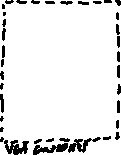
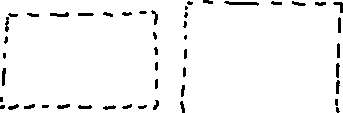
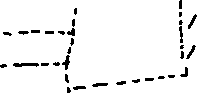
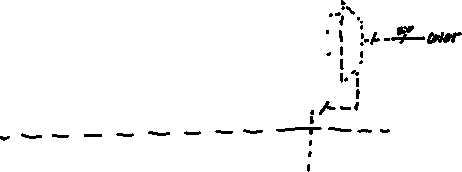
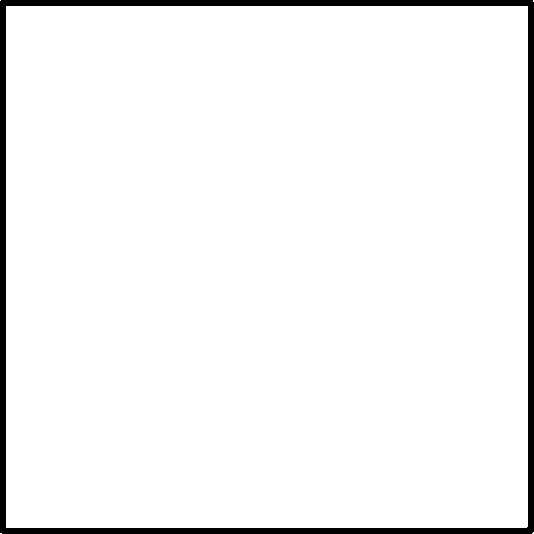
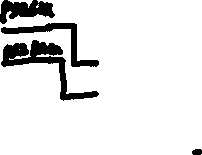
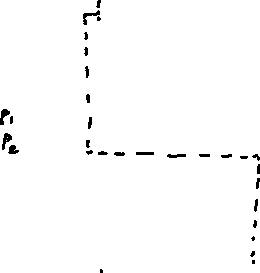
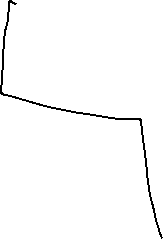
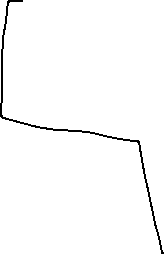
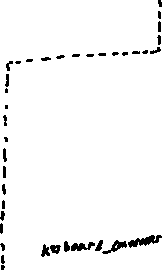
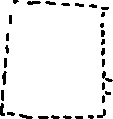
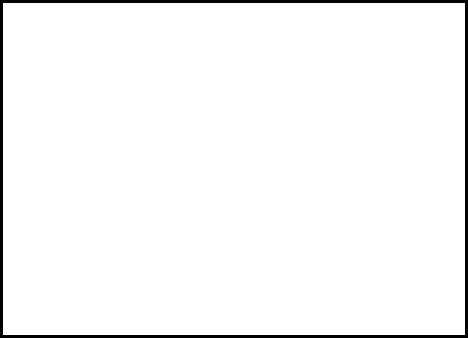
|  |  |
| --- | --- |
| *Title:* | ***Lab #12: FPGA Pong – ball*** |
| *Name:* |  |

# INTRODUCTION

You need to use Xilinx Vivado.

In this final lab, we will introduce the component that calculates all parts of the ball.

Here is a quick overview of what we are trying to accomplish:



# Start Xilinx Vivado

1. Open the top component of your project.
2. Declare and instantiate your ball component. Use the above image to determine what needs to connect where, and declare extra signals as needed. At this point all of your components should be created, and so your top component should be fully defined.
3. Open up ball.vhd, and declare all of the same ports defined above with the correct width.

# ball.vhd Function

The ball component has three primary purposes:

* Move the ball in correspondence to current x and y velocity.
* Calculate when the ball should be drawn
* Determine when the ball might collide with the paddles, and output a score when the paddle isn’t there to block

We will start by drawing the ball.

# Drawing the Ball

* We will need two constants for this: ball\_radius and ball\_rsquared. You can decide the radius yourself corresponding to how large or small you want the ball to be. I chose around 16 pixels.
* Create two 10 bit signals, ball\_x and ball\_y. This is the center pixel value of the ball. Initialize them such that the ball is in the center of the screen. (The resolution of the game is 640x480).

We will now continuously assign the ball\_on wire. The ball needs to be on when both video\_on is high and the current pixel is within the ball’s area. This one is a little hard to grapple, but we will use the Pythagorean theorem to determine if it should be drawn. This is the general idea:



We will compare the values of rsquared (a fixed size, since that is equal to the ball’s radius squared) with the value of . If the latter is smaller, then we know the current pixel is within the ball, and so we draw it. Otherwise, we are outside of the ball, and skip it. Note that we don’t need to care about (ball\_x)-(pixel\_x) versus (pixel\_x)-(ball\_x), since it will be squared anyways.

We must now build the large process block incrementally which will determine if the ball collides with the paddles, and where it moves if it doesn’t. This is a complex block, and so we will start with a piece of code to help.

Before copying over the code, lets define the following constants and signals (those with value **XXX** are to be copied over from your paddle.vhd component):

constant paddle\_length : integer := **XXX**;

constant paddle\_width : integer := **XXX**;

constant left\_border : integer := **XXX**;

constant right\_border : integer := **XXX**;

constant top\_border : integer := **XXX**;

constant bottom\_border : integer := **XXX**;

signal speed\_x: unsigned(2 downto 0) := "011"; -- 3 recommended

signal speed\_y: unsigned(2 downto 0) := "001"; -- 1 recommended

signal dx: STD\_LOGIC := '1'; -- 1 = right

signal dy: STD\_LOGIC := '1'; -- 1 = down

signal count: integer := 0;

signal missed\_ball : std\_logic := '0';

signal ball\_exited : std\_logic := '0';

signal game\_paused : std\_logic := '0';

signal reset\_counter : integer := 0;

Now with all of these constants and signals declared, create a process block triggered on frame and reset. First, check if reset is high. If it is, reset all of the signals to their default values. Else, check for a rising edge of the frame signal. When this signal is given, reset the score\_p1 and score\_p2 ports to zero. We will then check to see if the paddles missed, and if they did, we will stall the reset a little bit. Copy this code in the rising edge of frame if block:

if game\_paused = '1' then

reset\_counter <= reset\_counter + 1;

if reset\_counter >= 120 then

-- reset position and speed

ball\_x <= to\_unsigned(320,10);

ball\_y <= to\_unsigned(240,10);

dx <= '1';

dy <= '1';

speed\_x <= "010";

speed\_y <= "001";

count <= 0;

reset\_counter <= 0;

game\_paused <= '0';

missed\_ball <= '0';

ball\_exited <= '0';

end if;

elsif missed\_ball = '1' then

-- Let ball leave screen

if ball\_x < to\_unsigned(left\_border - ball\_radius - 1, 10) or ball\_x > to\_unsigned(right\_border + ball\_radius + 1, 10) then

ball\_exited <= '1';

game\_paused <= '1';

reset\_counter <= 0;

end if;

-- Keep moving ball

if dx = '1' then

ball\_x <= ball\_x + speed\_x;

else

ball\_x <= ball\_x - speed\_x;

end if;

if dy = '1' then

ball\_y <= ball\_y + speed\_y;

else

ball\_y <= ball\_y - speed\_y;

end if;

else

--YOUR CODE HERE

end if;

Feel free to examine the previous code block, but essentially it will just let the ball fly off screen when a player misses. We will focus on the ball collision logic now. To start, we will handle the top and bottom collisions, since they’re fairly simple. If the ball (- ball radius) is less than the top border, then we change dy to 1. And vice versa for the bottom border. Note that dy is the y direction of the ball. 1 is down and 0 is up. Also remember that we can’t just check ball\_y, since that is the center of the ball instead of the edge.

Now we will check the paddle collisions. Use either a case statement or an if statement with your dx signal to determine if it is going left or right (0 for left, 1 for right). The logic here is thankfully nearly the exact same, but it is still complex nonetheless.

First, in an if statement, check if the ball is in the same plane as the paddle (ball\_x – ball radius <= left\_border + paddle width, flip signs as needed for right side).

If it is, we then check if the ball is in the same horizontal layer as the paddle. To do this, use an if statement to check , (change signals as needed for right side).

Lets start with what happens when this *isn’t* true. Set missed\_ball to 1, and output a score for player 2. If it *is* true, we will flip the dx direction, and add one to count. Then, we will check if speed\_x is below some max value you will determine (5 or 6 is fairly fast). If its below the max value, we will see if the count is above a certain threshold (you choose, this corresponds to how many hits you need before it speeds up). If it is above the threshold, we increment the speed by one and reset the counter.

For the right paddle, do this same logic, but with speed\_y. Do keep in mind that in the previous lab, you determined a max speed for the paddles. You might want to ensure either experimentally or analytically that it is actually possible to react in time and hit the ball, or just make sure the paddle’s y speed is always faster than the ball’s y speed.

After both dy and dx have been handled, the last part of the process block is to move the ball itself. Simply use 2 if statements. If dx is 1 (right), move the ball\_x signal by x\_speed. Do the same logic with the dy signal.

# Compile and test it

Compile your code and synthesize it onto the FPGA. If it works, your game should be complete with the following features:

* W/S and UP/DOWN control the left and right paddles respectively.
* R key resets the game and scores.
* After hitting the ball enough, the ball will speed up to a certain limit.
* Whenever a player misses the ball, the opposing player scores a point

A TA can verify your working code and sign off your participation on their sheet.

# Troubleshooting

With us only instantiating one component today, we can be fairly certain we know where the issue is, so that’s a large plus! Here are some common issues:

* Is the ball really weirdly sized?
  + Check your constant for the ball radius. By adjusting it you should be able to get a size you’re happy with.
* Is the ball covering the entire screen / missing?
  + You’re likely seeing ball\_on as a constant. Look in your continuous assignment, think through when the signal is actually high. Make sure it gets set to zero when its not actively drawing the ball.
  + You could also run a simulation of the software. Once it simulates, add in the ball.vhd file to your waveform, relaunch the simulation, and look at your ball\_on signal. See if it’s a constant high or low.
* Is the ball clipping out of the top/bottom of the screen?
  + If it only clips a little bit, your borders are set too close to the edge. Move the top/bottom borders in a little bit. Alternatively, you may have forgotten to include the radius when determining if it should change direction or not. Keep in mind ball\_x and ball\_y are for the center of the ball, not the edge.
  + If it can fully move off of the screen, your logic for determining when the ball should bounce is incorrect. Look carefully through and think through when it’s supposed to bounce and when it doesn’t
* Is the ball jittering or acting erratically?
  + This is a very common bug and a very odd one. The main issue likely is a typecasting error in your logic. When running your statements determining if the ball is supposed to bounce, notice that ball\_y is of one type (likely unsigned), and the borders are integers. This can cause errors that are tied into how the FPGA casts this logic. To fix this, you must convert your integers into unsigned. for a reminder, the syntax in this case is

to\_unsigned(integer, (size)'length)

Integer is your entire calculation (could be just a border, border + speed, etc), and size is the size of the unsigned you are comparing it to. For example..

ball\_y + ball\_radius >= to\_unsigned(bottom\_border, 10)

would be the correct casting for the ball bouncing off of the bottom wall.

One important thing to note: the \*\* exponential operator expects integers, so when you’re drawing your ball, typecast both pixel\_x and ball\_x into integers (pixel\_x might need to be cast to an unsigned first). Do the same with the y values.