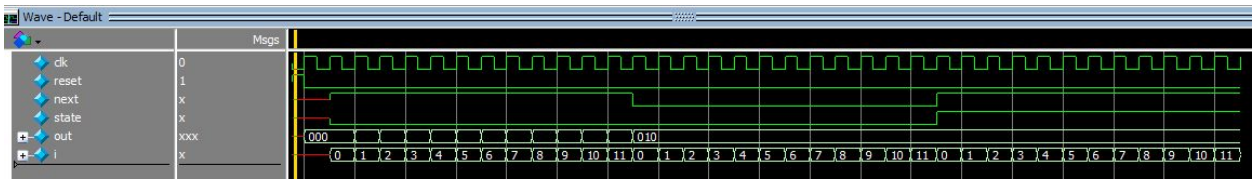
1. Logic:
    - LFSR: generates a pseudo-random 3 bit number.
    - player: outputs the player location based on input from the keyboard. If left is high, will move player one to the left and if right is high, will move player one to the right. Outputs it in decimal format. The rightmost column is column 0.
    - generate_row: generates a new top row for the entire board. Takes input from the LFSR to know which column to generate obstacle from. It can either generate obstacle rows for empty rows for the player to be able to move through.
    - obstacle_logic: controls the entire board for the obstacles. WIll shift down all obstacle blocks when requested to do so: this includes bringing down the newly generated row.
    - collision_detection: gets all the obstacle locations from obstacle_logic and the player location from player. Detects if blocks are overlapping and will output whether the game is over.
    - board_logic: grabs all the obstacle locations and the player location and will mash all the block locations together. The 2d output vector will have all the blocks that need to be shown by the VGA.
    - counter: counts the score the player gets.

2. System:
    - DE1_SoC: connects all the logic ports to the input/output ports of the FPGA.
    - clock_divider: given module. Used to create slower clock for the players.
    - video_drive: given module. Receives color data from the logic and output data to control the screen.
    - CODEC controller modules: given modules to receive and output the sound signal.
    - Keyboard modules: given modules to receive data from the keyboard and output the pressed button to the system.
    - update_screen: receives the game data from the logic modules, and outputs the color data for video drive module.

Result

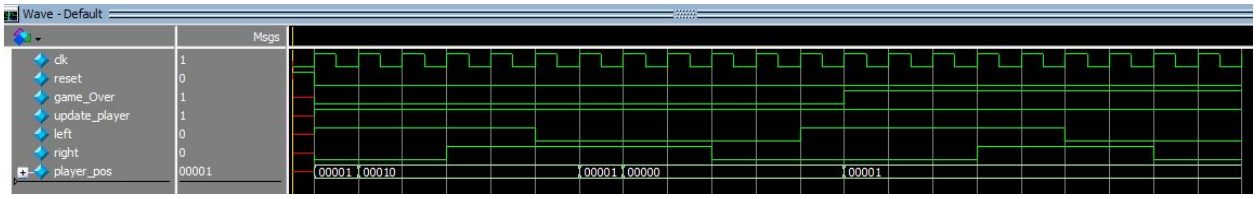
ModelSim Simulation

1. LFSR:



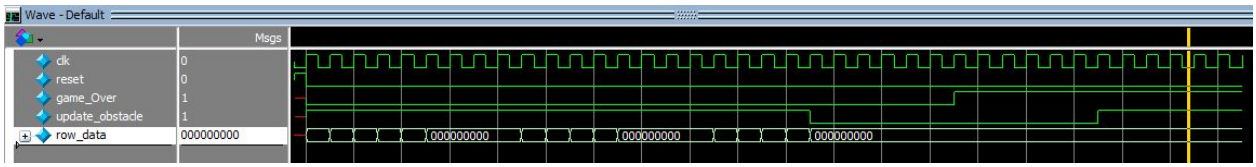
The testbench shows that the module works properly by generating a 3 bit pseudo-random number whenever next is high and state is low.

2. player:



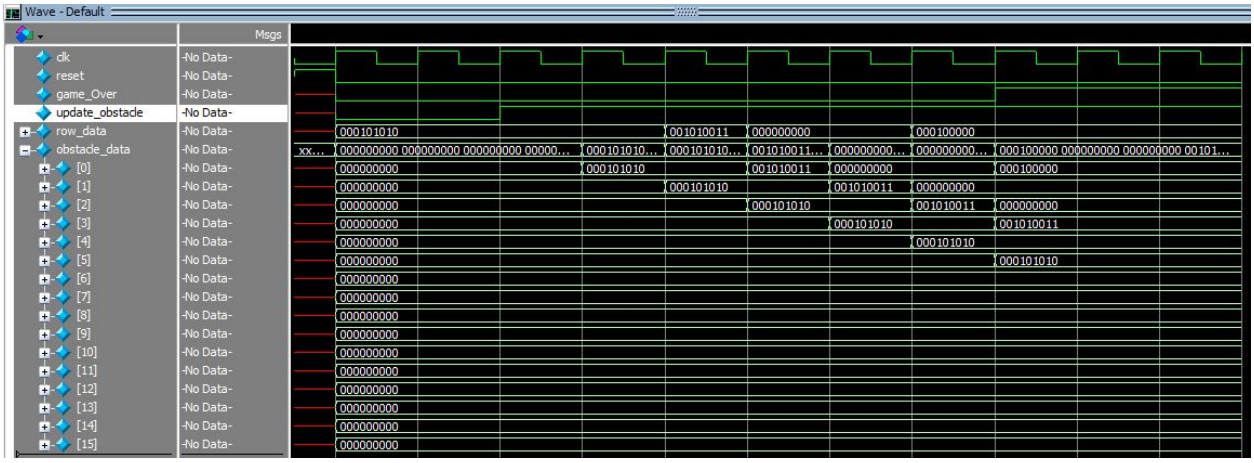
This testbench shows that as the player inputs a left signal, the onscreen player moves left. Similarly, if the player inputs a right signal, the onscreen player shifts right. We can tell it is working properly because it only works on the game_Over signal is low and that it stays within our given player bounds.

3. generate_row:



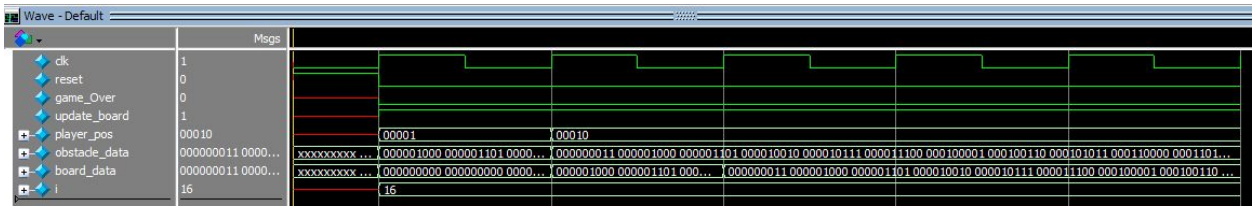
Generate_row generates the top row for the obstacle_logic. It functions correctly as we can see that it outputs obstacles for 4 cycles and then 4 empty rows. Furthermore, it stops outputting new rows when the game is over or if it does not need to be updated.

4. obstacle_logic:



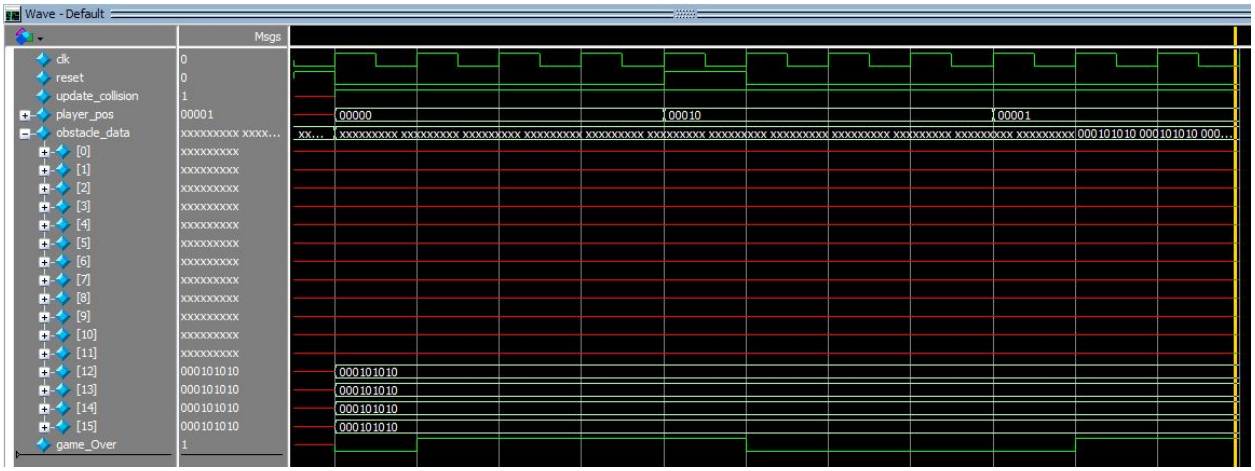
The testbench shows the obstacle_logic module shifting the obstacles down the screen. It works because it only shifts when the update is high and the game_Over is low.

5. board_logic



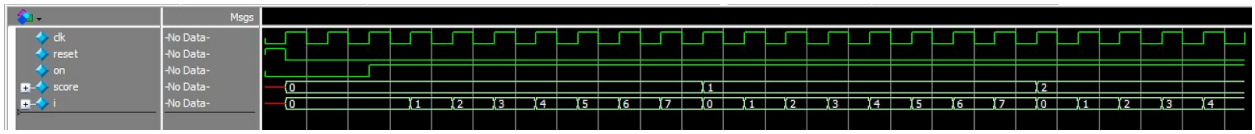
This module mashes the player and obstacle locations together. Here we can see that the module works because it the board_data only changes once when the play location changes. We also checked the data for both arrays to make sure that they are being mashed together. There are simple too many signals for them to be shown here.

6. collision_detection



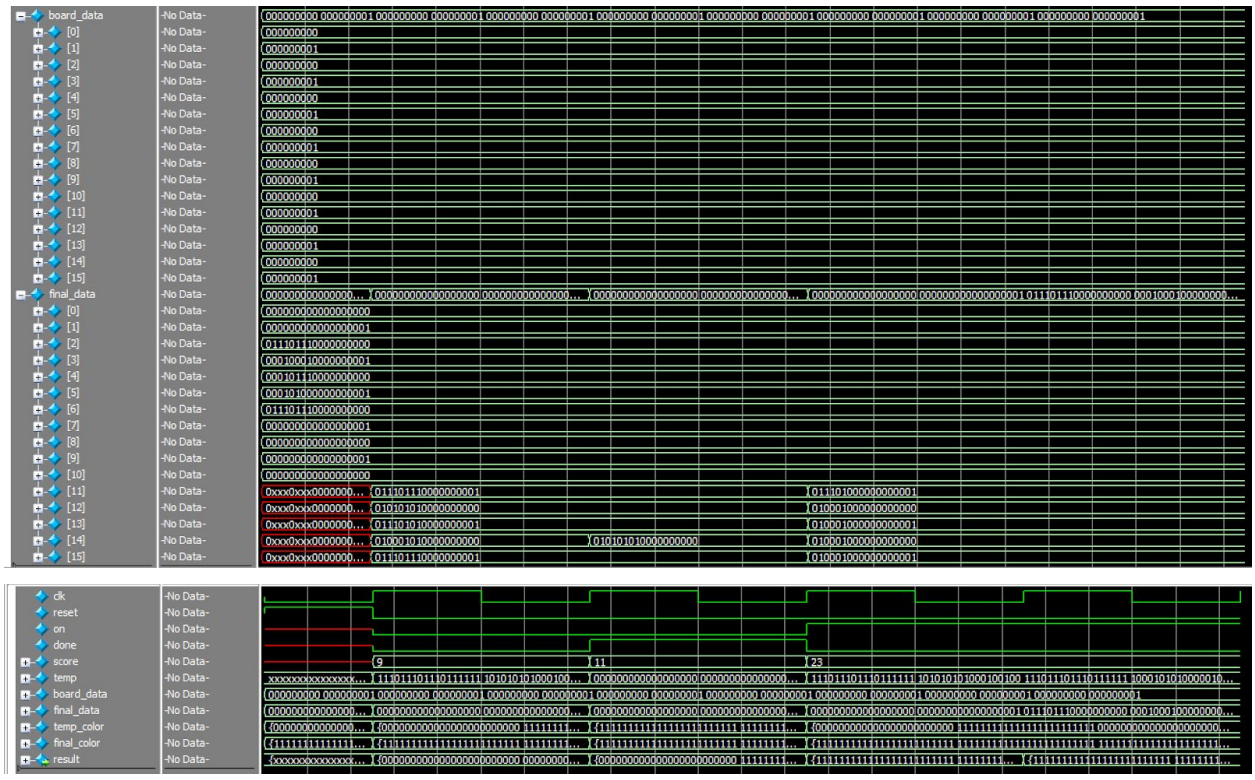
This testbench shows the correct functionality of Collision_detection as it sends the the game_Over signal only when the player location overlaps with the obstacles.

7. counter:



The testbench shows that the module reset the score when it is reset and when the game is over, otherwise it increases the player's score every 2 obstacle blocks (8 cycles).

8. update_screen:



This testbench shows that the module creates the correct data for the active screen according to the input, and outputs accurate data depending on the game state (on and done signal).

CSE 371A
Autumn 2019

Project Overview

This project was long and grueling. We came out of it smarter and enjoyed the experience. There was a lot to think about for this project as we needed to worry about the overall game flow and the game logic behind it. We also needed to make sure all of our modules would update at the correct time. We encountered some problems like the collision detection but was able to work through most of them. Ultimately we were able to create a fun and beautiful old school side scroller game. “Racing” was fun to build and a great game for everyone.

CSE 371A
Autumn 2019

Problem

There are four problems, one is about the given hardware, one is not solvable, but does not affect the program, two are solved:

a) collision detection: Finding out when the player collides with the obstacle is relatively straightforward as we just need to check whether they are occupying the same space. However, the difficult part came when we needed to fill out which locations to check. Quartus gave us problems when we wanted to parametrize much of the player and obstacle checks and was saying that the player location is variable and that we cannot use it to traverse a vector. We tried multiple methods such as using temp variables and even changing it to check on the clock edge. Ultimately none of it was successful and we resorted to doing a case statement. This is not every parameterizable as we need to make statements for each new case. The solution still works in our case as the number of player positions is limited, we can code all three cases.

b) keyboard: As keyboard was a new type of peripheral we use, there were some difficulties in beginning with it. However, after getting used to it, it is not that hard after all.

c) VGA driver: The way the new VGA driver works is totally different from the one we were given on lab 3. In the beginning, we could not get it to work as we thought the coordinates were our input, but in fact it was the output from the module. We resolved it with the help of the TAs.

Time spent

1. Reading: 30 minutes.
2. Planning and designing: 2 hour.
3. Coding and Debugging: 20 hour in total. The logic for the game was not so simple, and the addition of the peripheral (CODED, VGA and keyboard) made the main module complicated and hard to debug, especially when we cannot do testbenches for the system in a whole.
4. Testing: 2 hours. There is no way we can use the VGA at home that we know of, so we had to test it in the lab.
5. Writing Report: 2 hours. The block diagram was complicated as it includes a lot of modules.

Total time spent on Lab 4: Approximately 27 hours for both persons.