Lab 4 Report

UCLA Samueli School of Engineering
CS M152A Spring ‘24
Lab 5 TA Chenda Duan

Group 6

Tae Hwan Kim	506043010
Brad Lowe	905581956

Group 9

Zeckria Kamrany	405749185
Simon Traub	405796676

1 Introduction and requirement

In this final project, our group proposes to recreate the popular game “Flappy Bird” using an FPGA for control and VGA for graphics output. This game is recognized for its simple gameplay mechanics, where the player controls a bird attempting to navigate through a series of obstacles without touching them. The project involves digital logic design, programming in Verilog, FPGA board utilization, and the practical application of VGA graphics.

2 Design Requirements

2.1 Bird (Player)

The player was represented by a square on the screen. The player has the ability to jump. Without jumping, the player will simply float to the ground because of gravity. The player spawns at a specific location every time upon resetting the game. The player cannot go above or below the screen.

2.2 Obstacle Generator

There is a vertical green bar that has an opening that is roughly in the middle of the screen. The vertical bar moves at a constant rate to the left. If the player collides with the vertical bar, the player will die and the game will end.

2.3 Score Management

The score displays on the FPGA using the 7-segment display. Your score increases by 1 for every second you are alive. The score will be reset to 0 if you reset the game. The score will pause once the player dies (before you hit reset).

2.4 Reset

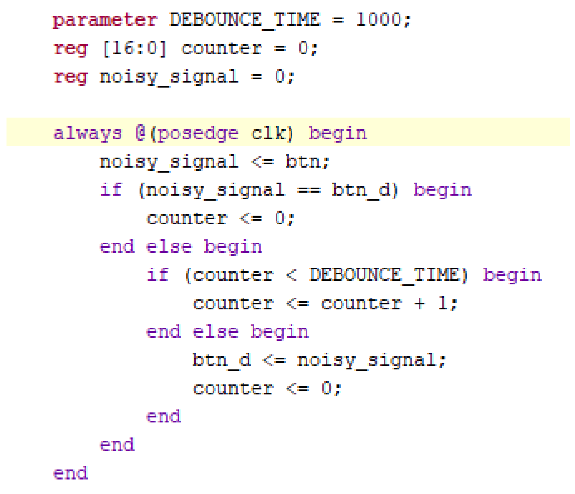
The reset button resets the game, which results in the player moving back to the original starting position. The vertical bar is also reset and will move from the far right of the screen once more. The score is also reset back to 0.

2.5 Graphic Display

The player is shown with a yellow square. The obstacle consists of a vertical green bar with an opening that is roughly in the middle of the screen.

3 Design and Implementation

3.1 Debouncer



```
parameter DEBOUNCE_TIME = 1000;
reg [16:0] counter = 0;
reg noisy_signal = 0;

always @(posedge clk) begin
    noisy_signal <= btn;
    if (noisy_signal == btn_d) begin
        counter <= 0;
    end else begin
        if (counter < DEBOUNCE_TIME) begin
            counter <= counter + 1;
        end else begin
            btn_d <= noisy_signal;
            counter <= 0;
        end
    end
end
```

Figure 3.1: Verilog for debouncer module

The debouncer module takes the raw signals from the RESET and JUMP buttons and the 500 Hz clock and outputs a refined signal to be used by other modules. This uses a much lower sampling frequency than the main clock to ensure that one press is not registered as multiple presses, and to create metastability between the buttons and switches. The refined signals are treated as the actual RESET and JUMP signals for the rest of the modules.

3.2 Clock Divider

```
always @(posedge clk) begin
    if (counter_10hz >= DIVIDE_FACTOR_10HZ - 1) begin
        clk_10hz <= ~clk_10hz;
        counter_10hz <= 0;
    end else begin
        counter_10hz <= counter_10hz + 1;
    end

    if (counter_500hz >= DIVIDE_FACTOR_500HZ - 1) begin
        clk_500hz <= ~clk_500hz;
        counter_500hz <= 0;
    end else begin
        counter_500hz <= counter_500hz + 1;
    end

    if (counter_1hz >= DIVIDE_FACTOR_1HZ - 1) begin
        clk_1hz <= ~clk_1hz;
        counter_1hz <= 0;
    end else begin
        counter_1hz <= counter_1hz + 1;
    end
```

Figure 3.2: Verilog for clock divider module

The clock divider block takes in the master clock and outputs new clock signals to use in other modules: a 1 Hz clock for updating the score value, a 10 Hz clock for the player movement including jump and gravity, a 500 Hz clock for updating the seven-segment display, which is necessary because only one of the anodes can be on at a time. The faster clock is also used for the purposes of debouncing, so we can refine the signal from the buttons.

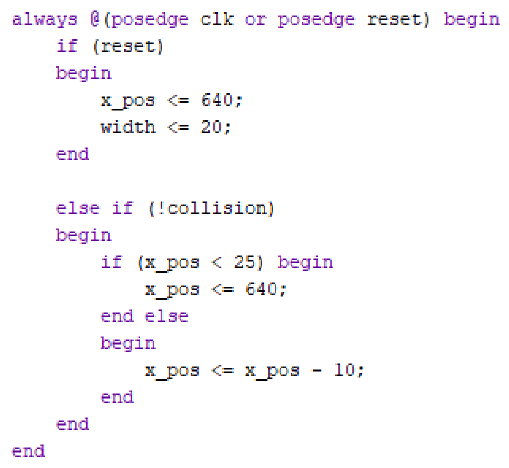
3.3 Player

```
always @(posedge clk_10hz or posedge reset) begin
    if (reset) begin
        y_pos <= initial_y; // or any initial value you need
    end else if (!collision) begin
        if (jump) begin
           if (y_pos >= 50)
           begin
               y_pos <= y_pos - 20;
           end

           else
           begin
                y_pos <= 30;
           end
        end

        else begin
            y_pos <= y_pos + 3;

            if (y_pos >= 470)
            begin
                y_pos <= 470;
            end
        end
    end
end
```

Figure 3.3: Verilog for player module

The player module takes in a 10 Hz clock, the jump, reset, and collision signals, and a gravity, jump force, and initial y position constants and outputs the current y position of the player. If the player is not jumping, then it slowly falls downwards. However, if the jump button is pressed then the player will shift upwards every frame as long as the button is held. The jump force represents the number of pixels by which this shift occurs. If the player touches the ground or ceiling, the downwards or upwards movement will stop, respectively. If a collision occurs, then the player stops moving altogether.

3.4 Obstacle

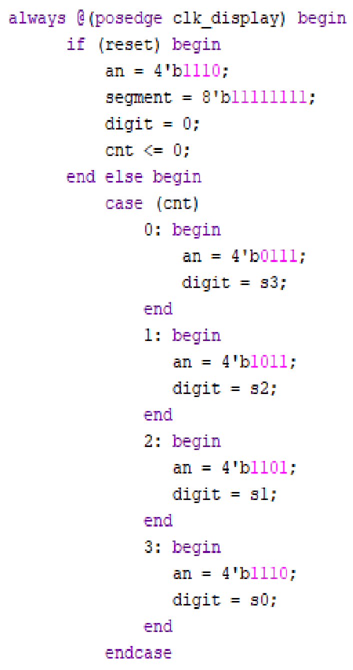


```
always @(posedge clk or posedge reset) begin
    if (reset)
    begin
        x_pos <= 640;
        width <= 20;
    end

    else if (!collision)
    begin
        if (x_pos < 25) begin
            x_pos <= 640;
        end else
        begin
            x_pos <= x_pos - 10;
        end
    end
end
```

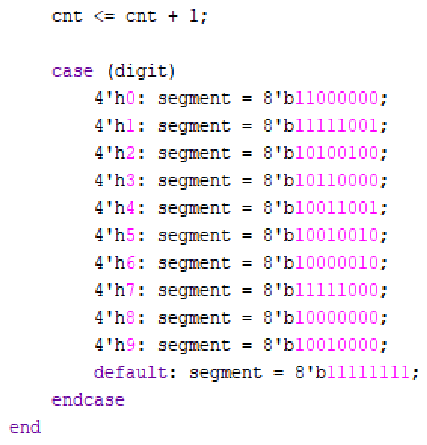
Figure 3.4: Verilog for obstacle module

In the obstacle module, we implement the bars that move across the screen that act as the obstacles for the player to avoid. This module takes in a 10 Hz clock, the reset and collision signals, and a speed constant that was defined in game_controller, and outputs an x position and width. The speed determines how quickly the x position is changed, and this position is updated every “frame” of the 10 Hz clock. Once the bar moves all the way to the left side of the screen, it moves back to the right so we can reuse the same bar multiple times.

3.5 Score Display



```
always @(posedge clk_display) begin
    if (reset) begin
        an = 4'b1110;
        segment = 8'b11111111;
        digit = 0;
        cnt <= 0;
    end else begin
        case (cnt)
            0: begin
                an = 4'b0111;
                digit = s3;
            end
            1: begin
               an = 4'b1011;
               digit = s2;
            end
            2: begin
               an = 4'b1101;
               digit = s1;
            end
            3: begin
               an = 4'b1110;
               digit = s0;
            end
        endcase
```

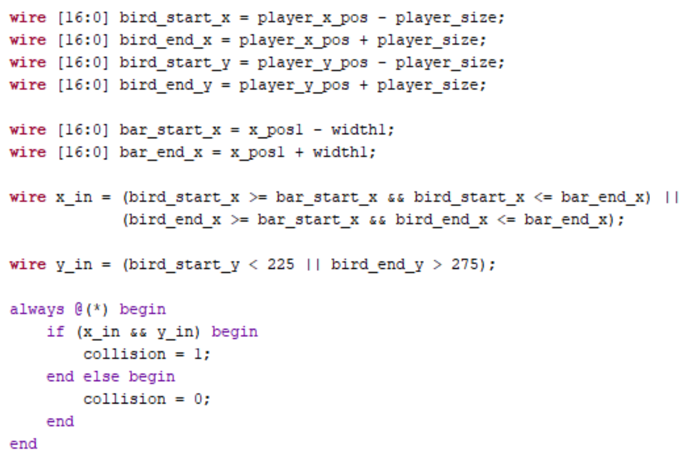


```
        cnt <= cnt + 1;

        case (digit)
            4'h0: segment = 8'b11000000;
            4'h1: segment = 8'b11111001;
            4'h2: segment = 8'b10100100;
            4'h3: segment = 8'b10110000;
            4'h4: segment = 8'b10011001;
            4'h5: segment = 8'b10010010;
            4'h6: segment = 8'b10000010;
            4'h7: segment = 8'b11111000;
            4'h8: segment = 8'b10000000;
            4'h9: segment = 8'b10010000;
            default: segment = 8'b11111111;
        endcase
    end
```

Figures 3.5, 3.6: Verilog for score display module

The score display module is designed to display a numerical score on a 7-segment display. To handle scores ranging from 0 to 9999, the module takes an input score and caps this score at 9999 for display purposes. This module uses two separate clocks; ‘clk’ for updating the score and ‘clk_display’ for controlling the display refresh rate. This module takes a reset input to reset the score and display to all zeros when the game starts over.

3.6 Collision Checker

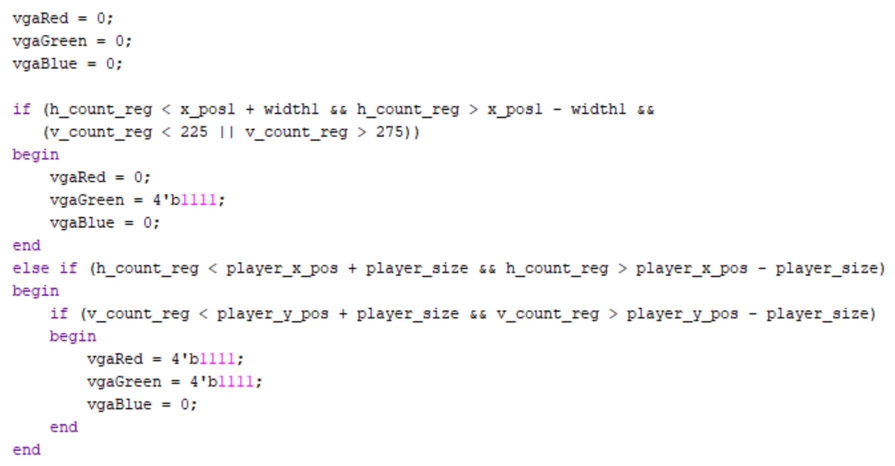


```
wire [16:0] bird_start_x = player_x_pos - player_size;
wire [16:0] bird_end_x = player_x_pos + player_size;
wire [16:0] bird_start_y = player_y_pos - player_size;
wire [16:0] bird_end_y = player_y_pos + player_size;

wire [16:0] bar_start_x = x_pos1 - width1;
wire [16:0] bar_end_x = x_pos1 + width1;

wire x_in = (bird_start_x >= bar_start_x && bird_start_x <= bar_end_x) ||
            (bird_end_x >= bar_start_x && bird_end_x <= bar_end_x);

wire y_in = (bird_start_y < 225 || bird_end_y > 275);

always @(*) begin
    if (x_in && y_in) begin
        collision = 1;
    end else begin
        collision = 0;
    end
end
```

Figure 3.7: Verilog for collision_checker module

The collision checker module checks whether the player's position is inside a specific box, which represents the position where we want to display the obstacle. The player's horizontal position ranges from start_x to end_x, and the vertical position ranges from start_y to end_y. These are calculated based on the player's position and width. If the player's position enters the collision box, the module outputs a 1, indicating that the player has collided with the obstacle. If not, it always outputs a 0.

3.7 VGA Display

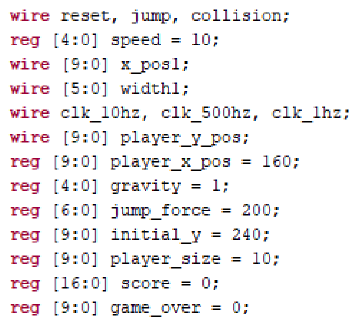


```
vgaRed = 0;
vgaGreen = 0;
vgaBlue = 0;

if (h_count_reg < x_pos1 + width1 && h_count_reg > x_pos1 - width1 &&
   (v_count_reg < 225 || v_count_reg > 275))
begin
    vgaRed = 0;
    vgaGreen = 4'b1111;
    vgaBlue = 0;
end
else if (h_count_reg < player_x_pos + player_size && h_count_reg > player_x_pos - player_size)
begin
    if (v_count_reg < player_y_pos + player_size && v_count_reg > player_y_pos - player_size)
    begin
        vgaRed = 4'b1111;
        vgaGreen = 4'b1111;
        vgaBlue = 0;
    end
end
```

Figure 3.8: Verilog for edited section of vga_sync module

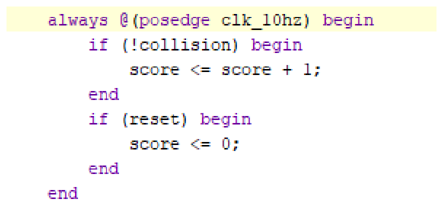
The VGA module uses the player's position and size, as well as the obstacle's position, to display graphics for these objects. By limiting the vertical and horizontal counts to between the position plus width and position minus width, we have implemented a way to display the player and obstacle graphics correctly on the VGA screen. In addition, we have introduced a small gap in the obstacle for the player to jump through by adjusting the offset of the obstacle. This is why green is only displayed if v_count_reg is less than 225 or greater than 275.

3.8 Game Controller



```
wire reset, jump, collision;
reg [4:0] speed = 10;
wire [9:0] x_pos1;
wire [5:0] width1;
wire clk_10hz, clk_500hz, clk_1hz;
wire [9:0] player_y_pos;
reg [9:0] player_x_pos = 160;
reg [4:0] gravity = 1;
reg [6:0] jump_force = 200;
reg [9:0] initial_y = 240;
reg [9:0] player_size = 10;
reg [16:0] score = 0;
reg [9:0] game_over = 0;
```

Figure 3.9: Definition of game-wide constants in game_controller



```
always @(posedge clk_10hz) begin
    if (!collision) begin
        score <= score + 1;
    end
    if (reset) begin
        score <= 0;
    end
end
```

Figure 3.10: Adjusting score based on clocks

The game controller module combines all the modules we've discussed above. The clock divider takes the master clock and outputs divided clocks. Two debouncers are instantiated for the jump and reset buttons. Both the obstacle and player modules take a 10 Hz clock from the clock divider, and they output their position and width to the VGA sync module. The check collision module determines if the player bumps into the obstacle by checking if the player's position matches a specific location where a collision would occur. The top module tracks the score, incrementing it by 1 as long as the player does not collide with the obstacle. If the reset button is pressed, the score resets to 0. The score system module takes clocks and outputs proper an and segment to show the number on the board display.

4 Simulation and Testing

We put our project through a rigorous set of simulation and testing to ensure that the implementation was bug-free and in compliance with pertinent regulations. In particular, we performed several visual inspections to make sure that the jumping, collision, and movement was in line with the project spec. We also created a testbench to specifically evaluate whether or not the scoring module and LED display worked correctly, and we are happy to report that the simulation operated exactly in line with expectations.

5 Conclusion

5.1 Design Summary

For this project, we implemented Flappy Bird with an FPGA board and a VGA display. In particular, controls on the FPGA board were used to manipulate the player character, a “bird” (represented by a 20 by 20 pixel square) that must “fly” (jump) through a series of small, 50 pixel holes inside of bars which move from the right to the left of the screen. We also implemented a score feature, in which the number of seconds that the player survived is displayed on the LED display on the FPGA board. A player encounters a “game over” once the bird collides with the bars, and at that point the game stops.

With respect to controls, there are two buttons the player can use. The “jump” button, which allows the bird to move upward for a temporary rate, and the “reset” button, which resets the state of the game. The latter can be used both during gameplay, and after a game over.

Here is a list of key modules we used to implement the project and a brief summary of each:

1. game_controller.v. This is the driver module for the entire project. It calls each of the remaining modules we will describe, and controls the reset register and “score”, a register that holds the number of seconds that the player has survived and will be displayed on the LED display.
2. clock_divider.v. This module controls the clocks used for different aspects of the game. In particular, there is a 1 Hz clock (for controlling the score), a 10 Hz clock (for controlling the frame rate) and a 500 Hz clock (for debouncing).
3. debouncer.v. This module assists with debouncing button presses.
4. obstacle.v. This controls the bar obstacle that the bird must attempt to travel through, including the rate it moves from the left to the right of the screen, the gap inside the bar, and the spawning of new bars.
5. player.v. This controls the player character, the most important feature of which is the jumping mechanics.
6. vga_sync.v. This is a key module because it regulates the display of the bird and the bar, displaying their motion at a rate of 10 frames per second.
7. check_collision.v. This checks for collisions between the bar and the bird. If a collision occurs, the “collision” register is set to 1, and the rest of the modules react by stopping the other modules in place.
8. score_system.v. This handles displaying the score register on the LED display.

5.2 Difficulties

The only difficulties we encountered were dealing with implementing the bird physics, specifically as it pertains to gravity and “flying” (jumping). Initially, we implemented it with quadratic physics (constant acceleration downward). However, as we progressed we realized that this was not only difficult, but also did not produce the fluid motion that we expected due to the constraints of the VGA display. Therefore, we changed our approach such that the bird moves down at a constant rate. When the jump button is pressed on the FPGA, it moves at a constant rate upward for a fixed period of time before returning to its linear trajectory downward. We found that this was not only easier to implement, it also produced more fluid motion for the player.