Charles Hung
May 18, 2006
6.111 – Digital Systems Lab
Final Project

This Snake is Down Right Fierce

Abstract

The purpose of this project is to implement a version of the classic game Snake on the FPGA. The player controls the snake's movement and the object of the game is to make the snake grow by eating apples that appear at random locations on the screen. The game ends whenever the snake collides with itself or the walls of the level. When the snake reaches its maximum length, the game will go on to the next level; there will be a total of three levels. There are three major design components to the project: a system to display graphics, a system that handles game logic, and a system to handle commands from the user. The display component is handled by using a VGA interface in conjunction with sprites and dual ported block ROM. The game logic is encapsulated in a major/minor FSM. The game state is handled by the major FSM while collision detection is handled by the minor FSM. The player will be able to send commands to the snake through a PS/2 keyboard. These commands are directly sent to a module that updates the position of the snake.

Table of Contents

List of Figures

I. Overview

The game starts by drawing the snake and apple in a default position. The snake starts moving when the player hits any direction on the keyboard. If the head of the snake collides with either an obstacle in the level or any part of itself, the game will stop, indicating the player has lost. Also, if the player inputs a direction opposite from the snake's current direction, the game will end. If the snake moves over the apple, the snake will grow in length and a new apple will be drawn to a random position on the screen. When the snake reaches its maximum length, the game will go on to the next level (Figure 1) and redraw the snake and apple in their default positions.

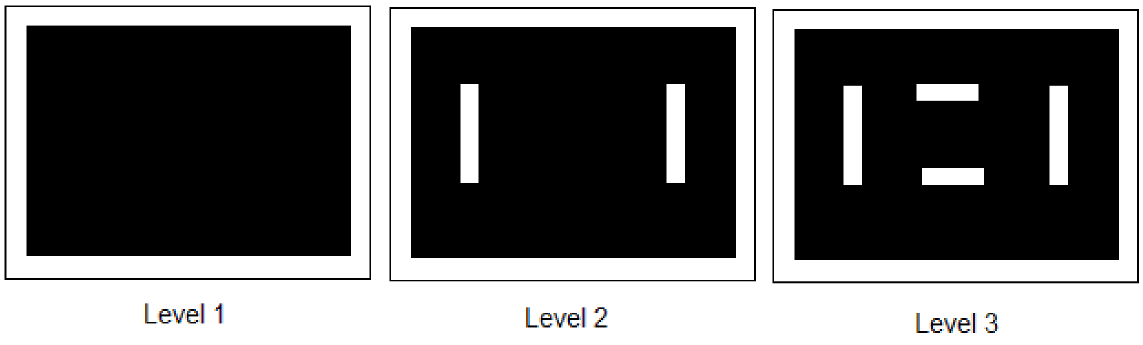

Figure 1. Picture of the levels

Figure 2 is a block diagram of the overall system. The flow of data starts with the keyboard input from the player. The keyboard clock and data are sent to a PS/2 module that outputs a hexadecimal number corresponding to the ASCII letter pressed. This number is sent to a user command module that outputs a 4 bit movement vector representing the direction. The snake register module takes the movement vector and determines the next position of the snake, which will be drawn when the pixel and line count return to 0. The new position is sent to the collision FSM, which will check for collision with the obstacles, snake, and apple. In addition, it will also latch a new position for the apple if the snake has eaten it. The collision FSM outputs a win or lose signal to the game FSM. If a win signal is asserted, the game FSM will enable the appropriate block ROM corresponding to the next level. If a lose signal is asserted, the game FSM goes into a game over state and disables further input from the keyboard. Otherwise, the snake is still alive and the game continues.

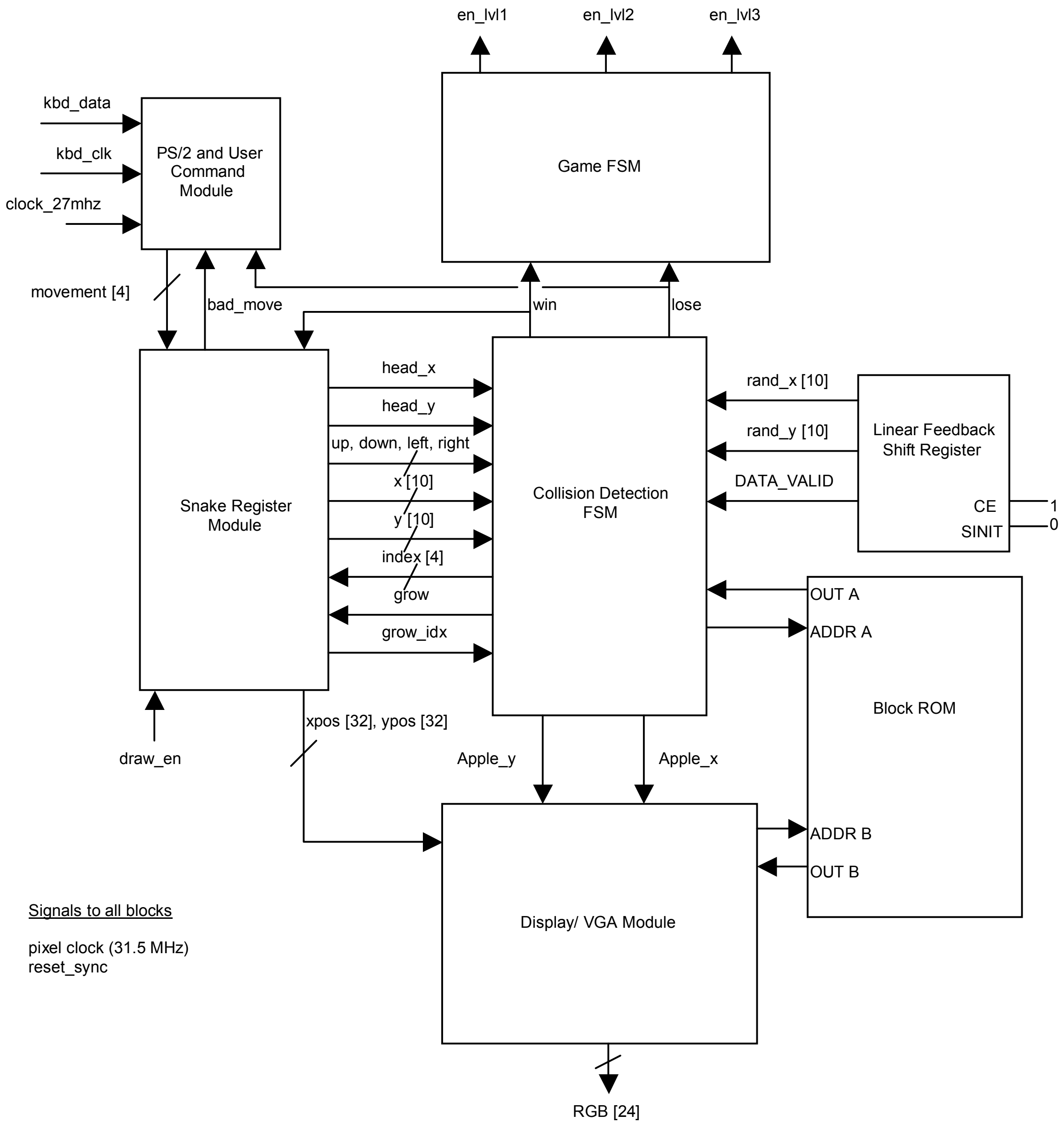

Figure 2. Overall system block diagram

II. Description

PS/2 and User Command Module

The PS/2 module takes the keyboard clock and keyboard data and determines which key was pressed. Each key has a corresponding hexadecimal code that the keyboard sends to the module. This code is then decoded to its ASCII value and sent to the user command module. The user command module turns the ASCII value into a 4 bit vector representing the direction: 1000 for up, 0100 for down, 0010 for left, and 0001 for right. If no key has been pressed yet, it outputs 0000. The output of the user command module is also regulated by the lose signal from the collision FSM. If the lose signal is high, further movement should be prohibited. We can achieve this by just having the user command module output 0000 when the lose signal is asserted.

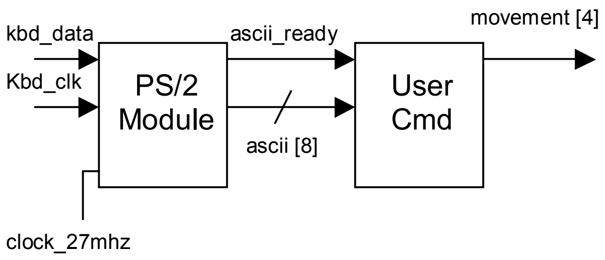

Figure 3. PS/2, user command modules

Snake Register Module

The snake is drawn in 32 square segments (see display module) so the snake register module has 64 registers, one register each for the x and y position of the segment. On a reset, the snake is drawn in its default position: the first 27 segments (the snake head) are drawn over each other (they have the same x and y position) and the remaining 5 are drawn next to each other. A grow index register stores the index of the register that is at the bottom of the stack of segments representing the head. The snake's position is updated when the current frame has finished being drawn (when the pixel count is 639 and the line count is 479). The snake can either move or grow. If the snake is simply moving, all the registers with an index less than or equal to the grow index are updated in the direction that this module gets from the user command module. The remaining registers are shifted down (eg. R28 takes on value of R27). If the snake is growing, it has collided with the apple. In that case we increment the stack of registers that will represent the new head (index 0 to grow index - 1) and keep the rest of the registers unchanged. The snake grows by incrementing the difference between positions stored in adjacent registers. This is done in an orderly fashion each time, starting at the register with grow index. Thus, at the end the grow index is decreased by one.

Collision Detection FSM

The collision detection FSM checks for collisions with the level obstacles, snake, and apple. To check for collisions with the level obstacles, it first takes the position of the head of the snake and converts it to the corresponding address in the ROM. It gets the RGB value of that pixel from the ROM. If the RGB value is 111 (white), then the snake has hit an obstacle and we go into the game over state. Otherwise, there is no collision with any obstacle. Next, it checks for collisions with the snake. It takes the position of the head and checks to see if the head touches any segment of the snake, starting from the tail of the snake. If so, the game ends. Otherwise, it proceeds to check for a collision with the apple. If the snake has collided with the apple, a grow signal is asserted for the snake register module. The collision detection FSM then finds a new position for the apple. It does this by taking a random value from a linear feedback shift register and checking to see if each corner of the apple collides with any obstacle. If not, then it checks to see if it collides with any part of the snake. If the new apple position fails any of these tests, a new random value is generated. If it passes, the value is latched as the current apple position.

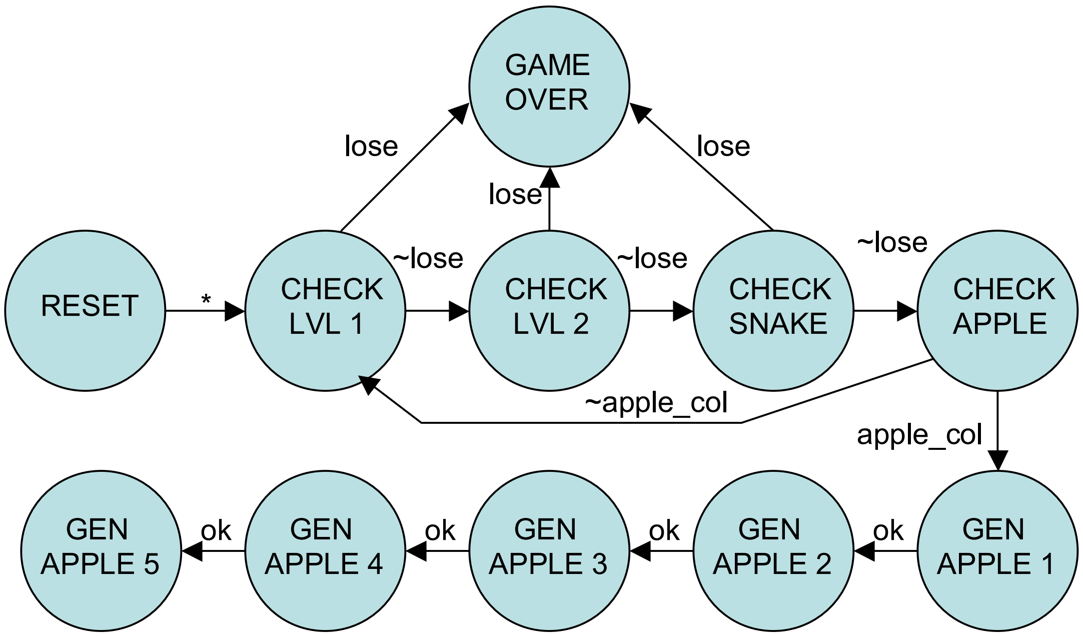

Figure 4. Collision detection state transition diagram

Game FSM

The game FSM keeps track of what level the player is on and whether or not the game has ended. It receives a win and lose signal from the collision detection FSM. If the win signal is asserted, it will enable the appropriate block ROM for the next level. If a lose signal is asserted, it goes into a game over state. Otherwise, the FSM remains in the current level state.

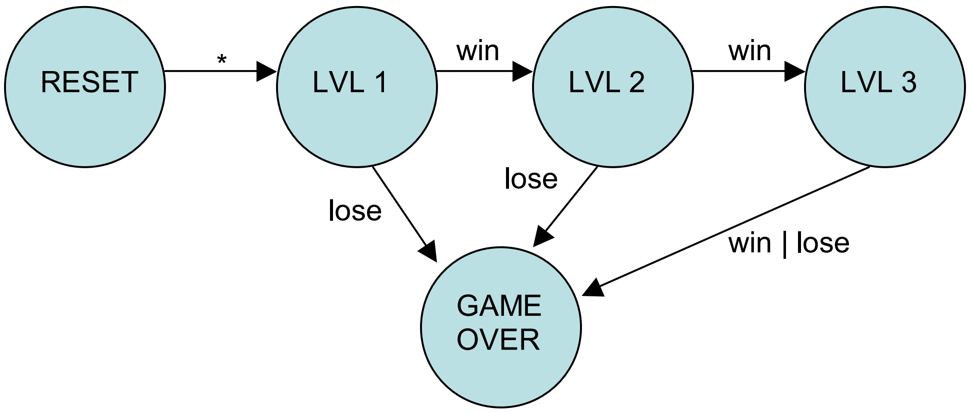

Figure 5. Game state transition diagram

Level Block ROMs

There are three dual ported block ROMs, one for each level. Each block ROM has 640 * 480 addresses for each pixel on the screen and a width of 3 bits. The 3 bits can either be the RGB value for white (111) or the RGB value for black (000). The obstacles and walls of each level are white. The ROM receives addresses from the display module to assign the right color for the background. It also receives addresses from the collision FSM to check for collision with the obstacles. The ROMs also have an enable signal so the game FSM can activate the appropriate ROM.

Display/VGA Module

The display/VGA module uses the position of the snake segments, the position of the apple, and the RGB values from the block ROM to draw each frame of the game. The block ROM contains the RGB values of background, which are either black or white. The snake and apple are drawn using sprites that output squares to the screen. To grow the snake, we increase the relative positions between adjacent sprites. Otherwise, if we are only moving, each segment takes on the value of the previous segment.

III. Testing and Debugging

Testing and debugging was done in an incremental fashion. I first coded the display/VGA module and configured the block ROMs to see if I could get the FPGA to output a static background. To instantiate the block ROMs, I used Coregen. I had some peculiar errors using Coregen. Somehow, the lengths of the addresses were being changed around and I could not figure out why. To get around this, I deleted the block ROM from the project and re-instantiated it. The next phase was to draw the snake and see if it would move and grow. I debugged this function visually by drawing different segments of the snake different colors. That way I could tell how each segment was actually being updated. I also had problems getting the snake segments to position themselves correctly next to each other. For example, by default, I always had the last five segments drawn next to each other with no overlap. I did this to give the snake some initial fixed length. However, I found that this never occurred and that the segments always overlapped. This is a problem that I am still currently fixing.

I then checked to see if the apple would output in random locations. I did this by assigning a button on the FPGA that would redraw the apple each time it was pressed. Other than minor syntax errors, I did not have too many problems with this part of the project. The majority of my bugs occurred when I connected the entire system together. Specifically, the game would end prematurely. The snake would be allowed to move for one frame and then the game would freeze. This meant the lose signal from the collision detection FSM was being asserted incorrectly. Unfortunately, I did not have enough time to examine this problem in depth. I suspect that the collision detection FSM is either sending the wrong addresses to the block ROM or it is receiving erroneous values from the block ROM.

IV. Conclusion

Although I was not able to get my system fully functioning, the final project was a still a challenging and rewarding experience. I thought that the implementation that me and Jae came up with was pretty interesting, especially the use of a ROM to store the background of each level. This made checking for collisions with the level, at the very least, conceptually easy. I also liked the idea for building the snake, because it made controlling and growing the snake relatively easy.

During the course of the project, I had to drop several ideas from my original implementation. Originally, I was going to have the player use a Logitech gamepad instead of the keyboard. However, finding the necessary calibration data to send to the gamepad was difficult. I spent a good amount of my time in lab trying to get the gamepad controller to work, and, in retrospect, I probably should've done that aspect of the project last. I could have used the FPGA buttons temporarily and then try to interface the gamepad.

I felt that the biggest problem I ran into was allocating my time on the project. I spent a lot of time trying to get each module to work perfectly before moving on to the next one. As a result, I wasn't able to get a working collision detection FSM. If I were given another chance to do this, I would do my design in a different way. I would first program a very simple game to make sure all the signals are synced properly. For example, I could make a snake that does not grow and have the game end when the snake collided with a wall. From there, I could build on the existing game and make it more complicated.