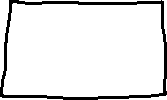
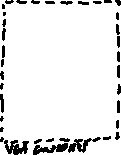
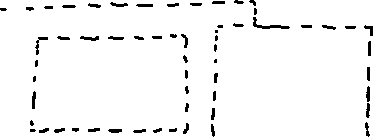
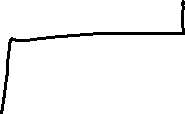
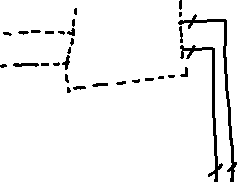
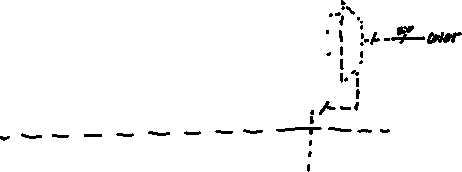
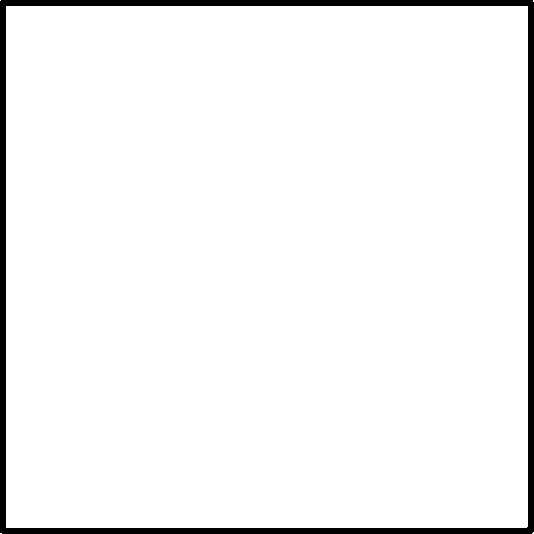
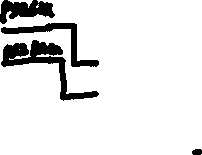
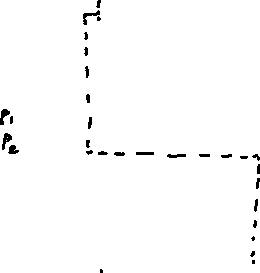
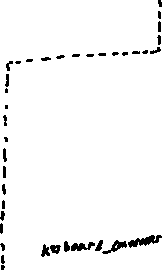
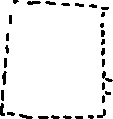
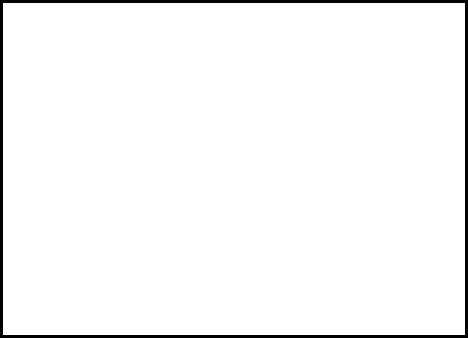
|  |  |
| --- | --- |
| *Title:* | ***Lab #11: FPGA Pong – paddle*** |
| *Name:* |  |

# INTRODUCTION

You need to use Xilinx Vivado.

In this lab, we will introduce the first actual part of our user interface: two interactive paddles.

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



# Start Xilinx Vivado

1. Open the top component of your project.
2. Go ahead and declare and instantiate your paddle component. Use the above image to determine what needs to connect where, and declare extra signals as needed. If a component doesn’t exist yet (such as p1y to the ball module) just declare the signal and leave it unused.
3. Open up paddle.vhd, and declare all of the same ports defined above with the correct width.

# paddle.vhd Function

The paddle component has three primary purposes:

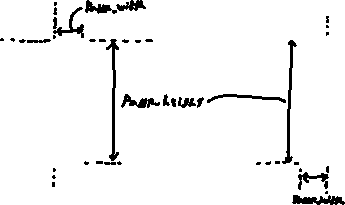
* Move the paddles in response to player input
* Calculate when the paddle should be drawn
* Output the current paddle locations for collision calculations in the ball component.

We will start by drawing the paddles.

# Drawing the Paddles

* We will need four constants for this: the left border, right border, paddle height, and paddle width. You can decide these values yourself corresponding to how large or small you want the paddles to be. Your borders should not quite be the edges of the screen, try shifting them 20 pixels or so towards the center instead.
* Create two signals, left\_paddle\_y and right\_paddle\_y. This is the y value of the top slice of the paddle. Initialize them such that the paddles are in the center of the screen. (the height of the game is 480 pixels). Keep in mind that just setting it to 240 will not be perfect, since these signals are the top of the paddle, not the center.
* We can already assign two of our ports, p1y and p2y, to these newly created signals. Just continuously assign these.

We will now continuously assign the paddle\_on wire. The paddles need to be on when both video\_on is high and the current pixel is within the paddle’s area. Using the left\_border, right\_border, paddle\_width, and paddle\_height constants, as well as the left\_paddle\_y and right\_paddle\_y signals, you can determine what the exact area is of each paddle, such as:



Now that the paddles are drawn, we will define three new constants: top\_border, bottom\_border, and paddle\_speed. These three constants can also be whatever you’d like. Having a paddle speed somewhere between 3 and 5 pixels per frame is a fairly good balance of speed and ease of control to start with. The top and bottom border should be placed not quite at the edge of the screen, but a few pixels in.

We must now build the logic determining if they are allowed to move, and how to actually move them. This logic will take place in a process block, triggered by the frame and reset signals. First, we will handle the reset. Whenever the reset is high, set the paddles back to the middle of the screen.

This next part is a little complicated, but the logic applies to both the left and right paddles. The inputs p1 and p2 are two bit vectors that correspond to the buttons being pressed. The MSB is a ‘hold’ bit. When this is active, the paddles stay still (this is if both or neither inputs are pressed). The LSB is the ‘direction’ bit. In our case, 0 is down, 1 is up. For more information, look at the keyboard\_controller.vhd file to see how the values are actually assigned.

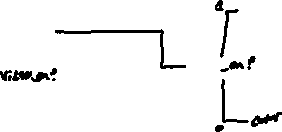
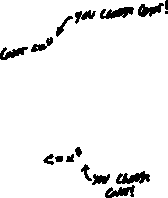
On the rising edge of each frame, we will look at these inputs. with a case statement, we need to implement the following (showing the left paddle / p1):

|  |  |
| --- | --- |
| **Case** | **Action** |
| 00 | Move down if not past bottom border |
| 01 | Move up if not past top border |
| 10 | Hold bit- No action |
| 11 | Hold bit- No action |

The hold bit is fairly self explanatory, just fill in a *“null;”* case for no action. For the others though it gets more complex. When its moving up, we will check to see if the current y position is greater than the top border plus the paddle speed. If it isn’t, we will subtract the paddle\_speed from the current paddle\_y (we subtract since it is moving up, or getting closer to 0). When moving down, we see if the paddle\_y + the paddle\_height is less than the bottom\_border – paddle\_speed. If not, we can add the paddle\_speed to the current paddle\_y

# Drawing colors

The last step before synthesizing our code is to draw the actual paddles on top.vhd. right now, they aren’t actually doing anything. We will create a process block, sensitive to video\_on, and paddle\_on for now (we will add ball\_on next lab). For this lab, the logic is simple. Implement the following branches:



This can be completed in a number of different ways, but the easiest is usually a nested if statement.

# Compile and test it

Compile your code and synthesize it onto the FPGA. If it works, you should be able to use the keyboard inputs (W/S and UP/DOWN) to control the paddles. The background should be your paddle\_on = 0 color, and the paddles should be the paddle\_on = 1 color.

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:

* Are your paddles really weirdly sized?
  + Check your constants for the left and right borders, the paddle height, and the paddle width. By adjusting them you should be able to get a size you’re happy with.
* Are the paddles covering the entire screen / missing?
  + You’re likely seeing paddle\_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 paddles.
  + You could also run a simulation of the software. Once it simulates, add in the paddle.vhd file to your waveform, relaunch the simulation, and look at your paddle\_on signal. See if it’s a constant high or low.
* Are the paddles clipping out of the top/bottom of the screen?
  + If they only clip a little bit, your borders are set too close to the edge. Move the top/bottom borders in a little bit.
  + If they can fully move off of the screen, your logic for determining when the paddles can move is incorrect. Look carefully through and think through when it’s allowed to move and when it’s not.
* Are the paddles 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 paddle is allowed to move, notice that paddle\_y is of one type (likely unsigned), and the borders/paddle\_speed 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..

left\_paddle\_y >= to\_unsigned(top\_border + paddle\_speed, left\_paddle\_y'length)

would be the correct casting for the left paddle up check