Lab 8: Pong Game

Purpose

In this lab, you will learn about:

- The creation of a complete application-specific, interactive system
- The control of a datapath using a finite-state machine
- Further exploration of processes and behavioral description in VHDL
- Additional techniques for integrating human input with a digital circuit

This lab will require you to design most of the modules in VHDL. If you are not comfortable with the concepts you have covered so far, you should review the VHDL lecture slides and modules you designed in earlier labs prior to coming to class.

Background Information

The game *Pong* is one of the first complete, standalone digital arcade games to reach mainstream popularity. *Pong*, released in 1972, is a simple tennis-like game that pits two players against each other on an almost-empty screen. A small, square ball bounces back and forth between paddles controlled by two players, who attempt to bounce it back at their opponent. If either player fails to hit the ball (it goes off the edge of their screen), that player loses the round, and their opponent gets a point.

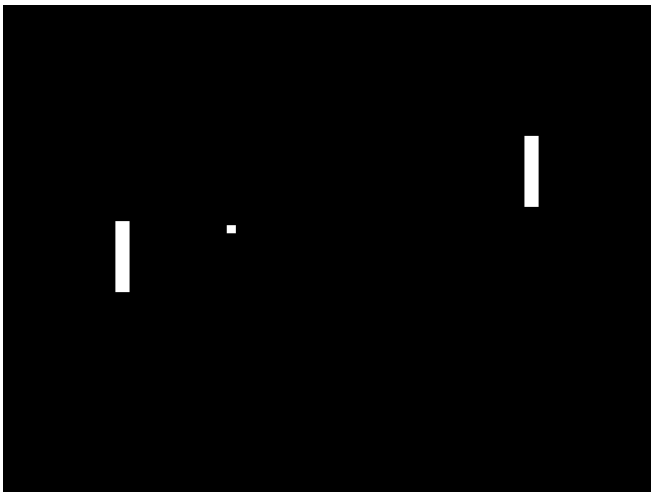

Figure 1: Mock-Up of *Pong*

The game of *Pong* consists of three main "entities": two paddles and the ball. You will design a datapath to control each of these entities and manage the flow of the game. Inside this datapath, we will implement a module for each entity. Each module will track the X and Y coordinates of its entity. Because the game "screen" will be 640 by 480, all X coordinates will be represented in ***10 bits*** (values from 0 to 1023, though only 0 to 639 should be considered valid). To simplify the design, we will also use ***10 bits*** to represent Y coordinates. As with the VGA controller, the ***top***

left corner is the origin (X=0, Y=0). Higher values of X move right, while higher values of Y move down.

First, we will discuss the design of the *paddles*. Each paddle has a fixed X-coordinate on the screen, and the only thing that will change is the Y-coordinate as the user moves the paddle up and down. The *paddle* module will take two one-bit inputs: "U", telling the paddle to move up, and "D", to move down. If the U input is 1, the paddle will move up by 4 (***decrease*** Y by 4). If the D input is '1', the paddle will move down by 4 (***increase*** Y by 4). If both are '1', the paddle should stay in position.

There is one slight trick to this: we need to prevent the paddle from going off the screen. To handle this, you will perform the following check: if Y < 100 don't decrease Y, and if Y >= 440 don't increase Y. Additionally, to prevent having to design two separate components, we will use two instances of the same paddle module to implement two paddles. To recap, the paddle module will have the *interface* (input and output ports) shown in Figure 2.

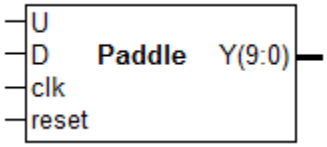

Figure 2: Interface to the Paddle Module

Next, we will discuss the *ball*, which will bounce back and forth from one player's side to the other. For the ball, we will track the current direction it is traveling in separate X and Y-components, or its *velocity*. We will store the velocity using two flip-flops, one for X and the other for Y. One flip-flop will tell us whether the ball is moving in the positive-X direction (1, ***right***) or the negative-X direction (0, ***left***), and the other will do the same for Y (1 is ***down***, 0 is ***up***). These values are recapped in Table 1 and Figure 3.

X	Y	X direction	Y direction
0	0	Left	Up
0	1	Left	Down
1	0	Right	Up
1	1	Right	Down

Table 1: Ball Direction Encodings

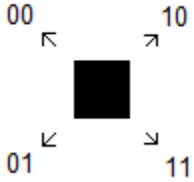

Figure 3: Ball Direction Encodings (X = MSB, Y = LSB)

When the ball collides with a paddle or the border of the screen, the ball will bounce away from the object it collides with. This will be done by reversing direction away from the collision. Using the velocity encoding table above, this can clearly be accomplished by flipping either the X or the Y bit. Therefore, we will use ***T flip-flops***

to store the X and Y direction. Whenever we want to change direction, we will pass a '1' to the corresponding flip flop. All of the possible collisions in the *Pong* game are shown in Figure 4, along with the flip-flop that will toggle in each case.

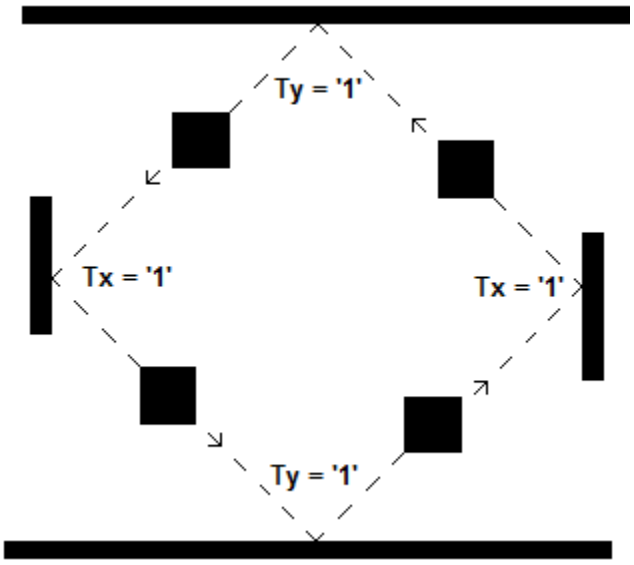

Figure 4: Possible Collisions in *Pong*

From Figure 4, we can see that there are only four possibilities:

1. The ball collides with the top border (Ty = '1')
2. The ball collides with the left paddle (Tx = '1')
3. The ball collides with the bottom border (Ty = '1')
4. The ball collides with the right paddle (Tx = '1')

You will be designing a module (in part during the pre-lab) that will detect which of these collisions has occurred. The interface to this module is shown in Figure 5.

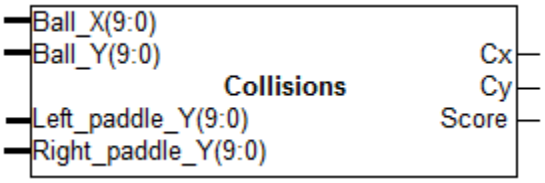

Figure 5: Interface to the Collisions Module

This module will take, as input, the X and Y coordinates of the ball, as well as the Y coordinates of both paddles. If there is a collision that would require the ball to change its velocity in the X-direction, **Cx** will be '1'. Likewise, if a collision requires the ball to change its velocity in the Y-direction, **Cy** will be '1'. *Observe that we can connect these **Cx** and **Cy** outputs directly to the inputs of the T flip-flops for the velocity flip-flops!* Also note that certain constant values, like the width and height of all entities, as well as the X coordinates of the paddles, are all handled internally. Finally, this module also has a ***Score*** output that will be '1' when either player scores (the ball goes past the opponent's paddle untouched).

Next, we will discuss the module that will move the ball's position as the game progresses. This module will have four direction inputs: XU, XD, YU, and YD. These correspond to moving the X or Y coordinate of the ball either up (U) or down (D). These inputs will be connected directly to the Q and Q' outputs from the direction/velocity T flip-flops. Additionally, because this module has sequential logic, it will require clock and reset signals. Finally, it will also have a ***hold*** input: whenever this *hold* input is '1', the values of XU, XD, YU, and YD will be ignored, and the ball will be set to its starting position (X = 640/2 = 320 and Y = 480/2 = 240). This module will output the current X and Y position (each ten bits) of the ball on the screen.

The movement of the ball, as a function of XU, XD, YU, YD, is given as follows:

1. If XU = '1' then X = X – 2
2. If XD = '1' then X = X + 2
3. If YU = '1' then Y = Y – 2
4. If YD = '1' then Y = Y + 2

The case where XU and XD are both '1' will be ignored (as they are coming from complementary outputs of a flip flop, this should not happen). The same holds for the case where YU and YD are both '1'. Note that unlike the paddle module, the ball module does not need to wrap (check bounds) for the X and Y coordinates, because this will happen naturally thanks to the collision module toggling directions. To recap, the interface to the ball position (Ball pos.) module is as shown in Figure 6.

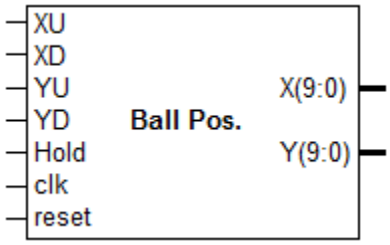

Figure 6: Interface to the Ball Position Module

If you have been careful, you might notice that there is a big problem with this design: if the above logic is evaluated every clock cycle at 25 MHz, the ball will move around the screen too quickly for the users to see it. This is similar to problem that the one-pulse module was used to address in previous labs, and we will use a similar solution here, called the "one-pulse-per-cycle" module (OPPC for short).

The OPPC module will take *two* inputs (**Di** and **E**, in addition to clock and reset) and have a single output **Do**. The OPPC module will ensure that when the input **Di** becomes high, the output **Do** will become high for a single clock cycle. The output will not become high again (regardless of the input value) until the other input **E** goes through a *complete* cycle (goes from 0 to 1 and then back to 0). This serves to limit the rate at which our datapath runs to the rate at which **E** cycles. We will provide a module to generate a suitable signal for **E**.

For simplicity, you can assume that whenever **Di** goes from 0 to 1, the value of **E** will still be 0. Example simulation output is provided in Figure 6 below. Make careful

note that if **Di** = '1' for multiple **E** cycles (from 400 ns to 700 ns below) the output will become '1' once at the start of each new **E** cycle.

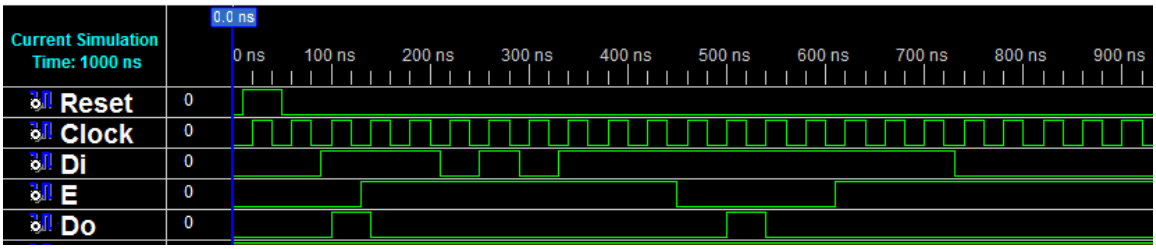

Figure 6: Sample Simulation for the One-Pulse-Per-Cycle Module

We will place an OPPC module on the **Cx** and **Cy** outputs from the Collisions module, as well as both outputs of the T flip-flops that go to the ball position module.

The completed datapath will be laid out as shown in Figure 7. The datapath will have seven one-bit inputs: **clock** (25 MHz) and **reset**, **hold** (discussed later), and **LU**, **LD**, **RU**, and **RD** (which are the user buttons for moving both paddles up and down). This module will have one one-bit output **Score**, and four ten-bit outputs **Ball_X**, **Ball_Y**, **Left_paddle_Y**, and **Right_paddle_Y**. You should follow Figure 7 as a loose guideline, as it may not match your design exactly.

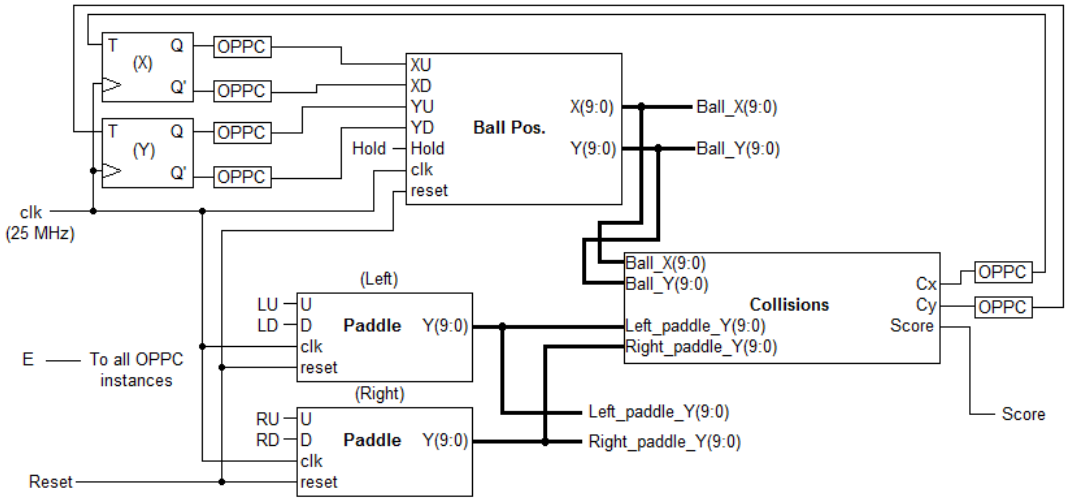

Figure 7: *Pong* Datapath

Note that the OPPC modules obviously also require connections to the clock, reset, and E inputs (as indicated), but these are omitted in Figure 7 for brevity.

The *Pong* game will be displayed on a monitor using the VGA port of the Virtex II Pro board with the VGA controller you have already designed. For simplicity, we are providing you with a bundled module that includes a working implementation of the VGA controller, as well as a module to draw the ball and paddles (given their positions). This module takes as input a 25 MHz clock, a reset signal, the 10-bit X and Y coordinates of the ball, and the 10-bit Y coordinate of both paddles. This module outputs the VGA control signals (HSYNC, VSYNC, BLANK_Z) and RGB color signals. Finally, this module also has an output **E** that will serve as the "rate-

limiting" signal for the OPPC modules, and needs to be passed back to the datapath. The interface to this module is shown in Figure 8.

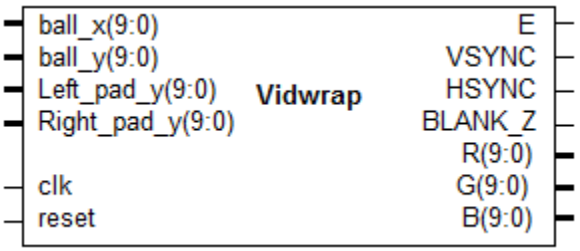

Figure 8: Interface to Video (VGA) Wrapper Module

Finally, you will also need to implement a controller FSM to accompany the datapath. This controller will take two inputs (in addition to clock and reset signals): one will be a **Start** signal from a push button that starts the game, the other is the **Score** output from the datapath indicating that a round has ended (one player scored). This FSM will have a single one-bit output **Hold**, which will be sent to the datapath.

The control FSM will function as follows, and can be implemented as Moore machine with two states:

- The FSM should tell the datapath to hold until Start is pressed
- Once start is pressed, the FSM should not hold until a player scores
- Once a player scores, the FSM should hold again until Start is pressed

The top level design will also make use of the clock generator module you made in Lab 6, which divides the 100 MHz system clock by 4 and buffers it, resulting in a 25 MHz clock.

Every module in your system that uses a clock (except the clock generator) should be connected to the <u>same</u> 25 MHz clock!

Summary and Top-Level Schematic

To summarize, you will be designing and building the following modules (with sub-components):

1. One-pulse-per-cycle FSM
2. *Datapath* controller FSM
3. Pong datapath
    a. Paddle module
    b. Collision detection module
    c. Ball position module
    d. T flip-flop (if no suitable library version exists)
4. Clock Generator (using IP Core Generator)

You will be provided with the following module to use with your code:

1. Video (VGA) subsystem wrapper

The top-level schematic for this design is shown in Figure 9. In order to use our design with the VGA port on the FPGA, we will be interfacing with the Virtex II Pro System's XSGA Module.

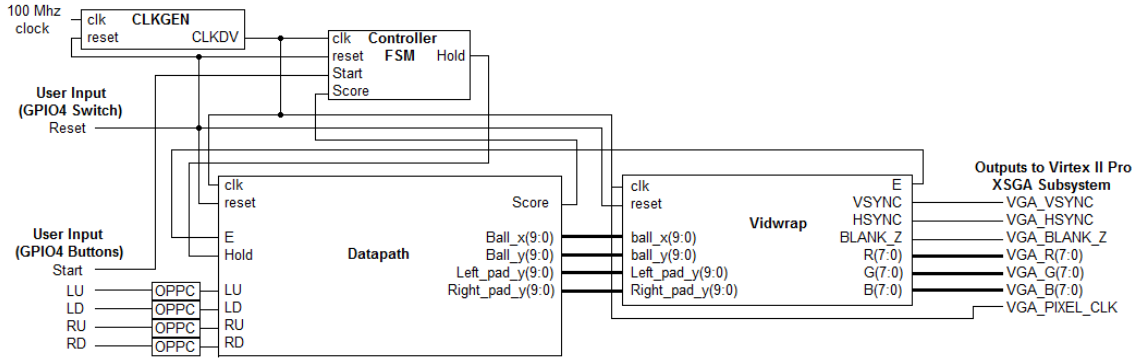

Figure 9: Top-Level Schematic

Notice that the XSGA module on the XC2VP30 requires a connection to the 25 MHz pixel clock, in addition to the RGB, sync, and blank signals.

Pre-Lab Questions

(Done individually)

*NOTE: We will be checking the prelabs **at the beginning of lab.** Any prelab not ready by the beginning of your lab section will not receive credit. All prelabs must be done individually, not in groups.*

1. Draw the state transition diagram (as a Moore machine) for the one-pulse-per-cycle module, using Figure 6 as a guide. This can be done in as few as four states. Recall that you are to assume that E = '0' whenever Di first goes from 0 to 1.
2. Draw the state transition diagram (as a Moore machine) for the control FSM.

For the next questions, recall that the (X,Y) coordinates of each entity refer to the ***top left corner pixel***. The widths of each entity are given in Table 2.

Entity	Height (pixels)	Width (pixels)
Ball	24	24
Paddle	64	24

The ***X-coordinate of the left and right paddles*** are 72 and 592, respectively. The functional vertical "borders" of the game region (against which the ball will collide) are Y = 24 and Y = 456.

In order to ensure that entities never overlap visually, we would like collision to be detected before any objects actually overlap (see Figure 10 for an example). The way we will do this is to check two constraints:

1. Y-coordinate constraints:
    a. Is the lowest pixel of the ball below the highest pixel of the paddle?
    b. Is the highest pixel of the ball above the lowest pixel of the paddle?

2. X-coordinate constraints:
    a. Are the paddle and ball next to each other, as shown in Figure 10?

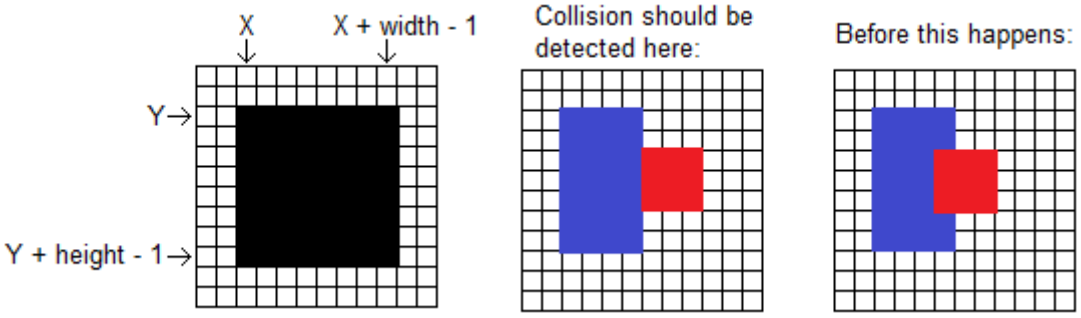

Figure 10: Examples of Entity Collision

In terms of the BallX, BallY, PadLeftY, and the constants above:

3. Give a logical expression, using addition, subtraction, and comparisons (<, >=, =, etc.) for whether the ball has collided with the left paddle.

4. Using the same operations, give an expression for the **Score** output from the collision module. The **Score** output will be true whenever the ball has passed either paddle. For our purposes, we will consider the ball to have passed either paddle if ***any*** pixel of the ball occupies ***any*** row that the paddle is in.

As an example, the following will serve as the logical expression for the **Cy** output from the collision module:

```
Cy = (BallX < TopBorder) or (BallX + BallHeight >= BottomBorder)
```

Where `TopBorder` = 24, `BallHeight` = 24, and `BottomBorder` = 456.

The following final question is a challenging conceptual question, and while you should give it some thought, you are not required to answer it.

5. Can we guarantee, for this system, that the ball *will* occupy the space shown in Figure 10, that leaves no pixels between the ball and the paddle?

IMPORTANT: READ BEFORE YOU BEGIN

- Use **C:\user** (appears as *My Documents*) as your working directory
  - Do **NOT** use your ENIAC (S:) drive or a flash drive
  - Do **NOT** use C:\users (files **will be deleted** at logoff)
- Save regularly
- At the end of a work session, archive your project from Xilinx (Project → Archive) and save this .zip archive to YOUR ENIAC (S:) drive or a flash drive.

- This is a **two-week** lab on the schedule. However, because it is the last week of the semester, to ensure all groups receive the same amount of time to complete the lab, you will be required to demo by the following times (by section):

  - Monday: 4/16 at 4:00 PM
  - Tuesday: 4/17 at 6:00 PM
  - Friday: 4/20 at 4:00 PM

  Groups who do not meet this deadline will not receive credit for the Lab 8 demo.

- It is **very important** that you follow the instructions **carefully** in order to avoid as many issues as possible.

In-Lab Assignment

Part 1: Build the FSMs

First, you will implement the two simple finite state machines you designed as part of the pre-lab: the one-pulse-per-cycle FSM and the controller FSM.

1. Create a new project and give it the name **Lab8**. Place the project in the **C:\user\YourPennKey** folder.
2. Using VHDL, implement the Controller FSM following the description above.
3. Conduct a simulation of the Controller FSM.
4. Using VHDL, implement the one-pulse-per-cycle (OPPC) FSM.
5. Conduct a simulation of the OPPC FSM module.
6. Create schematic symbols for both FSM modules.

Part 2: The *Pong* Datapath

The next part of the lab involves developing the components of the *Pong* datapath. The majority of this part will be done in VHDL. It is strongly recommended that you review Lab 6 for instructions on how to use VHDL processes.

7. Either search the Xilinx schematic library for a suitable T flip-flop module (rising edge-triggered), or implement your own in VHDL.
8. Simulate the T flip-flop (even if you are using a library version – put a wrapper around it and test it).
9. Create the module to implement the Paddle, following the description above.
10. Simulate the paddle module.
11. Create the module to implement the Ball position module, following the description above.
12. Simulate the ball position module.
13. Using the results of the pre-lab, design the Collisions module (remember to add logic to the ***Cx*** expression for collisions with the right paddle).
14. Combine the datapath elements together (along with the OPPC module), following Figure 7 as a guide.
15. Simulate the datapath (as best you can with a waveform).

Part 3: The *Pong* Game

Finally, you're going to combine all of this together and program it on the board.

16. Download the **Video Wrapper Module** (remember, click on the link, right-click on the resulting page, and click "Save Page As" to save the VHDL file to the computer; **DO NOT COPY AND PASTE CODE INTO YOUR OWN MODULE**).

17. Combine all of the modules designed so far into a top level module, following Figure 9 as a guide. Consult the Lab 6 documentation for instructions on how to create the clock generation module using the IP core generator.

18. Create pin assignments for your top level design. Use the 5 push buttons on the peripheral board for the Enter, LU, LD, RU, and RD inputs, and use a switch for the reset signal. Consult the Virtex II Pro manual for the VGA (XSGA) pin assignments.

19. Generate a programming file.

20. Connect the VGA port on the Virtex II Pro ***main board*** (not the peripheral board) to a monitor via VGA cable and program the FPGA using iMPACT.

21. Give a demo to the TA.