Illya Kuzmych (2127069), Ruvim Piholyuk (2128297)
EE/CSE 371 22sp
Due June 6, 2022
Lab 6 Report

Pong

Design Procedure:

In this lab we were tasked with implementing multiple peripherals in a high-level FPGA design. We used a source of significant memory storage, VGA display peripheral, and user inputs via a N8 controller to recreate the classic game Pong. We utilized a DE1_SoC FPGA in order to test our modules, from which we used the left and right signals from the N8 controller to direct the position of the user's paddle which would move only one time for every time the button was pressed. Also, we utilized the start button on the N8 controller as a reset button in order for the player to reset the game once they have lost and the screen has been cleared. In order to ensure the ball would "bounce" according to its own trajectory we use the "line drawer" module from Lab 5 to essentially track the center of where the ball should be and to draw a circle around the position. Using the inputs from the user, the system will draw around what the user is expecting (ball and paddles) and update their positions depending on user input.

Our most difficult task was designing the algorithm that changed the trajectory of the ball based on whether it hit a paddle (and what part of the paddle it hit), or whether it hit a wall. In order to decrease complexity, we had the AI paddle's center always be in line with the center of the pong ball. This reduced logic and still allowed for an enjoyable game that tested the user's skill. In order to create a functional program, we used the circle's position to determine its next behavior: whether to lose the game, or bounce in an appropriate direction. We decided that upon hitting a wall, the ball should negate its current slope and keep traveling the same vertical direction until it reaches a paddle or misses it. However, the change in slope was necessary for whenever the user's paddle hit the ball, as simple game physics dictated that if a ball hit the paddle farthest from the center of it, the ball should travel in a shallower slope, increasing in slope the closer the ball hits the paddle's middle. We divided our paddle into seven sections: 3 for each side and one for the middle. We implemented a ROM module that stored different slope targets that would read based on the ball's position in relation to the position of the paddle, utilizing a generated .mif file to store the initial slope contents. For example, if the ball was in position 2 (on a 0-index based ROM), then we would give it the steepest slope, as it was closest to the middle aside from hitting the middle of the paddle coming from the left side. However, if it was in position 6 (farthest position to the right), then we would give it the shallowest slope, held in address 6 in the

ROM. This algorithm significantly increased the complexity of our design and allowed us to creatively expand on our bare-bones implementation prior.

Task #1 (Pong):

In Task #1 We began with deciding how we were going to represent the “pong ball” for our game. We decided that we would be able to manipulate our line drawer algorithm that we created in Lab 5. We adjusted the algorithm to instead of passing in x0, y0, x1, and y1 as inputs, to instead pass in x0 and y0 and a new slope value and to calculate x1 and y1 values internally based on the initial value of the line and provided slope. This allowed us to generate a trajectory for a ball from any position on the VGA screen. We accomplished this by using the slope formula to configure the positions of x1 and y1 depending on the position of the ball on the screen and the desired slope. Once x1 and y1 values were calculated, the module output an x and y value that were the working center point of the ball. Figure 1 shows the logic we used to generate a ball utilizing the VGA display driver, utilizing the geometrical equation for a circle and checking if the scanned values were within the perimeter of the circle, and based on that would trigger the corresponding RGB values. In the figure below, x and y circle position variables represent the center coordinates of the “ball”, and x and y represent the current pixel being scanned by the VGA.

```
if ((((x - xCirclePosition)**2) + ((y - yCirclePosition)**2)) <= (10**2))
   r <= 8'd255;
   g <= 8'd255;
   b <= 8'd255;
```

Figure 1. Formula for the shape and position of “Pong Ball”

And in Figure 2 we display the logical block we used to generate the new x1 and y1 values based on the slope. Even though it would sometimes generate x1 and y1 values that are out of bounds or “off-screen” for the VGA, the logic in Task #3 took care of any ball that would get close to the side walls or the paddles.

```
if (~start || reset) begin
   if (x0 == 629) begin
      x1 <= 10; y1 <= y0 + 609 * slope;
   end // if (x0 == 629)

   else if (x0 == 10) begin
      x1 <= 629; y1 <= y0 - 619 * slope;
   end // else if (x0 == 10)

   else if (y0 == 459) begin
      y1 <= 20; x1 <= x0 + 439 / slope;
   end // else if (y0 == 459)

   else if (y0 == 20) begin
      y1 <= 459; x1 <= x0 - 439 / slope;
   end // else if (y0 == 20)
end // if (~start || reset)
```

Figure 2. Logic to generate x1 and y1 based on x0, y0 values and the slope

Task #2 (Paddles):

In Task #2, our goal was to create a user-controlled paddle towards the bottom of the VGA screen and a computer-controlled paddle which followed the position of the ball. We included the N8 controller as part of our design here to take in user input and move the user-controlled paddle left and right across the screen. We also implemented a "onePress" module that would only allow the user-controlled paddle to move in a specified direction for a single click, and a user would have to let go of and re-press the keys to move the paddle again. This prevented a game where the reaction time of the user would have to be on par with the speed of the 50 MHz clock. Initially, or upon reset, we set a default value for the user controlled paddle value to be 84 pixels wide, as to have 7 different 12-pixel wide sections on the paddle to work in correlation with the ROM-slope-algorithm utilized in Task #3. Upon a left or right trigger from the N8 controller, the user-controlled paddle moved 10 pixels in the corresponding direction and stayed put unless redirected. On the other hand, the computer-controlled paddle was set to follow the trajectory of the pong ball, thus never losing and always hitting the ball back from its center. Both paddles used data points to hold the points of the edge of each paddle and the vgaOutputs module outputs RGB values corresponding to each objects position. For example, if the user paddle was scanned, then its color was set to white. If the computer paddle was scanned, then its color was set to red. Figure 3 shows the logic we used in vgaOutputs to generate a 84-pixel wide computer paddle that followed the position of the ball.

```
else if ((y >= 9'd0) && (y <= 9'd10) && (x >= (xCirclePosition - 10'd42)) && (x <= xCirclePosition + 10'd42)) begin // computer-controlled paddle
   r <= 8'd255;
   g <= 8'd0;
   b <= 8'd0;
```

Figure 3. Logic to generate a computer-controlled paddle of red color

Task #3 (Collisions):

In Task #3 our goal was to utilize the logic for the pong ball and paddles to update the trajectory of the ball when the ball either hit the left or right walls or either paddle on the screen. Also, to clear the screen when the ball is at the bottom and is not touching the user-controlled paddle. We

did this by looking 10 pixels away from the (x,y) coordinates given from our updated Line Drawer module for a wall or AI paddle. If we found either, we multiplied the slope by negative one and used logic from Line Drawer to decide the trajectory of the ball based on the new slope and logic to find new x1 and y1 from Line Drawer. If the user paddle was hit we found a new slope based on the position of the ball on the paddle. We utilized a single port ROM that holds data to change our slope based on the part of the paddle that was hit. Then the trajectory was set in Line Drawer once again and the score incremented and was output to the 2 7-Seg Hex displays HEX1 and HEX0. In order to ensure that we only updated trajectories once a barrier was just hit we have logic for the Collisions module to only update trajectory when the check flag was true. Right after a collision was made, the check logic was turned off and was only turned back on after the ball had moved at least one pixel away from the barrier. Figure 4 shows the logic which we built our trajectory-changing algorithm on, and how we implemented a ROM to update the current slope after the ball hit the paddle.

```
        */
       if (((paddlePositionChecker >= 0) && (paddlePositionChecker <= 6))) begin
          lose <= 0;
          collisionTrue <= 1;
          score <= score + 1;
          x0 <= circleX;
          y0 <= circleY;
          if (paddlePositionChecker < 3) begin
             currentSlope <= ROMSlope * -1;
          end // if (paddlePositionChecker < 3)

          else if (paddlePositionChecker == 3) begin
             currentSlope <= currentSlope * -(ROMSlope);
          end // else if (paddlePositionChecker == 3)

          else begin
             currentSlope <= ROMSlope;
          end // else
       end // if (((paddlePositionChecker >= 0) && (paddlePositionChecker <= 6)))
    end // if (circleY == 459) && check)
end // always_ff
```

Figure 4. Logic to change the trajectory upon hitting a paddle

System block diagram:

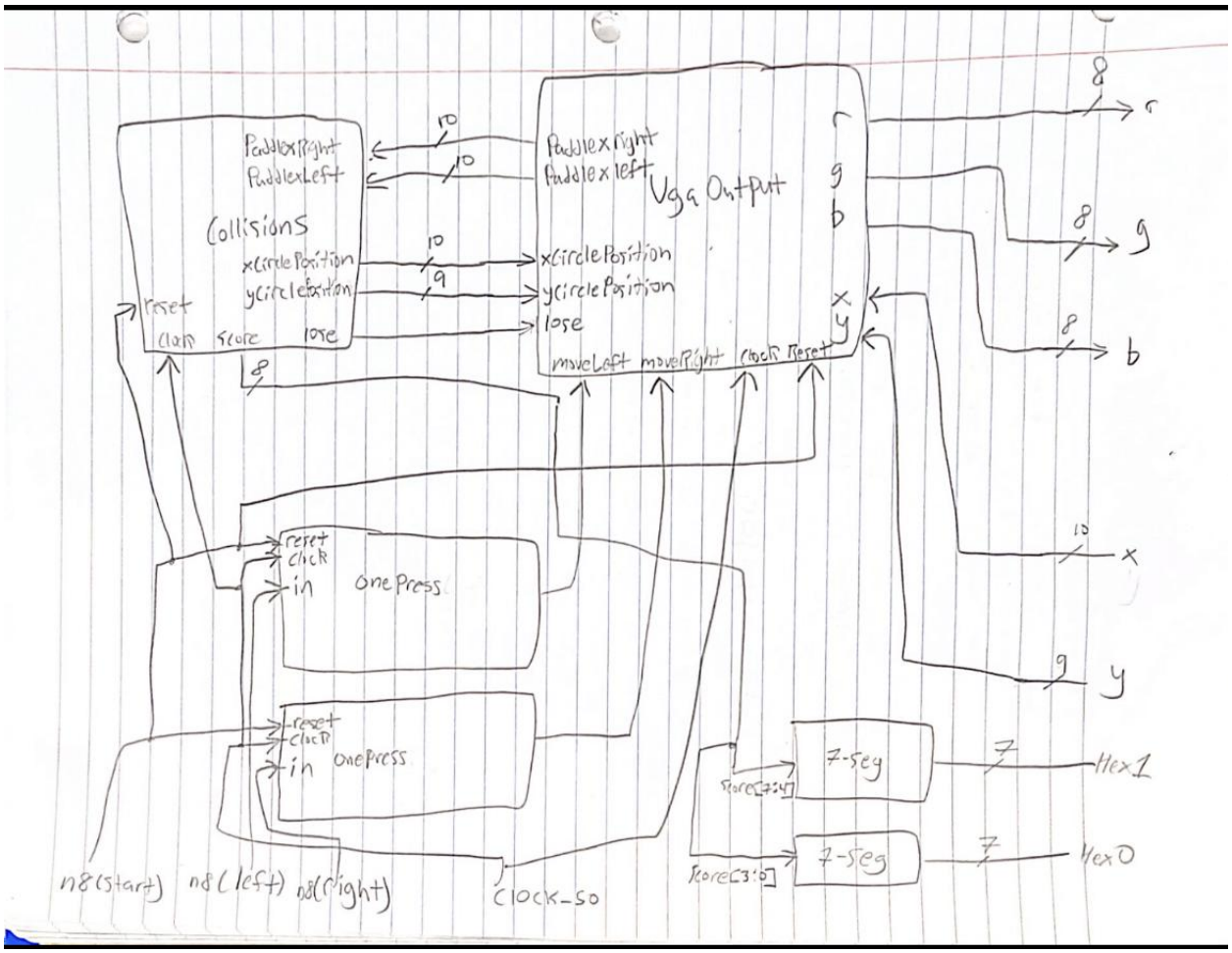


Figure 5. System block diagram for Lab6 implementing high level modules for "Pong"

The image in Figure 5 is a high level representation of how our system works to implement the classic game of Pong. We pass down the left and right arrows from the n8 controller in order to interact with the onePress module so that every time the arrow is pressed the paddle will not move again until the corresponding direction is unselected and selected again. The start button on the n8 controller is used in place of a reset button to press when the player loses and the screen clears. Collisions decides where to draw the pong ball and the trajectory of the said ball while vgaOutput decides where the user and AI paddle goes and outputs the colors and placement of each part of the same to the vga. Score is also output from Collisions so the player can track their score in hex on LED1 and LED0.

Results:

Overview:

In this lab, we experimented with and developed our skill in designing and implementing more algorithms while utilizing multiple peripherals at the same time. After some careful planning, we decided to implement the classic arcade game “Pong” on hardware utilizing the N8 controller for user input and the VGA screen to display our game. In order to create a functional system we needed to alter our old line_drawer algorithm to take only start x and y coordinates alongside a slope, which would then internally generate the trajectory of the new line which dictated the direction of our ball. Then, we built a simple program that altered RGB colors at specific indices on the VGA screen if they overlapped with the current position of our user and computer-controlled paddles. The simulations below show how we tested and implemented each part of our design.

line_drawer:

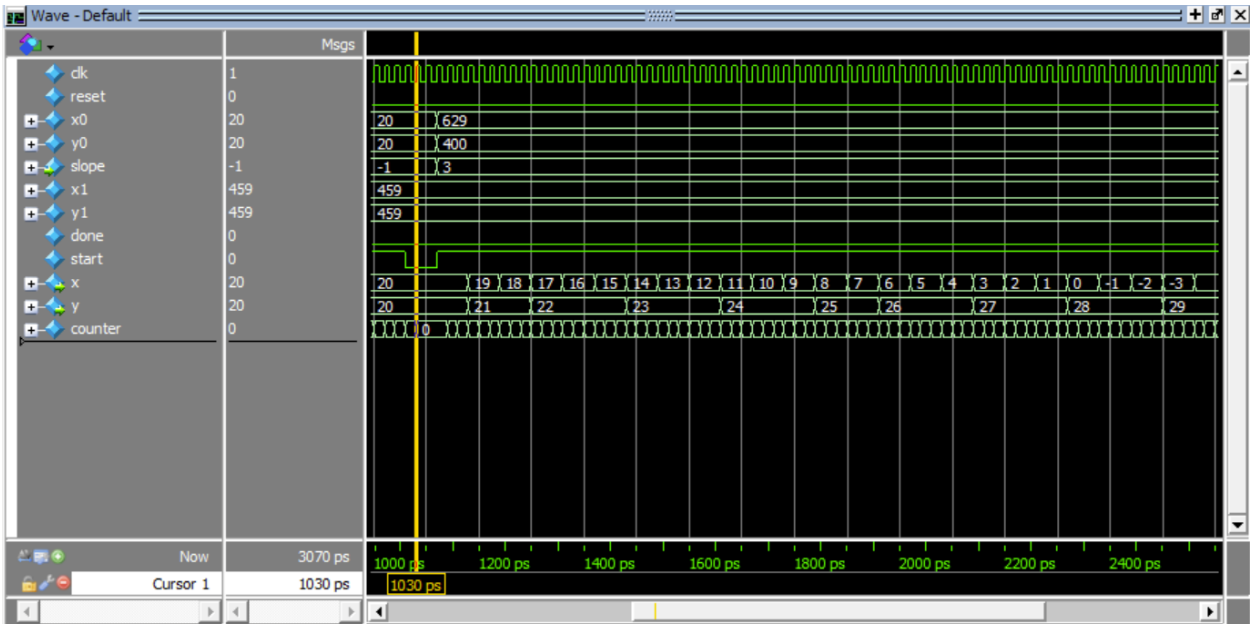


Figure 6. Simulation for the new line_drawer algorithm

In Figure 6 we show our completely functional, new implementation of the old line_drawer algorithm that we implemented in Lab 5. Our main change is that now instead of passing down the beginning and end coordinates of the line, we pass in the start coordinates and slope which directs the trajectory of the line. For example, in our simulation we show an example which is commonly hit during the Pong game. Here it shows what would happen if the computer-controlled paddle hit the ball towards the right-most wall from the angle of the user. The ball hits the wall, updates its start coordinate with the new slope,

and then updates the x and y outputs which are then sent to the VGA to color. We implemented line_drawer in collisions to constantly update the VGA which then updated the colors in vgaOutputs.

collisions:

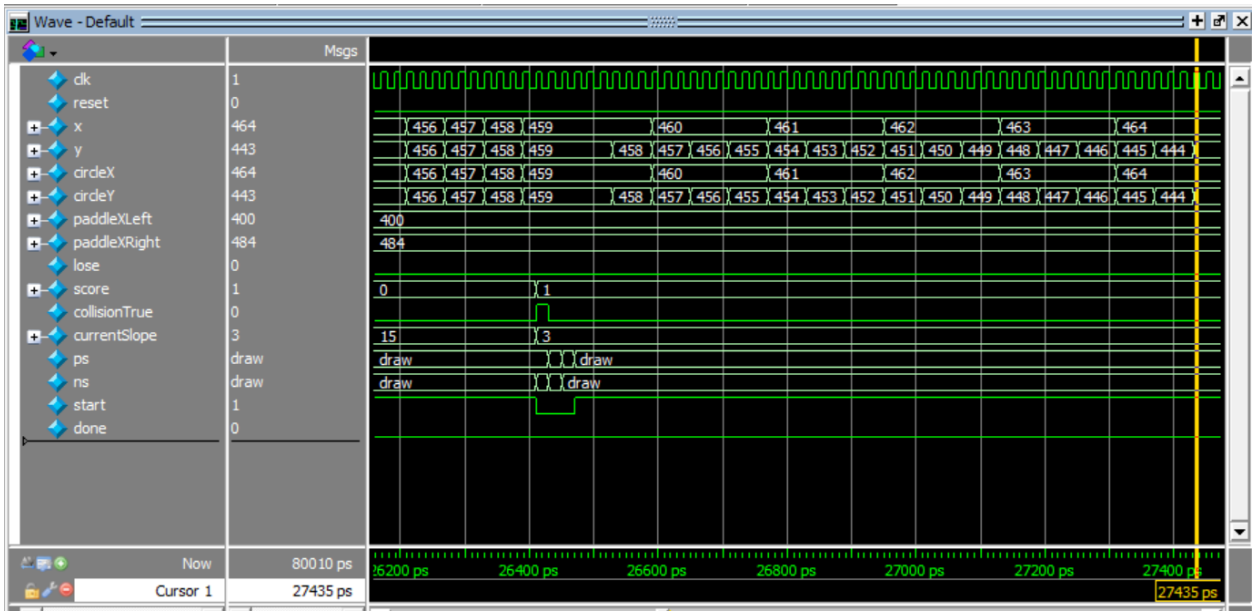


Figure 7. Simulation for the Collisions Module

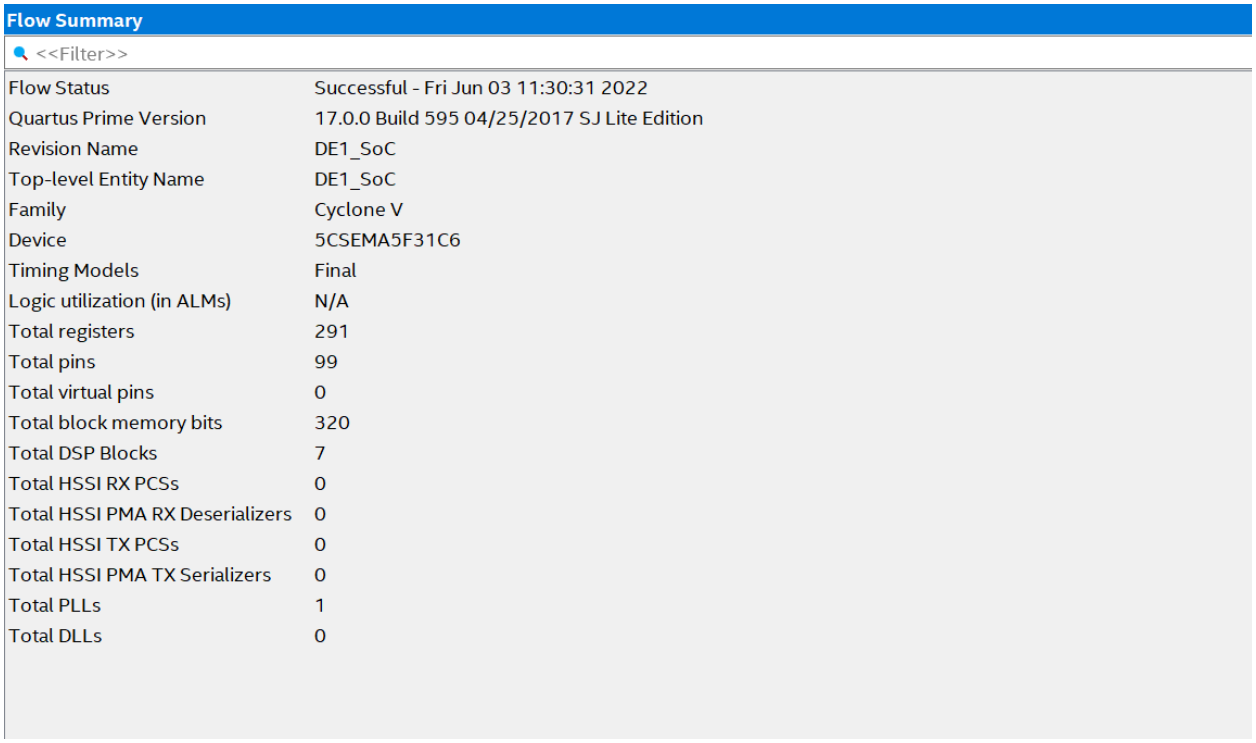
In Figure 7 We convey the functionality of the Collisions module by showing the behavior of the “ball” when it is intended to bounce off of the user paddle. We see that the x and y values from the ball are approaching the user paddle and once it is hit the score increments from zero to one. Also, see that the trajectory is updated via the line_drawer module, current slope logic once a collision has happened. The ball is headed in a new direction where it will hit another barrier and the same logic will be applied to update the trajectory.

vgaOutputs:

Figure 8. Simulation for the VgaOutputs Module

In Figure 8 We convey the functionality of our VgaOutput module that is essentially tracking the placement of the user controlled paddle and the AI paddle. That same logic is passed down to the Collisions module in order to track collisions based on the position of the paddles. Also, we can see that the center position of the ball is passed from the Collisions module and the colors for the ball are set based on that position and the positions of the paddles. The user controlled paddle and the ball are both white and the AI controlled ball is red. Simulation is done at a smaller scope because to draw the entirety of the paddles and the ball will take over 307,000 clock cycles and giving the smaller window gave us the proof that the logic in VgaOutputs is working as we expected.

Flow summary:



Flow Summary

<<Filter>>

Flow Status	Successful - Fri Jun 03 11:30:31 2022
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	291
Total pins	99
Total virtual pins	0
Total block memory bits	320
Total DSP Blocks	7
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1
Total DLLs	0

Figure 9. Flow summary of the overall system in Lab 6

Experience Report:

Our group found the lab to be difficult. Our biggest hurdle was implementing logic to deal with collisions, especially against the user-controlled paddle. We had difficulty figuring out how to implement our design in a straightforward manner and keep track of the ball coordinates. We then realized that changing the line_drawer to instead take in the start coordinates and the slope allowed us to easily predict where the ball is going. In collisions we kept track of the center of the ball and never let the ball go outside the VGA screen by setting flags that would trigger every time the ball was near any of the borders. Implementing the collisions module was then a bit easier, as we simply paused the game for a couple of clock cycles to allow the registers in line_drawer to update upon a collision, and then re-initialize the movement of the ball. This lab took us approximately 31.5 hours, broken down as follows:

- Reading – 30 minutes
- Planning – 4 hours
- Design – 4 hours

• Coding – 8 hours
• Testing – 5 hours
• Debugging – 10 hours

Partner Work Summary

	Ruvim	Illya
Reading	✓	✓
Planning	✓	✓
Design	✓	✓
Architecture	✓	✓
Coding	✓	✓
Code Commenting		✓
Testing	✓	✓
Debugging	✓	✓
Report	✓	✓

We did the lab nearly entirely together over zoom working on the same tasks.