CSM152A Final Lab Report

Russell Jew (ID: 604477316)
Jeffrey Jiang (ID: 904255069)

Introduction

For our project, we implemented a basic version of the game Breakout or Brick Destroyer (https://en.wikipedia.org/wiki/Breakout_(video_game)). The basic idea of the game is that the user controls a paddle and there is a ball constantly on the move, colliding with objects and changing directions. The goal of the game is to destroy all bricks on the display, by making the ball collide with the bricks. The game ends when the player destroys all bricks or when the player fails to block the ball from reaching the bottom of the display.

The implementation of this game required the use of VGA to display the graphics on the monitor, buttons to play the game, and the 7-segment led display for the score. There are four buttons used in the game, two of them to move the paddle left and right, one button to pause the game, and another to reset the game. The current score of the game was displayed on the 7-segment led display, updating and increasing the score each time a brick was destroyed. Our implementation of the game involved heavy use of digital logic in order to control the movement of the ball and track of all the objects to display on the screen. Monitoring the ball's direction and collisions required the use of many different modules all interacting with each other.

Perhaps the most difficult part of this project was that all the data represented in the project had to be implemented in a way so that it could be displayed on the monitor through a VGA connector. Most of the data had to be represented in the form of rows, columns, or pixels. We had to set the frequency of the VGA signal to the correct value to display it on the monitor. Because of the way the VGA displays things on the monitor (line by line, left to right, top to bottom), we had to account for that when designing how to represent all the data.

The color of all the objects on the screen had to be set in terms of their Red, Green, and Blue colors. In order to display things on the screen, the pixel, row, or column at that location would be set to one and when the monitor sweeps over that location, then it would be displayed. To determine when the ball collided with something, we would check if the pixel of the ball overlapped or is going to overlap with the pixel of anything else (paddle, walls, or bricks) at the moment when the monitor swept over that region. To determine the direction of the ball, we would account for the current direction of the ball and which side the ball collided with. We also added the functionality of allowing the user to change the direction of the ball using the paddle. The paddle was split into three regions and depending on which region the ball hit the paddle and the ball's current direction, we would make the appropriate change to the ball's direction as required. In addition, the bricks were represented as a two dimensional array of blocks. This required the use of many different modules in order to repeatedly display the bricks row by row, column by column and was also needed in order to keep track of which blocks were destroyed and which blocks still needed to be displayed.

Design

This project was split into multiple different modules for cleaner code and easier testing purposes. In fact, as we learned while doing this project, we found that some of the logic that we used required the use of modules to implement correctly. The modules that we used directly from the top level pong module are clocker, hvsync_generator, debouncer, paddle, ball, all_bricks, brick_disp, score, score_disp. Most of these modules also contained additional modules which will be discussed in the remainder of the design. Our clocker module has now been used in several of the previous projects and essentially just generates different clocks for different purposes.

The first thing about our design that was essential to making sure that we had anything at all was making sure that the VGA port was working. For the most part, we largely used the ideas of a pong example module that we found online (http://www.fpga4fun.com/PongGame.html). Through this example, while playing around with the parameters that were set for us in the example, we found that the VGA port was actually extremely sensitive to every number and clock frequency that we used and our first day or two of the project was spent on coming to realize the sensitive nature of the VGA. hvsync_generator is the module that we created that largely does all the VGA work for us. The inputs it takes are a 25 MHz clock signal, and it generates all the necessary signals to output correctly to the VGA port. The other values it outputs are cnt_x and cnt_y, which represent the pixel number in the x and y coordinates respectively, which will be used frequently throughout the rest of the code. One thing to notice is that the display itself is actually multiplexed across the screen, in that at any given moment, only a single pixel may be refreshed. However, this happens at such a fast rate that our eyes cannot perceive this normally. Therefore, we can think of cnt_x and cnt_y as the position that our display is currently refreshing. This is useful for a multitude of further applications in our code.

Next, we have our standard debouncer. Since we are using the buttons on the FPGA board as the inputs, we need a debouncer to make sure that the analog, mechanical operation of the buttons are not affected by jitter and noise. As we did in previous lab reports, we essentially just use a lower-frequency sampling of the input to determine when the input has stabilized by testing stability over 4 different samples. This allows us to create four signals representing their buttons: left, right, pause, and reset.

Then we have the first two components of the game that the player can interact with: the border, ball, and paddle. The border is rather simplistic, so it does not have its own module. The original design of these components follow very closely to those sourced in the pong example that we found online (http://www.fpga4fun.com/PongGame.html). However, due to the complexity of the game, we were forced to augment these designs intensely. We also played around with many of the values only to find that the timing was very minute. Each of these components take cnt_x and cnt_y as an input, as they are coordinate based and return a single value ball, border, or paddle which are 1 if at cnt_x and cnt_y they are present. The paddle is also relatively simplistic but there is additional complexity in that we needed to implement the movement of the paddle with respect to the four buttons that we have. An additional thing that we tried to implement (and is still present) was another set of three variables that determined where on the paddle we hit.

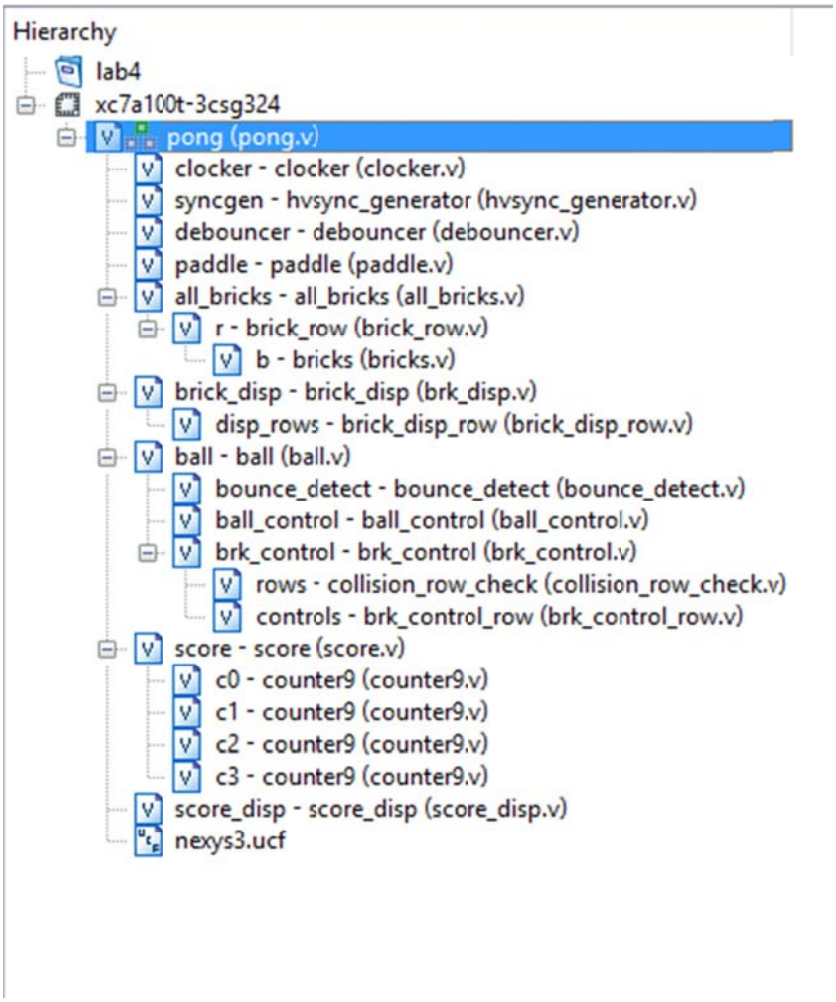
The ball module is, however, very complex, and we split the ball class itself into 3 different modules. Two of these modules deal with the ball collision detection and responsive

movement of the ball. Collision occurs by taking each cnt_x and cnt_y point and determining all the components that are at that pixel at that instant in time. If the ball is at that point and another component is at that point, we consider there to be a collision, and most often this causes a bounce. Since there is also the collision control, this module also deals with when the game is over, if the ball hits the bottom border. The ball_control simply deals with the collisions, if it took place, and changes the ball's position accordingly. It also has the capability of changing the ball's speed, but that functionality is not being used at the moment. The last deals with how to control the bricks when the ball collides with a brick, which will be described with the bricks.

The brick modules are also incredibly complex simply due to the two-dimensional nature of the blocks. Because we wanted to have a 2 dimensional array of blocks, we needed to use an array of the modules, which was able to automatically parse our 2-dimensional bus such that we could create a two-dimensional brick array display. However, as a result of our decision to do this, there were many future consequences. The main thing was that we needed to implement a total brick module and a row module for every module relating to bricks. This is because, as we learned throughout this process, we cannot access the memory since the stacked 2-dimensional buses that we used cannot actually be passed into modules. However, the benefit is we can pass them through an array of modules to lower the dimensions to one in the module to do all the processing and return to increase the dimension again. This allows for us to do some complex packing that would not be possible with single dimension arrays. Like the other components on display, we have a value equal to whether or not a visible brick is at the current cnt_x and cnt_y. In addition, there is a complete 2-dimensional array to pinpoint which of the 64 blocks we have is the one that was hit, so that we can also pinpoint that block to remove, if the ball ends up hitting it.

Related to the brick, we have brick_control and brick_display. These two modules serve as two essential modules in the processing of the bricks. brick_control is a module that is placed in the ball class that outputs a 2-dimensional array just telling which bricks are enabled currently. When the array value is turned to zero, this means the brick is effectively dead (gone from the screen). Since it is gone from the screen, the ball cannot collide with it, and it is not, in terms of the game, in the game anymore, even if it is checked with every clock cycle. The brick display just allows for the game to display the bricks, coloring adjacent bricks differently so that there is no confusion as to where bricks are. It is able of returning a singular value for if any visible brick is hit and another value for all the even blocks and all the odd blocks, for the display purposes.

Finally, the score classes are relatively simple and mirror what we did in lab 2 largely, since we are displaying the score on the 7-segment display. The only difference would be that instead of having a set 1 Hz clock, we used the combinational logic of determining whether there was a collision with a brick or not as our "clock." The only difficulty we had was in making sure that we were able to accurately choose the pulse to tell the score to increment when we hit multiple blocks in quick succession.



Project Hierarchy

Simulation

Our debugging and simulation efforts were done both through iSim and the implementation and just playing the game. Because of the incredibly modularized nature of our code, we were able to do easy tests on many small components of the code. Many modules were tested through a testbench, one of which is shown in the file we created: tb.v.

A lot of our early debugging began with just dealing with the VGA. After acquiring example code, we tried to type our own variation following from the logic we thought we understood from the code. However, nothing appeared on the VGA port. With no way of understanding what was wrong, we just loaded the example code into the FPGA to find the VGA displaying a basic pong game. Since we could not understand all the iSim bit gibberish that the hvsync_generator was producing, we could only try to change values to see what would happen. What we found was that the VGA was very sensitive and there were only very specific numbers that we could change a certain value to, even though the example said that it would change the center of our VGA. In addition, we had to test through implementation the boundaries of our VGA display and how to display things using cnt_x and cnt_y. In the end, we

learned that we had to make sure that all initials are properly set and that cnt_x and cnt_y would represent the locations of each object.

One trouble we had with using iSim, however, was that it did not support the fact that we used 2-dimensional registers. It signalled to us that we probably did not want to use them terribly frequently, but it was ultimately necessary for the simplicity of our code. Therefore, the best we could do was to test our row class. However, since the higher level module usually did not have many instructions other than instantiating the row versions of the class, this was usually sufficient. The 2-dimensional registers proved to be quite a challenge when coding, especially due to the fact that iSim would not work. The 1-dimensional registers were done with relative ease, but when extending our ideas from the 1-dimensional row of bricks to the 2-dimensional matrix of bricks, there was a large increase in ambiguity. When syntax errors came around saying that 2-dimensional inputting was not legal in this mode, even though we had done some similar inputs before, we were really confused. In addition, we learned that we were unable to randomly access the value at any row and column, but we had to first extract the row and then the column by creating new modules. In addition, there was a slight amount of ambiguity in which dimension would be chosen, which worried us at the time of coding.

One concrete example of something that did not extend as we wanted to the second dimension was the brick_control module, which controlled the enables of each brick, in conjunction with the logic of brick collision checking. For some reason, the collision would not transfer the data from the ball to the brick_control module, which was outside of the ball module previously. Instead, we fixed this by using an idea from the testbench, which tested solely the collision check module and the original brick_control module, which worked. As a result, we decided to merge the brick_control and the collision check module together and add it into the ball class, decreasing some of the complexity in the top module in exchange for some invisibility of where the collision logic, which arguably does belong with the ball module, exists. The collision check module still exists in our project, even though it is currently unused.

There was one problem that we ended up not being able to figure out: changing the ball speed as a result of hitting the paddle at specific spots. When running similar simulations on iSim we found that the code seemed to be working as intended and when removing various if statements, it would work as intended to a certain extent (up to the fact that the speed setting could overflow / hit zero, which was not intended). However, when we added the if statements in, for some reason the speed would be set to a constant value of the middle speed. One of the attached testbenches provides test code that tries to find the place of the error, but the simulation showed that the code was working as intended, as mentioned. Therefore, since the speed change was just a stretch goal, we just let it go, leaving the speed as the middle speed.

```
module tb;
reg clk;
reg collision;
reg[7:0] bricks;
reg rst;

wire [2:0] brk;
wire row_collision;

wire [7:0] brk_en;

initial
begin
  clk = 0;
  collision = 0;
  rst = 0;
  bricks = 8'b00000000;

  #500
  collision = 1;
  bricks = 8'b00001000;

  #500
  collision = 0;
  bricks = 8'b00000000;

  #500
  collision = 1;
  bricks = 8'b00000000;
end

always #5 clk = ~clk;

collision_row_check crc (.row_collision(row_collision), .brk_num(brk), .clk(clk), .collision(collision), .bricks(bricks));
brk_control_row bcr(.brk_en(brk_en), .clk(clk), .rst(rst), .brk_num(brk), .row(row_collision));

endmodule
```

Test Bench for collision_row_check & brk_control_row

Conclusion

This project was an open-ended project that required all the techniques and concepts that we learned in the previous labs and also required learning new concepts such as interacting with the VGA connector. For our project, we decided to do an implementation of the game Breakout. Our project required the use of buttons to control the paddle, pause the game, and reset the game, the 7-segment led display for the score, and also required the use of displaying things using VGA.

One of the difficulties we had with this project was keeping track of all the objects to display and correctly displaying all the objects on the monitor. This was especially difficult with the 2-dimensional registers for the bricks as the 2-dimensional registers could not be simulated using iSim. Instead we had to split up managing the 2-dimensional register of blocks into many different modules and test the submodules separately.

Overall our project was successful, even though we ran into a few difficulties along the way. We fulfilled all the requirements of the grading rubric we made in the project proposal. We were able to set up the game and display the all the necessary objects on the screen (Game Setup). We got user input to control the paddle moving it left and right (User Input Functionality). We correctly implemented all the logic of the game - the collisions, the direction and movement of the ball, destroying bricks, losing the game (Game Functionality). We implemented a pause and reset button (Pause/Reset). Lastly, we implemented the score using the 7-segment display that updated every time a brick was destroyed (Score/High Score). In conclusion, our project was a interesting and difficult project that expanded on many of the topics we learned in the

previous labs and challenged us to learn new things, such as displaying things on the monitor using a VGA connector and managing 2-dimensional registers. If we had more time, we would have loved to improve our capabilities with VGA by adding more features to more advanced versions of the game, such as displaying a menu or having power-ups to improve the replayability of the game.