Lab Report
Digital Electronics Lab Course
FPGA Pong Game Project

Advanced Lab Course (FOPRA)
Technical University of Munich

Aloysius Farrel 03758167
Bonifasius Auriel Putranto 03809183

17 August 2025

Contents

1 Introduction

Pong is a two-player game that simulates a table-tennis game. Each player has control over a paddle that can move upwards and downwards. The paddle can hit the ball and toss it in the other direction. When the ball goes over the paddle, the other player gets a point. fig. 1 shows a typical pong game.

Within this project, we utilized almost all basic functionalities of the FPGA that we learned in this course, such as modules, operators, cases, if-else statements, always blocks, clocks, counters, button inputs, hex displays, and VGA monitor output.

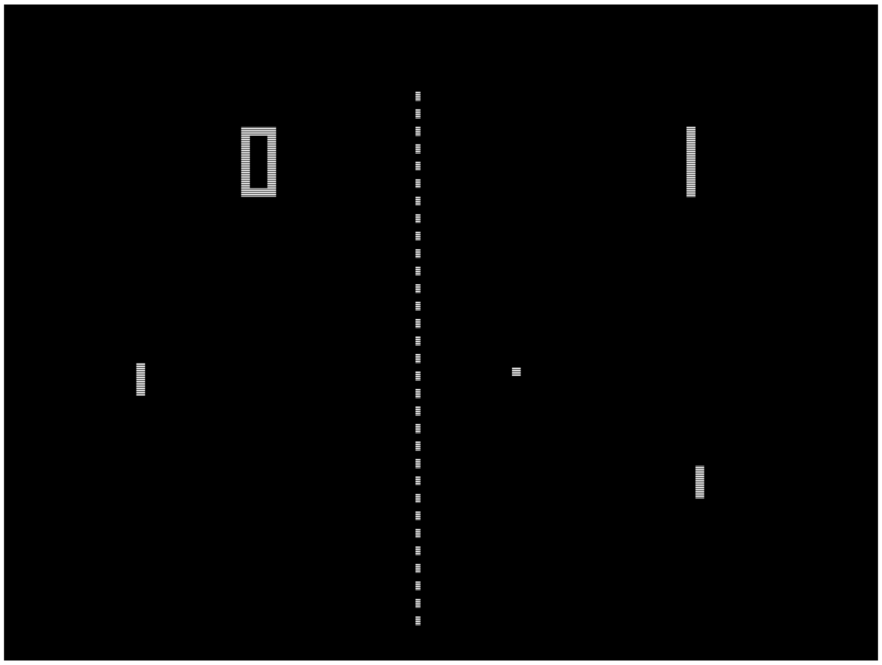

Figure 1: Pong from the Atari Arcade Hits 1 software released in 1972 by Atari interactive [1].

2 Gameplay

The controls for the paddles are the four buttons on the FPGA board. The two right-most buttons control the upward and downward movement of the right paddle, and the two left-most buttons control the left paddle (see fig. 2).

The game starts only when both players press at least one of their control buttons simultaneously. Once the game starts, players can move their paddle to hit the ball and redirect it to their opponent. Each time the ball hits a paddle, its velocity and the velocity of the paddles will be increased. A player scores when the ball passes by the opponent's paddle.

When a player scores, after a delay of 0.5 seconds, the ball's position is reset to a random point along the middle line. The magnitude of the ball's velocity and that of the paddles are also reset. The initial direction of the ball is randomly chosen.

A pause and reset switch are also available. If the pause switch is turned on, the game will pause, and only continue once the switch is turned off again. If the reset switch is turned on, the entire game will reset, including the scores.

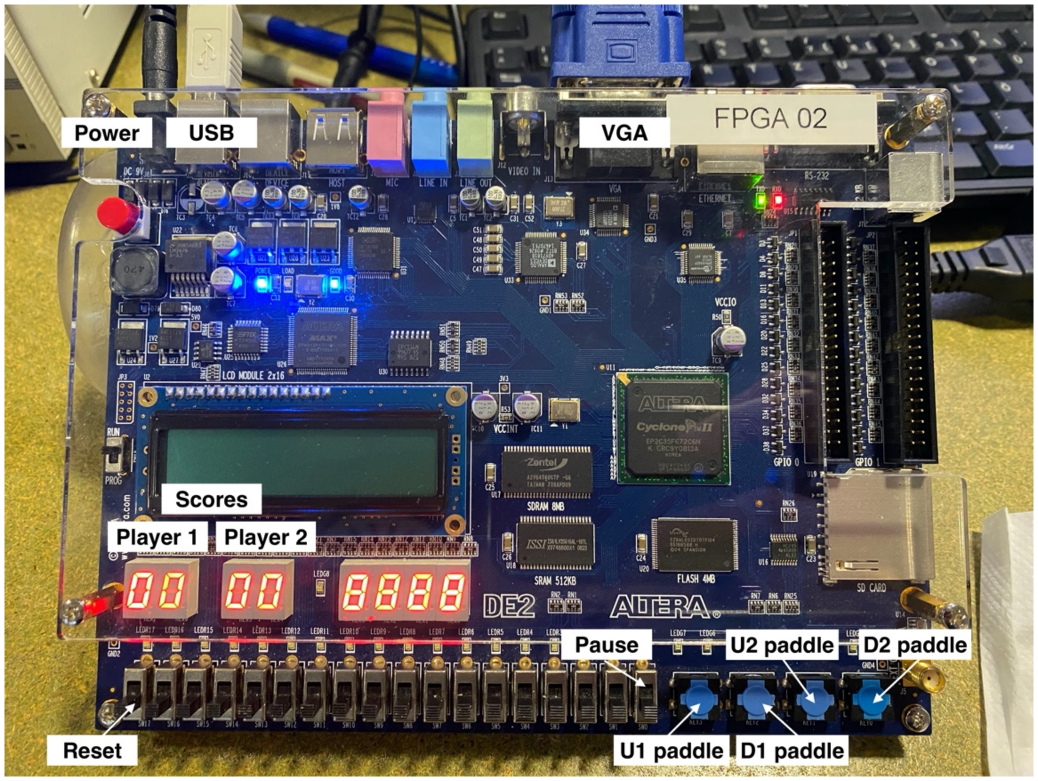

Figure 2: FPGA for this project with the corresponding buttons for the game.

3 How does it work?

3.1 Main Module

In this game, we used four key buttons as two controllers. We used the module `debouncer.v` [2] from the lab course to remove the "bouncing" effects of these buttons. Two switches are also used for the reset and pause switches. `CLOCK_50` is also inserted as input, which then will be converted into 25 MHz and 100 Hz clocks using counters. As output, we used the VGA display for the gameplay and four HEX displays to show scores for each player. These will be further discussed in the next subsections.

In the Figure 3, we can see a simplified schematic of the main module. The input is sent to the logic and will be computed in the state machine with a clock of 100Hz as a refresh rate. Then, the results of the computation, e.g., position of the ball, paddles, and scores, are converted to pixels and binary numbers. The pixels will then be shown by a VGA display, and the binary numbers will be shown by a HEX display.

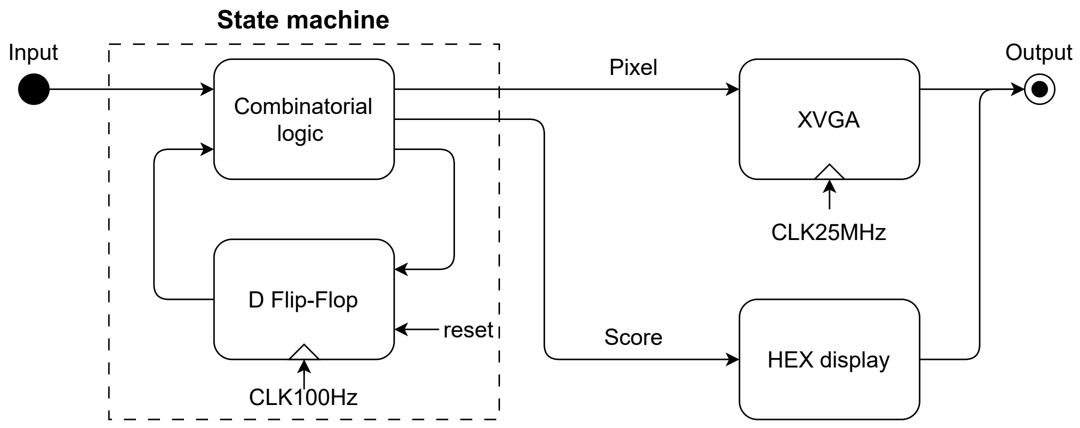

Figure 3: Simplified schematic of the main module consists of input, state machine, VGA display, HEX display, and output.

3.2 Finite State Machine

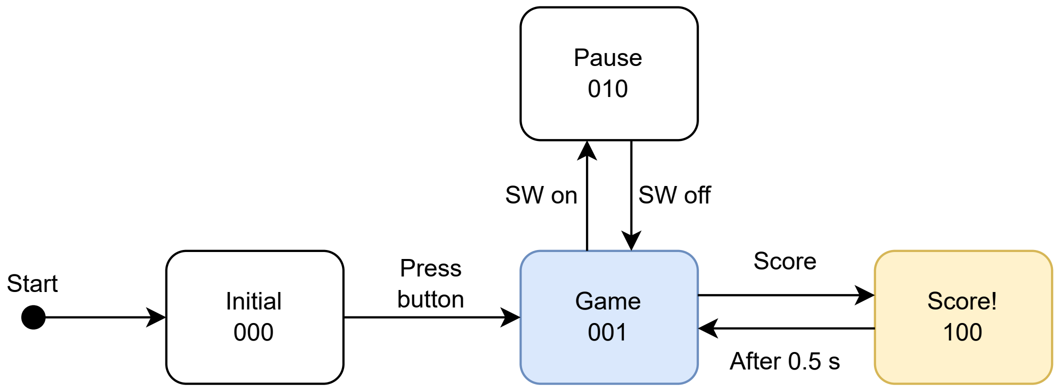

Figure 4: State machine for the gameplay. We start from the initial state (000) to the game state (001). The game can be paused (010) by turning on the switch. If a player scores, it will be move to the score state (100) and back to the game state after 0.5 s.

In the gameplay, we defined four possible states for the state machine: (000) initial state, (001) game state, (010) pause state, and (100) score state. The state machine's combinatorial logic is controlled by a clock of 100 Hz. The clock is chosen so that the movement of the ball and the paddle can be seen. Figure 4 shows the schematic of the state machine.

Once the code is embedded in the FPGA or if one switches the reset, we will start in the initial state (000). The scores are reset to 0, the paddles' position, the ball's position, and speed are reset to the initial value (defined by the parameter).

After both players press one of their control buttons at the same time, the game starts (001). The ball starts moving towards a random direction, which is generated by a random number module (see section 3.6). In this state, players can move the paddle using the control buttons at a specified speed. The position of the ball and paddles is refreshed with the clock 100Hz. The speed of the ball and the paddles is initialized at 1 pixel and 3 pixels every 0.1 ms, respectively. This state continues until one of the players hits a score.

During the game, one can also switch the pause switch and pause the current game (010). In this state, the position of the ball and the paddles freezes. If it is switched off, then the game will continue.

If a player scores, the game switches to the score state (100). In this state, the magnitude of the ball's and paddle's velocity is reset. Then, the position and the direction of the velocity of the ball in the next game are randomized using the random number module. After 0.5 s (counted using *counter_score*), the game switches back to the game state with initialized ball position and speed.

3.3 Collision Detection

The conditions for the collisions between the sprites (hitboxes) are defined using the wires: `ball_hits_paddle_1`, `ball_hits_paddle_2`, and `ball_hits_boundary` which take boolean values.

As an example, when the ball hits paddle 1, its coordinates are now within the hitbox of paddle 1, as seen below:

```
wire ball_hits_paddle_1 = (ball_x <= paddle_1_x + paddle_size_x ) &&
                          (ball_y >= paddle_1_y) &&
                          (ball_y <= paddle_1_y + paddle_size_y);
```

This causes `ball_hits_paddle_1` to return `True`. The sign of the x-velocity of the ball is then flipped, causing the ball to bounce (when the ball hits the boundary, the y-velocity is flipped).

A hit counter was added to fix a bug where the ball got stuck bouncing back and forth on the edge of hitboxes due to the collision detection clock flipping the velocity faster than the ball can move out of the collision area.

The hit counter starts with a value of 0, and is set to 1 when the ball collides with another object. While hit counter is 1, the ball has no collision, i.e. its velocity will not be flipped, even if it is within the hitbox of another object. This prevents repeated sign-flipping of the velocity while the ball is moving out of the collision area.

Once all collision wires are back to `False`, the ball is out of the collision area, and the hit counter is set back to 0, allowing collision to happen as normal.

3.4 HEX Display

To display the game scores, we use four HEX displays. The four displays are split into two pairs of two, with each pair showing the first and second digit of the score of a player. The current scores are converted digitwise (0-9 for each digit) to binary using the `hexdisplay.v` module. These binary numbers are sent as output to the HEX displays.

```
module hexdisplay (in_HEX, out_SEG);
input [3:0] in_HEX;
output reg [6:0] out_SEG;
always @ (in_HEX)
    case (in_HEX)
        4'h0: out_SEG <= 7'b1000000;
        4'h1: out_SEG <= 7'b1111001;
        4'h2: out_SEG <= 7'b0100100;
        4'h3: out_SEG <= 7'b0110000;
        4'h4: out_SEG <= 7'b0011001;
        4'h5: out_SEG <= 7'b0010010;
        4'h6: out_SEG <= 7'b0000010;
        4'h7: out_SEG <= 7'b1111000;
        4'h8: out_SEG <= 7'b0000000;
        4'h9: out_SEG <= 7'b0011000;
    endcase
endmodule
```

Listing 1: hexdisplay.v used to convert number in binary to output binary for the HEX display.

3.5 VGA Display

The VGA display was programmed with the `xvga.v` module from the lab. The display resolution was 640x480 and the VGA module was driven with a 25 MHz clock.

All the sprites in the game were defined by a wire for each sprite: `ball_on, paddle_1, paddle_2, middle_line`. Each of these wires take on a boolean value based on whether the x and y-coordinates of the VGA display are within the boundaries of the sprites. As an example, the wire to define the ball sprite is shown below:

```
wire ball_on =  (x >= ball_x) && (x < ball_x + BALL_SIZE) &&
                (y >= ball_y-disp_shift) && (y < ball_y + BALL_SIZE-disp_shift);
```

The wire `ball_on` defines the boundaries of the ball as a square with size `BALL_SIZE`. Within this boundary, `ball_on` returns `True` and otherwise `False`.

The parameter `disp_shift` is used to shift the hitboxes (the hit conditions) of all objects below their sprites. This was done due to an overflow error, where if the hitboxes of the paddles or ball hit the upper border, where $y = 0$, at speed, the collision logic sometimes does not update fast enough to stop their movement, resulting in the hitboxes moving into "negative" y and causing an overflow, teleporting the hitboxes to the largest possible y-coordinate.

The solution was to shift all the hitboxes down by a constant `disp_shift`, such that we can define the upper collision border at $y =$`disp_shift`, and not at $y = 0$, preventing the overflow error in case the hitboxes hit the upper border at speed. Meanwhile, the sprites remain unshifted (i.e. starting at $y = 0$), such that the game plays out normally within the display bounds.

The sprite display was then programmed using a conditional assignment of `VGA_R, VGA_G, VGA_B` using the sprite display wires as the conditions and updated by the 25 MHz clock, as shown in the following code snippet:

```
always @ (posedge clk25) begin
  VGA_R <= (~blank && (ball_on || paddle_1 || paddle_2 || middle_line)) ? R : 10'
    b0;

  VGA_G <= (~blank && (ball_on || paddle_1 || paddle_2 || middle_line)) ? G : 10'
    b0;

  VGA_B <= (~blank && (ball_on || paddle_1 || paddle_2 || middle_line)) ? B : 10'
    b0;
end
```

The ∼`blank` condition ensures that all three channels are set to 0 outside of the active display area. Within the active area and within at least one of the sprite boundaries defined by the wires `ball_on, paddle_1, paddle_2, and middle_line`, all three channels are set to their corresponding values with the variables `R, G`, and `B`. The three variables initially start with their maximum value (white). Each time the ball hits a paddle, a random number, taken from `random_number.v`, is added to the color values. This creates a random sequence of colours that the sprites use throughout the game. Outside of these sprite boundaries, the channels are all set to 0 again, resulting in a black background.

3.6 Random Number Module

The "random" number used to initialize the position and direction of the ball makes use of a continuously running counter. The module always outputs the value of the counter, which runs until it reaches a limit defined in the input, where it then resets.

Since the time between goals are indeterminate, the value of the counter at the time when the ball position and velocity are reset is unpredictable, creating a pseudo-random initial position and velocity.

```
module random_number(
  input clk,
  input [9:0] limit,
  output reg [9:0] number
);
integer counter=1;

always @ (posedge clk)
    begin
      counter = counter +1;
      if (counter > limit) counter=1;
      number = counter;
    end
endmodule
```

Listing 2: random_number.v used to generate a random number between 1 and a maximum number *limit* to set a random initial position of the ball.

4 Bugs

A bug that wasn't able to be fixed was that for faster velocities, there is a chance that the collision between the paddle and the ball doesn't register correctly, and the ball is able to phase through a paddle. Also, at higher paddle speeds, it is also possible for the paddle to disappear when it hits the upper edge of the screen.

These two bugs are most likely related, a possible cause is that at higher speeds, the collision between the ball and paddle or between the paddle and the screen edge does not have time to be registered, because the clock speed is too slow.

In the case of the ball, this means that the ball has already passed through the paddle before its velocity is reversed, hence why it phases through. For the paddle, this means that the paddle is able to pass through the screen edge and cause its position to overflow (because it can't handle negative y-positions), teleporting the paddle to the largest y-coordinate.

A Source code

The source code is available at `https://github.com/aloysf/Project_Pong_VHDL`.

B Bibliography

[1] "Pong." (2017), [Online]. Available: `https://de.wikipedia.org/wiki/Pong`.

[2] P. Schmakat, M. Kleinhans, and A. Wendl, "Digital logic: Introduction to fpga circuit development," *TUM*, 2022.