CE100 Lab 7 Write Up

Student Name: David Krieger - 1427712

Lab Section: 2 (12:00-2:00)

Date 3/11/16

Description

The purpose of this lab is to implement the game, “Loop it” using Xilinx, a Basys 2 board, and a VGA monitor. Loop it is a game where a paddle is moving forward and backward (based on user input) on a horizontal plane. This user controlled paddle attempts to ‘catch’ a ball that is bouncing off the roof and lower plane of the game boundaries. In order to catch this ball, the ball must enter the null space of the paddle. The null space is the zone in between the upper and lower square of the ‘paddle’. The user receives a point every time the ball passes through the front part of the null space. However, if the user’s paddle, top and bottom square, hit the ball, then the user loses a life (or ‘paddle). This means the user can score a point by having the ball pass forward through the null space of the paddle, and still have the ball hit the paddle’s rear parts. This will result in both a score and life loss. The user can also lose a life if the paddle moves too far forward or backward on the plane it is bounded to.

The user plays each life by hitting pb2 to start, and maneuvers using pb0. If the user loses all of his/her lives, then they must press pb1 to start a new game. If the user loses a life due to a ball collision, the ball will freeze and the paddles will flash for 4 seconds. If the player loses a life by the paddles falling off, the game will also hold for four seconds and then reset as well. Upon a final life loss, the appropriate loss animation will occur and the game will then loop indefinitely until the user resets the game. During this infinite loop the paddle is gone and the user has no input. Also note that the ball’s position is randomized at each new generated ball, regardless of the state.

Methodology

VGA display - The VGA display is managed through pixel values that are constantly being updated through a clock. Students have to get all objects of the lab to be displayed on the board, as well as have object interaction,

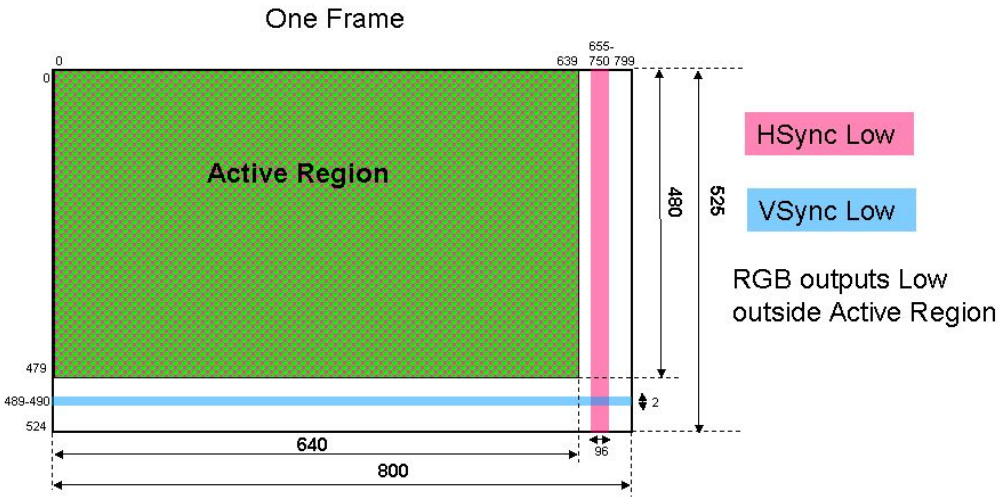

Fig. 1. Corresponding pixel rows & columns for display management

a. The State Machine - The statemachine is incharge of managing all of the output signals to the various parts of the design. It tells when the paddle should be displayed, if the ball should move, if the paddle should be moved… etc (much deeper explanation in the results).
b. The VGA controller - The students had to track each pixel position to be displayed on the board. This is done using two 8 bit counters that counted the columns from 0-799 and rows 0-524. Furthermore, in order to ensure all of the pixels move and lined up correctly, horizontal and vertical synchronizers were added (Fig 1.). The students also created an edge dector using the VS signal as its input. The VS signal went from low to high once every frame. This frame value acts as a way to move objects by any pixel amount they want PER frame. A frame is the time it takes to refresh the screen to display objects on the screen. Professor Schlag explains the time it takes to display a single frame of the game (fps) to be, "Transmitting one frame takes 800 x 525 x 40ns = 16,800,000ns = 16.8ms, so the monitor is being refreshed roughly 60 times per second: at 60Hz."
c. Null Space/Border - The null space of the screen is a blue rectangle. It all of the pixels from rows 0-479 and columns 0-639 subtracted by the game region. This space should have no animation on it and exists as a pseudo border to the game.
d. Game Region - The game region is defined a rectangle defined as the rows from 256 - 471 and columns 7-631. This provides an 8 pixel gap from the left, right, and bottom parts of the monitor, and a 256 gap to the top.
e. Carpet - The carpet region exists from rows 400 to 415 and columns 55 to 583. The carpet will also move right to left when the user is actively playing the game (moving the paddle back and forth). This is managed by hardcoding blocks of 32 to fill up the carpet space. These blocks are then split into two different colored blocks of 16. These blocks 'move' by subtracting one pixel per frame, and updating the result on the next frame of the display board.
f. Paddle - The paddle's location value was saved using two updown counters, one for the vertical, and one for the horizontal. The paddle rests on top of the carpet and exists from rows 383 - 399 and 335 - 351, while the column was varying based on its position (but always 16 pixels wide). The row values causes the paddle to have two squares of 16 pixels tall, and have a null space of 32 pixels in between. The paddle was moved using the frame counters from the VS edge detector. The horizontal movement of 1 frame per second was controlled using the frame output from the rising edge detector. The vertical dropping of the paddle was managed by the falling edge detector of the VS signal. By using both the rising and falling edge output of this detector, we can use it to move the paddle at 2 frames per cycle (falling edge also occurs once per frame cycle).

g. Ball - The ball's location value was tracked using two updown counters as well. The ball's initial position was randomized and moves left (columns decreasing) at one pixel per frame. The horizontal position moved either up or down (row values) based on its collision between the roof of the game boundary (256) or the top of the carpet boundary (399). The ball's position is reset upon a new life or upon hitting the left end region of the game border. This reset position is a partially random row value and a fixed column value (game region border - 8, because the ball is 8 pixels wide).
    i. The random number is generated from an 8 bit counter being tied high and only taking 6 of the bits. This means a random number from 0 to 64 is taken in by the initial ball load position. This number is then added to its starting location of row 256 to generate a random row value from 256 - 320.

Results

This lab offered a lot of freedom in how to implement all of the required functions for this lab. Here I explain the process I took to accomplish the implementation of the game.

a. State Machine - My machine consists of 7 inputs and 10 outputs. I will first explain each state of the machine.
    i. Start game - Occurs as the first state of the game, and when pb1 is pressed after losing all of your paddle lives. Nothing is moving and will stay in this state until pb2 is pushed and will transition to the play state. No other inputs have an impact (besides pb3 global reset).
    ii. Play - Here the user is moving their paddle back and forth trying to score points using pb0. It stays here until the user either collides with the ball or has the paddle fall off of the carpet. It will transition to either paddle fall or flash. No other inputs have an impact (besides pb3 global reset).
    iii. Flash - This means a collision was detected. The paddle will flash and the ball will be frozen. This state says here for 4 seconds and will go back to the idle state if the user has more lives left. If the user is out of lives, the game will instead transition to the no paddles state. If the user still has remaining lives it will transition to the idle state.No inputs have an impact (besides pb3 global reset).
    iv. Paddlefall - The paddlefell off of the carpet. This state will remain here for four seconds until either moving on to the idle or no paddle left state (see flash state for qualifications of transition). No inputs have an impact (besides pb3 global reset).

v. Idle - After losing a life, but still having remaining ones, this state will sit idle and having nothing moving. It transitions out when pb2 is pressed and will re-enter the play state.
vi. No paddles - After the user has lost all of their lives, this state will exists after their appropriate 'loss' animation (paddlefall/flash). The paddle becomes hidden and the ball and carpet will move indefinitely until the user presses pb1 to reset the game back to idle. No other inputs have an impact (besides pb3 global reset).

b. The inputs and outputs of my state machine are as follows:
   i. Inputs - All corresponds to telling when the state machine should change
      1. pb1/2 - I/O input
      2. Foursec - Tells the system when 4 seconds have passed.
      3. Nolives - If no lives are detected then this is high.
      4. Collision - A collision is detected
      5. Paddlegone - The paddle has now fallen off the stage.
   ii. Outputs - Tells various parts of the system to do something
      1. Move carpet - Tells when the carpet should move. Sends signal to the pixels schematic.
      2. Moveball - Tells when the ball should move. Sends signal to the pixels schematic
      3. MovePaddle - Tells when the paddle is can be moved. Sends signal to the pixels schematic
      4. Resettimer - Sends to ScorenTimer schematic where it will reset a 4 bit counter for the flash and 4 sec calculation.
      5. Decrementlives - Sends to ScorenTimer when the lives should be decremented.
      6. Loadlives - Sends to ScorenTimer to load the value 3 back into the lives counter.
      7. Paddlehide - Sends to the top level when the paddle should be shown on the VGA output.
      8. Timecount - Sends to ScorenTimer when the 4bit counter should start counting (for flashing & 4 seconds).
      9. Resetposition - Tells when the ball and paddle should reset positions, connected to the load values of their counters.

c. Everything else went according to plan except for my collision detection (check schematic collision edge). I originally tried just checking to see if the ball's position ever aligned with the paddles position (and gate), but it still wasn't

working. I wound up creating an edge detector so it could find went it went from low to high (no collision to collision) and it worked. However, because this signal only went high for one clock cycle, the ball and paddle jaggedly moved through each other. I fixed this by creating a small series of gates which went high and stayed high upon edge detection. This output signal is called collision hold and is sent to my pixels schematic which holds the current position of the ball and paddle (tied to the count enable of both objects, 4 counters total).

d. My score detection worked as the same as the collision. I didn't use a state machine to control the scoring logic of my lab. Instead I had a signal that went high after the ball crossed the front column of the paddle in the null row space. I also have it anded with a not collision signal so it won't constantly increment the score if it hits and stays in the 'jackpot' pixel space of the null row and front column .
e. My FPGA output management is the exact same from the prior labs. The right two most anodes display the score which is tracked in ScorenTimer. The left most anode displays the current lives which is also in the ScorenTimer schematic. Anode 2 is tied high causing it to always be off.

Questions:

a. The max input delay is: pb0 at 11.16 ns
b. The max output delay is: Grn1 at 17.3 ns
c. The max FF input delay is: 12.55 ns
d. The frequency is: 79.68 MHz

Conclusion

This lab was actually my favorite. I enjoyed the freedom to design my lab from my own personal logic. Some parts were unorthodox, but I designed them to perfectly fix the problem I was having. Specifically I remember someone suggesting AND'ing the ball space with the paddle space for a collision, and it didn't work for me. I instead played around and created the entire collision edge detector and hold value. This change then messed with my management of the ScorenTimer schematic. I had issues trying to control the collision edge detection while also maintaining a four second output and collision reset value. I fixed this by adding more output values to my state machine so I could better control my 4 bit timer (reset timer, timer count).

At first this lab felt really overwhelming, especially the VGA pixel management. I wasn't sure how to set things up or what was even going on. Once I got the VGA monitor to turn on and started to define regions, it really took off. I slowed down towards the collision area I have

discussed earlier. Once the collision management was correct I went on to work on my statemachine. Although it took sometime to implement and fully get the state machine going, it worked perfectly well. However, as priorly said I had errors with my foursec signal and the collision signals working properly. I wound up using my state machine outputs and connecting them to various LEDs on the FPGA board so I could see when they were going high in what appropriate state. Once I could see what was wrong with some of my logic, I fixed it piece by piece until every state worked according to specification.

I would like to suggest to the professor to make lab 6 a state machine lab using the VGA monitor. This would allows students to become more comfortable with pixel management and prep them for lab 7. I really felt off in the deep end in the beginning of this lab as I was trying to figure out how to work with the columns and rows of the VGA monitor.

**For my state machine diagram, I used the one w/ the signature on it.