Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Flappy Bird, playable on an FPGA

Design Procedure:

In this lab we were tasked with implementing a complex program that involved the use of a VGA screen, a form of memory storage, and a form of external input of our choice. We chose to implement a custom version of Flappy Bird, that is able to switch between four different difficulty settings, has personal best scores for each that are saved to a RAM module, displays the current difficulty and both the current score and personal best score on the FPGA itself, outputs a sound when the game is over, and uses an N8 controller as its main form of input.

Essentially, Flappy Bird is an auto-side-scrolling game in which the player controls a "bird" that can move up or down. If the player presses a specific button, which in our implementation is the "B" button on the N8 controller, the bird will fly upwards. If no button is pressed in between frames of the game, the bird will gradually fly downwards until it hits the bottom. At regular intervals, pairs of pipes appear that the player must fly in between: the two pipes are positioned so that there is a gap in between the two of them. If the player flies into a pipe, or flies into the ground, the game is over. If the player is able to fly through a pipe successfully, the score will be incremented by one point. If the player flies into the ceiling, the bird will remain unable to fly through the ceiling; hitting the ceiling does not end the game. This continues until the player eventually collides with either a pipe or the ground, where the player will be asked to reset and try again.

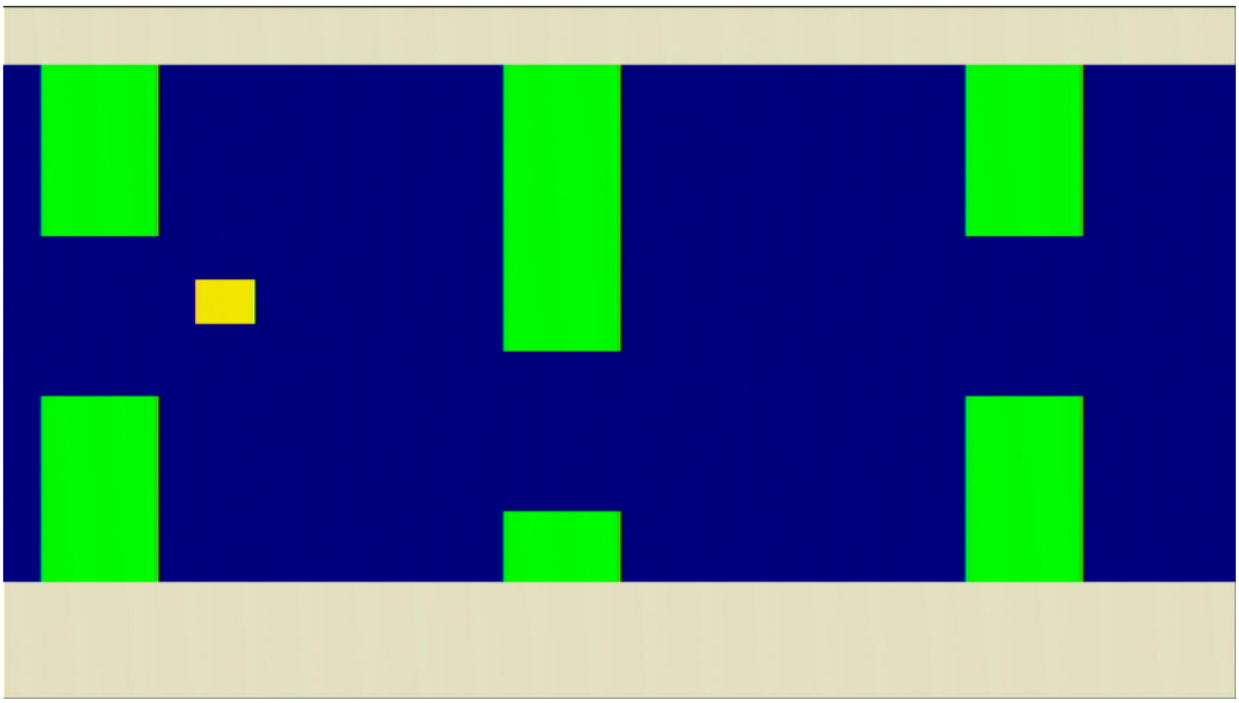
In our version of Flappy Bird, the game initially begins in a "start menu" screen, where the player is able to choose their difficulty using the N8 controller's D-Pad and start the game. The VGA screen will be completely gray while in the start menu, and the game will begin once the player presses the "start" button. Once in a game, the sky color will differ depending on the difficulty setting, and the frequency of pipes will also change. Two clocks are used in the game: the 50 Mhz CLOCK_50 from the FPGA's crystal oscillator and a slower clock 22 orders of magnitude slower than the 50 Mhz clock. The former clock is used to update the VGA screen, calculate complex logic, and find random numbers for the pipe pattern that should come next, while the latter is used to control the game's framerate so that the game's updates in terms of bird and pipe locations are slow enough so it can be playable. If a pipe pair has been avoided by the bird, the score will increment by 1 once the pipe pair despawns from the screen. If the bird collides with either a pipe or the ground, or the A button is pressed, the screen will

turn red except for the bird to signal that the game is over and all game functionality will cease; an audible sound will also play to signal that the game is over. Once at this game over screen, the player must press select to reset the game and begin again.

In terms of inputs, an N8 controller is used as the core “input” for the game. All buttons are used on it for various things: the directional D-Pad is used to choose the difficulty setting, the select button is used to reset the game, the start button is used to start the game, the B button is used to make the bird go upwards, and the A button is used to end the game early.

The outputs for the game are a 640x480 VGA screen and a six-digit HEX display. The VGA screen is used to output the current state of the game, and has differing behavior depending on whether the game is at the start menu, the game itself, or at the game over screen. The HEX displays are used to display both the current score and the personal best for the currently selected difficulty level, with the max score for either being 511.

The following figure shows a screenshot of what the VGA could display while playing the game:



Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Figure 1: A screenshot of Flappy Bird gameplay, while playing at the 4th and hardest difficulty setting.

To be frank, Flappy Bird is a conglomeration of several, much smaller components that all sum together into a complex program. Below are various explanations for how many of these smaller components work within Flappy Bird to eventually create an entire playable game:

Difficulty Selector:

Four difficulties are able to be selected while playing this game: with the difficulties benign assigned clockwise across the directional pad on the N8 controller. The lowest difficulty is associated with the “up” button, the next easiest is “right”, second hardest is “down”, and the hardest is associated with the “left” button. Each difficulty has its own respective sky color and personal best score: the easiest difficulty will be powder blue, second easiest is sky blue, second hardest is cobalt blue, and hardest is navy blue. The initial personal best scores for each difficulty also differ, as for example, the hardest difficulty initially has a personal best of 0, while the easiest has an initial personal best of 4. The following figures show what is shown on the VGA while at the start menu and selecting difficulties:

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Figure 2: What the start menu and initial PBR are at Difficulty 1 (up)

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Figure 3: What the start menu and initial PBR are at Difficulty 2 (right)

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Figure 4: What the start menu and initial PBR are at Difficulty 3 (down)

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Figure 5: What the start menu and initial PBR are at Difficulty 4 (left)

The difficulty can only be changed BEFORE the start of a game: if a game is started by pressing the start button, the difficulty will be locked until the game ends and is reset back to the start menu. Likewise, if the game is over and is at the game over screen, the difficulty will still not be changeable.

Button Input Synchronizers:

Because the N8 controller uses buttons as its primary form of input, it is important to ensure that the button inputs are synced with the FPGA's clock, and that they are only considered to be "active high" inputs on the first clock cycle after activation. So, a flip-flop is used to ensure this behavior. At every clock cycle, the flip flop checks to see if

a button input is active, AND if at the previous clock cycle, the button was off. If this is ever true, the flip-flop outputs a high signal, otherwise it will output a low signal.

Figure 6 shows the SystemVerilog used to do this behavior:

```
module input_ff(clock, in, out);

   //i/o

   input logic clock, in;

   output logic out;

   //create control signal that tracks whether the input was on at the prev clock cycle
   logic was_active;

    always_ff @ (posedge clock) begin
        // Set out high only when in is high and was_active is low
        out <= in && ~was_active;

        // Update was_active to current state of in for the next cycle
        was_active <= in;
    end
endmodule
```

Figure 6: SystemVerilog of how the input flip-flops allow a positive output for a button press only at the first clock cycle of activation.

The A, B, start, and select buttons all make use of this functionality in order to keep them all synced to the FPGA clock and only allow positive input for a single clock cycle. Since the directional buttons are used to determine difficulty and difficulty cannot be changed while playing the game, they do not require synchronization.

Clock Divider:
The clock divider module from EE271 is used to create the slower clock used throughout the Flappy Bird game. Using the 22nd setting, this clock is much, much slower than CLOCK_50 and can be used to control the speed and frequency of what appears on the VGA screen and the changes amongst the locations of the bird and pipes.

However, this is only one component of what is used to implement the slower clock. The slow clock created by the clock divider is not the slow clock used by the FPGA; instead, this clock is fed into a flip-flop that only allows a positive signal to be outputted at a positive edge of the slow clock. Effectively, this makes it so that the slow clock is only

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

active high for a single clock cycle of CLOCK_50, allowing it to act as a regular pulse instead of an oscillating clock.

Game States:

There are three game states while playing Flappy Bird: the start menu, the actual game, and game over screen. The difficulty module already explained the behavior of the start menu, but what if the game is started, what if the game ends, and what if the game is reset? To handle the control signals used to determine which state is currently active, an FSM is used that outputs control signals for the rest of the modules. If the start button is pressed while in the start menu state, the game will transition to the in game state. Likewise, if the game should end due to either a collision or the A button being pressed, the game will transition to the game-over state, where it remains there until the game is reset.

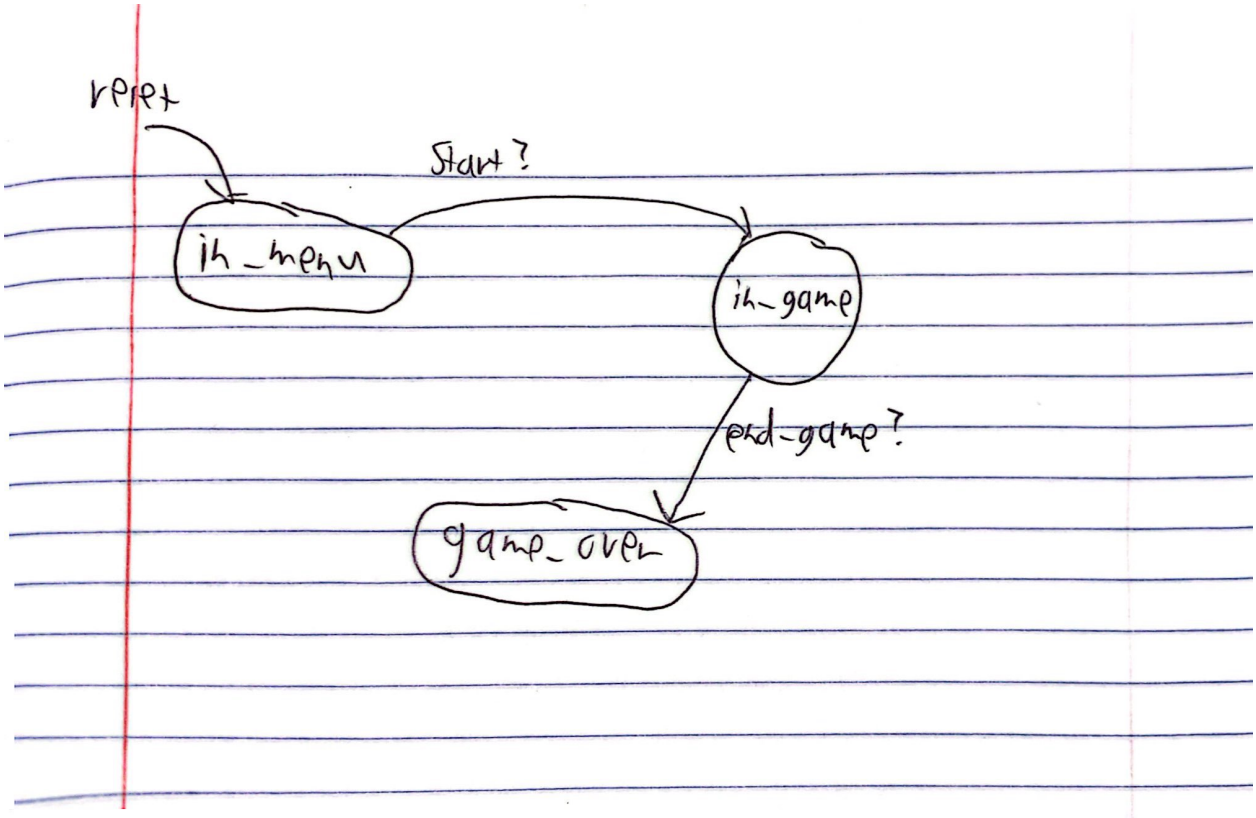
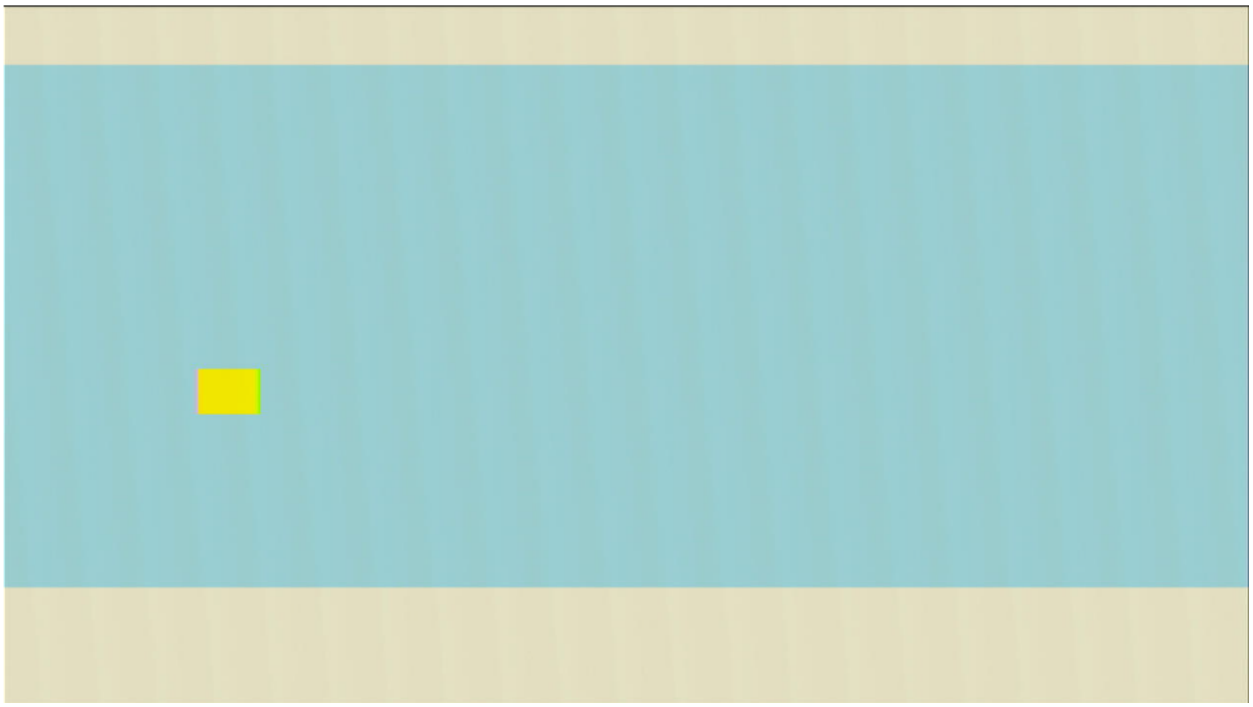


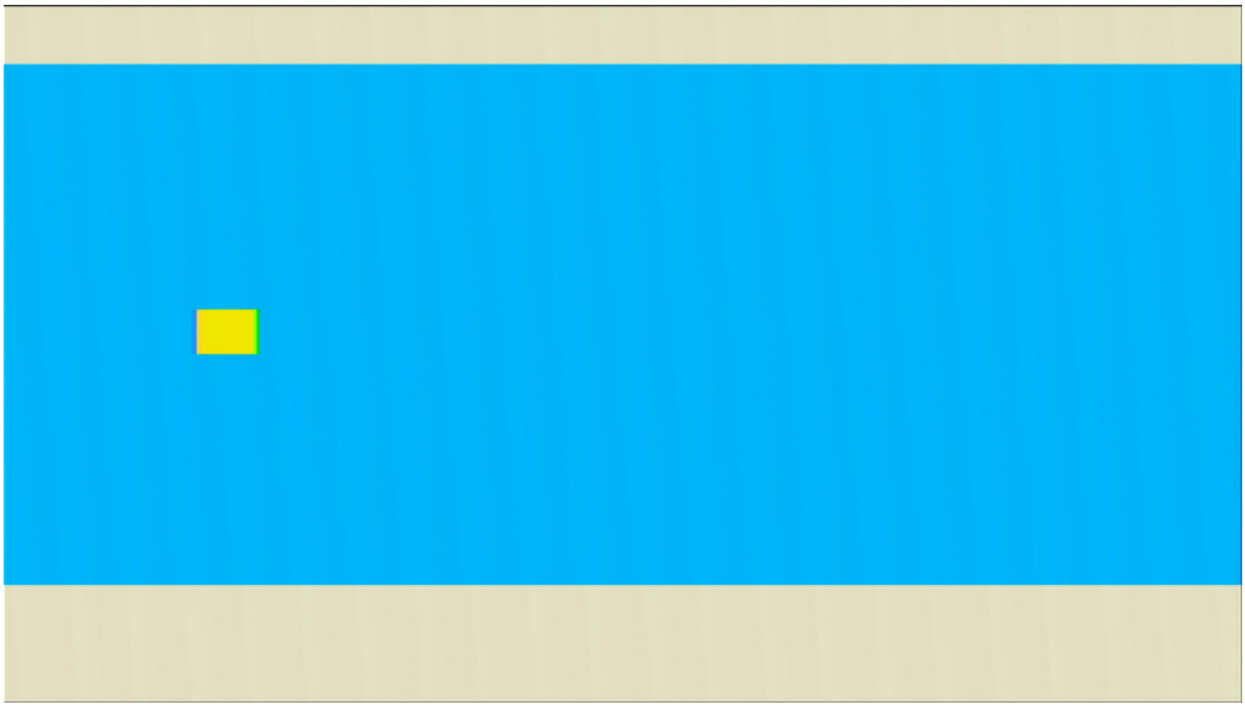
Figure 7: Simple FSM of the Game State module. State names for each state are synonymous with their respective control signals that turn on while in that state.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

In terms of what the game should look upon the pressing of the “start” button, the following figures display the game screen for each difficulty:

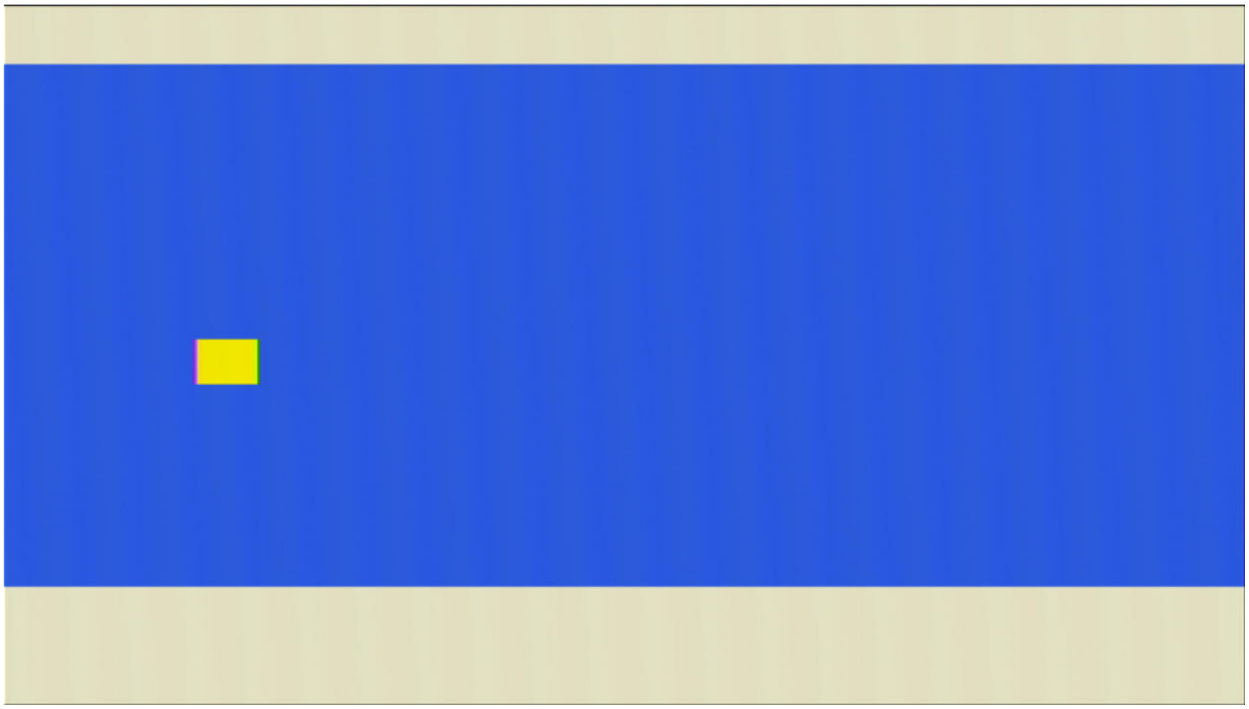


“In-Game, Difficulty 1”



Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

“In-Game, Difficulty 2”



“In-Game, Difficulty 3”

“In-Game, Difficulty 4”

And finally, if the game is over, the VGA screen will change the sky color to red and remove all ground and pipes from the screen, shown below:

"Game Over"

Figures 7A-7E show the VGA screen at various states and difficulty settings. After enough clock cycles, pipes will begin to appear

Score:

The score of the game is handled by a simple counter linked to three of the FPGA's HEX displays. When an input signal "increment" is active, a flip-flop is used to increment the current score being stored in the program by one, with this occurring until the maximum score of 511 is reached where it will then remain at 511. If the reset signal is ever active, the score will be reset back to 0.

The score is then linked to three instances of a "seg7" module, which allow the score to be displayed onto three 7-segment HEX displays as visible digits. HEX2 is used as the hundreds place, HEX1 is used as the tens place, and HEX0 is used as the ones place.

For example, if the current score is "2", HEX2 and HEX1 should both show 0's, while HEX0 should show a 2.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

PBR:
The Flappy Bird game keeps track of all four personal best scores for each difficulty level using a 1-port 4x9 RAM module to store, read, and potentially overwrite personal bests. The difficulty currently selected is used as the address for the RAM, the data in is the current score, the data out is the personal best record (PBR) for the current difficulty selected, and the write enable is a boolean that determines if the current score is greater than or equal to the current PBR; in which case the old PBR is overwritten and becomes the current score.

The current PBR selected will be displayed on the HEX display, directly to the left of the current score of the game. Similar to the score module, HEX5 displays the hundreds place, HEX4 displays the tens place, and HEX3 displays the ones place.

The following two figures show the different possible states of the HEX displays depending on the current PBR and current score:

Figures 7A and 7B: The HEX displays outputting data while playing Flappy Bird, with the left image showing that the PBR remains unchanged while the current score is less than the PBR, and the right image showing that the PBR becomes the current score once the latter is less than or equal to the PBR.

A 1-port 4x9 RAM is used due to there being **four** difficulty states, and the fact that the maximum score, 511, is a **nine** bit number. Since we do not need read and write to be separate, a single address port is used.

A .mif file is used to initialize the PBRs of the game upon startup. Figure 8 shows the contents of this .mif file:

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Addı	+0	+1	+2	+3	ASCII
0	4	1	2	0	....

Figure 8: The contents of personal_bests.mif, the file used to initialize all four difficulties' personal bests.

Game Over Sound:

When the game is over, aka when the current game state is in the game over state, the FPGA will output a constant, droning sound to play alongside the game over screen to encourage the player to reset the game as soon as possible. The sound used for this is a 2-second long, 48000 Hz A3 note that loops over and over as long as it is being read. A 96000x24 ROM module is used to store this note within the FPGA, with the data itself being found in a .mif file named note_A3_two_sec.mif. If the game is currently active, i.e. if the game state is either at the menu or in the game itself, the game over sound is disabled and the ROM cannot be read. However, if the game state is at the game over screen, at every clock cycle of the CLOCK_50 clock the ROM is allowed to be read from and the address currently being read is incremented by 1. The data out of the ROM is then divided by 2 in order to produce our desired, quieter game over sound.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

```
module game_over_sound(clock, reset, write_ready, data);

   //i/o

   //clock is our clock, reset resets data, write_ready determines if we will write at next clock cycle
   input logic clock, reset, write_ready;

   //data is the data outputted from RAM
   //output logic [23:0] data_out_left, data_out_right;
   output logic [23:0] data;

   //addresses are each of the 96K possible locations we can store information in
   logic [16:0] address;

   //logic to store full volume version of sound
   logic [23:0] data_full_volume;

   //now that i/o has been introduced, instantiate our rom
   loss_sound A3 (.address, .clock, .q(data_full_volume));

   //at every clock cycle, increment address by 1 if write is ready unless reset
   //if reset, make address go back to 0
   //if max address is reached, counter should loop back around to 0
   always_ff @ (posedge clock) begin
      if (reset == 1'b1) address <= 17'd0;
      else if (write_ready & ~(address == 17'd95999)) address <= address + 17'd1;
      else if (write_ready & (address == 17'd95999)) address <= 17'd0;
      else address <= address;
   end

   //make output quieter than original by a factor of 2
   assign data = {{1{data_full_volume[23]}}, data_full_volume[23:1]};

endmodule
`timescale 1 ps / 1 ps
```

Figure 9: The implementation of the game_over_sound module, incorporating the 96000x24 bit ROM module named loss_sound.

Bird Input:

Since the game itself runs off of the slower clock and not CLOCK_50, it is imperative that the program is able to keep track of whether the B button was pressed in between positive edges of the slower clock so that the bird can correctly jump up.

To fulfill this task, a clock-domain-crossing is required. Whenever the slow clock is not currently high (every clock cycle of CLOCK_50 except for the positive edge of the divided clock), the flip flop checks to see if the B input has been pressed; and if so, it records that it has been within a separate logic structure. On the next pulse of the slow clock, the flip-flop outputs the result of this logic structure, whether it be high (B was pressed before slow clock pulse) or low (B was not pressed). Not only does this implementation only allow the bird input to be active for a single clock cycle, it also allows information from the CLOCK_50 clock domain to cross to the slower clock domain so that the player can press the B button any time before the slower clock pulse is high and their input will be recognized.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

```
module bird_input(fast_clock, slow_clock, b_in, bird_in, b_in_was_on);

   input logic fast_clock, slow_clock, b_in;

   //both fast_clk and slow_clk are only active for one clk cycle

   output logic bird_in, b_in_was_on;

   //

   //logic to store current state of b_in before next + edge of slowclock
   //logic b_in_was_on;
   always_ff @ (posedge fast_clock) begin
      //check to see if slow_clk is active
      if (~(slow_clock)) begin
         //if not, check if b_in is on
         if (b_in) begin
            //if so, set b_in_was_on to true
            b_in_was_on <= 1'b1;
         end
         //if not, it remains what it was before
      end
      else begin
         //if the slow clock is active, update output to whether b_in became true or not, then set b_in_was_on to off
         bird_in <= b_in_was_on;
         b_in_was_on <= 1'b0;
      end
   end
endmodule
```

Figure 10: The implementation of the bird_input clock domain crossing module.

Bird:

The “bird” in Flappy Bird is a simple yellow rectangle that moves up and down. Since the bird on the screen can only move up and down, it is not necessary to track its horizontal location in this module, as only the vertical location can be changed. The screen is 480 pixels tall, so 9-bit logic registers are used to store the locations of the top and bottoms of the bird.

The VGA screen formats its coordinates such that the top left corner of the screen is the origin, (0, 0). So, the coordinate for the top of the bird will be a lower value than the bottom of the bird.

The bird in the game is 30 pixels tall by 30 pixels wide, so for all possible changes to its vertical position, the top of the bird must be 30 pixels away from the bottom.

Upon a reset, the bird will return to its original position on the screen, so that its top-left pixel is at (99, 159) and its bottom right pixel is at (129, 189). So, the initial states of the top and bottom bird positions are 159 and 189 respectively. The bird will remain in this reset state until the game is started by the pressing of the start button, where it will then progress to normal behavior. If no bird input (the B button fed into the bird_input module mentioned earlier) is detected, the bird will go down by 10 pixels due to gravity. This is done by increasing both the top and bottom coordinates by 10. However, if bird input is detected, the bird will instead move up by 30 pixels. This is done by decreasing the top

and bottom coordinates by 30. However, if the bird collides with the ceiling when it tries to fly upwards due to the distance between the top of the bird and the ceiling being less than 30, the bird will instead be placed directly below the ceiling, at bird_top = 39 and bird_bottom = 69.

This essentially allows the bird to operate as a rolling register that can be added to or subtracted from at every slow clock cycle pulse, mimicking “real” movement.

Pipe Manager:

The pipes for Flappy Bird are easily the most complicated part of the game. At regular intervals while playing the game, a pair of pipes will appear on the right of the screen, with a gap in between the two of them, and at every new frame the pipes will move leftward in order to force the bird to avoid them, as colliding with a pipe will end the game.

However, since our Flappy Bird game is implemented with hardware instead of software, this implementation becomes extremely complex due to the requirement of needing all functionality to be synthesizable as boolean logic. Due to the difficulty selector, our game has four possible modes with four different rates at which pipes spawn in. Each pipe pattern must also be unique, as there is no way to simply generate multiple versions of a module in SystemVerilog and thus a module for each pipe pattern is necessary. The Pipe Manager module handles the combined functionality of the pipes, making use of an adjustable clock, a random number generator, and 8 pipe modules in order to orchestrate the timing, movement, and locations of pipes on the VGA screen.

Adjustable Clock:

First up is the adjustable clock. Synced to the slower clock pulse, the adjustable clock acts as a counter with an adjustable maximum value where once the maximum value has been reached, a high signal is outputted with the output returning to a low signal at the next slow clock pulse. This allows for a clock with a changeable frequency between “high” signals that will signal the creation of a new pipe pair.

If the difficulty is currently 1, the easiest difficulty, this adjustable clock will output a 1 every 30 slow clock cycles. If the difficulty is 2, it will output a 1 every 25 clock cycles. If difficulty is 3, it outputs 1 every 20 clock cycles, and if the difficulty is 4 it outputs a 1

every 15 clock cycles. The frequency of new pipes appearing becomes quicker as the difficulty is increased.

The SystemVerilog used to make sure that this adjustable clock only outputs a 1 for a single clock cycle of the slow clock is similar in nature to the code used to create the slow clock pulses that only last a single cycle of CLOCK_50.

Random Number Generator and LSFRs:

The next component used in the pipe generation is a random number generator, made possible via an 8-bit LFSR (linear feedback shift register). LFSRs are a series of flip flops with logic gates connected to the “bit 0” flip flop and outputs from inner flip flops that are able to simulate random number generation. The outputs of each flip flop in the series are then combined into an 8-bit register that can rapidly change at every clock cycle.

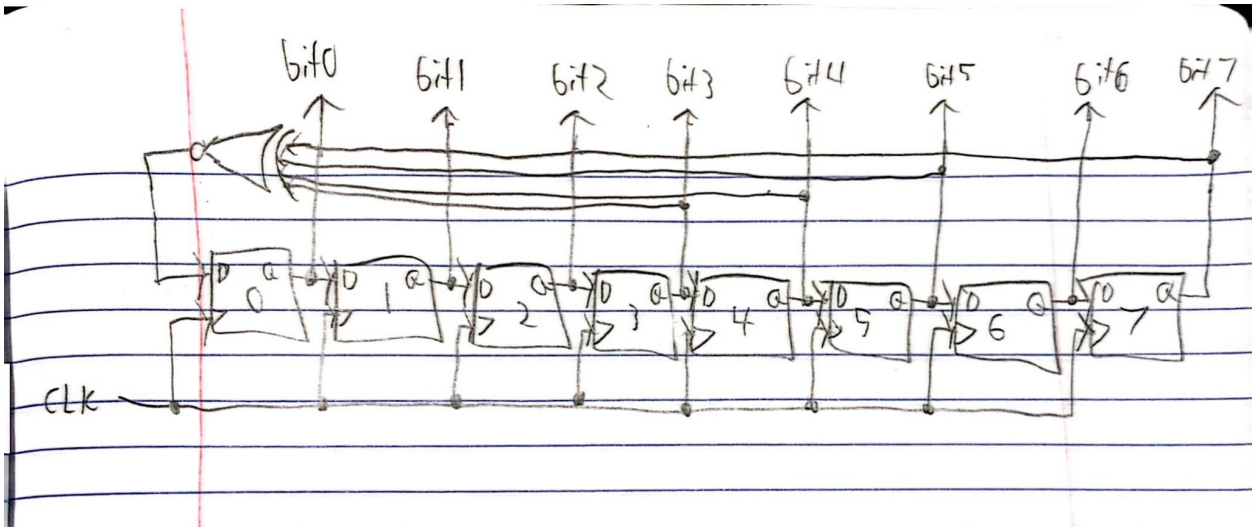


Figure 11: A diagram of the 8-bit LSFR used for random number generation.

The boolean equation used to determine the state of the “bit” 0 at every clock is the following:

```
always_ff @(posedge fast_clock) begin
   bit0 <= (~(random[7] ^ random[5] ^ random[4] ^ random[3]));
end
```

Figure 12: The boolean equation used to determine the state of bit0 in the 8-bit LSFR.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Once all 8 bits have been combined into an 8-bit register, logic is used to divide the random number by 8. The remainder after this operation is the type of pipe pattern we want to generate at the next clock cycle. For example, an LSFR output of 9 will generate pipe pattern 1, and an output of 19 will generate a pipe pattern 3.

However, two things must be kept in mind. First, pipes are only generated at the positive edge of the adjustable clock, meaning that all numbers generated before this are not considered. Second, the next pipe pair generated cannot be a pattern already present on the VGA screen, as our implementation incorporates 8 instantiates of the pipe module with one for each pipe pattern. This means that the patterns currently present on the screen must be kept track of, with a pattern being considered “used” if it was chosen to be placed onto the VGA screen for the game at the positive edge of the adjustable clock. The pattern is considered “used” until the pipe pair disappears off the screen, and while this is the case, if the LSFR asks to generate a pattern currently used it will instead keep the pattern output it found at the last clock cycle, which will always be a new pattern not yet seen in game yet.

Here’s a rundown of what might happen. A positive edge of the **adjustable clock** occurs, with the LSFR linked to **CLOCK_50** asking the pipe manager module to generate pipe pattern 4 at the next **slow clock** pulse. This occurs, and pipe pattern 4 appears in the game. This prevents the random number generated by the RNG module from being 4 until the pipe with pattern 4 exits the game screen. So, if the LSFR generates the number 5 at the next **CLOCK_50** positive edge, followed by the number 4 again, the random number generated will remain 5 due to it being a pattern not yet seen on the game screen yet.

All three clocks (CLOCK_50, slower clock, and adjustable clock) are used in this module. However, since all are built off of CLOCK_50, clock skew is not a concern with the Flappy Bird game.

Pipes:

The third component of the pipe manager module are the pipes themselves. Each pipe module outputs a set of four coordinates: the x-coordinate of the left side of the pipes, the x-coordinate of the right side, the y-coordinate of the top of the hole between the two pipes, and the y-coordinate of the bottom of the hole. When a pipe pattern is not on

screen, each location is initialized as the bottom right corner of the VGA screen: (639, 479). This area is inaccessible by the player and thus cannot cause any issues with collision later on.

Eight pipe modules are used in this game: with 7 unique patterns. Pipe pattern 3 and Pipe pattern 4 are the same pattern. To set up each pipe pattern, the pattern number and the y-coordinates of the opening between the two pipes are initialized as parameters.

Keeping in mind that the y-coordinates of the Flappy Bird game are flipped so that the 0 coordinate is the top of the VGA screen, the following figure shows the y-coordinates of each boundary of each hole between each pipe pattern:

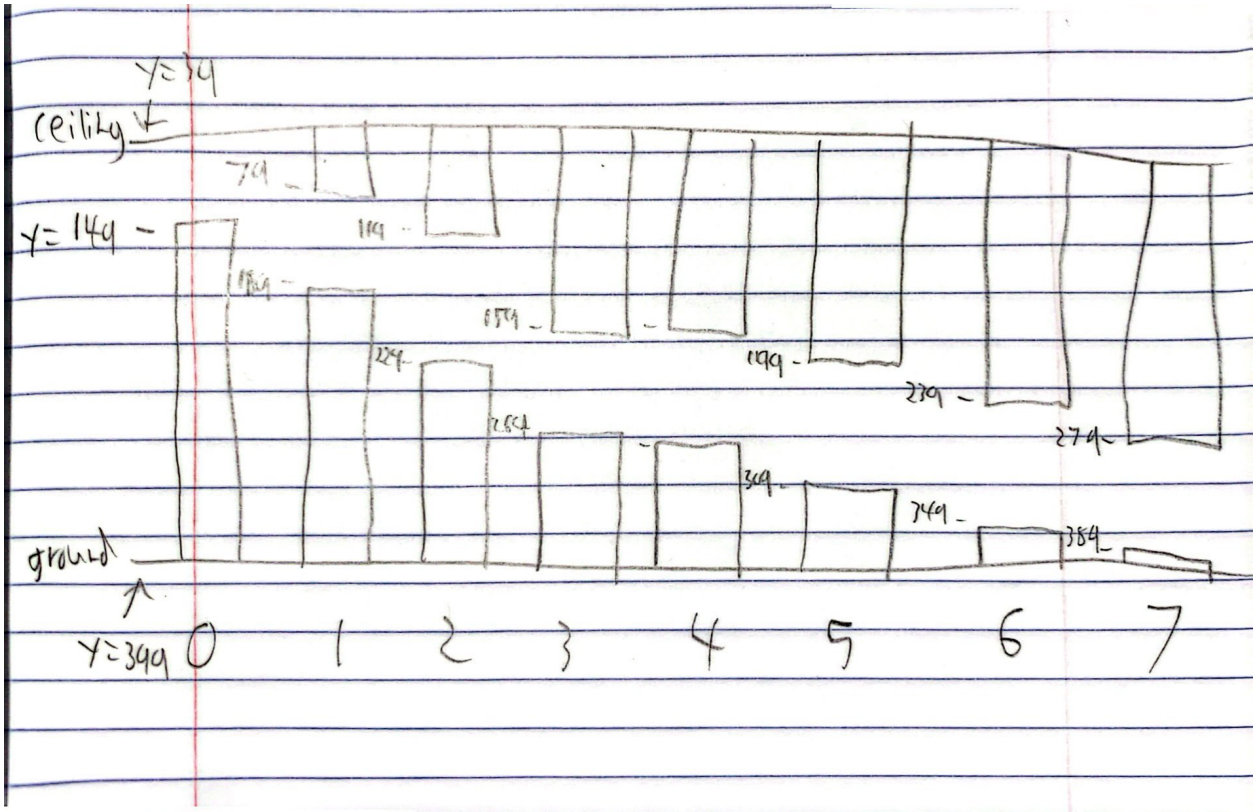


Figure 13: A drawing showing each 8 possible pipe patterns that can be generated by the Flappy Bird game. Note that the ceiling is at y=39 and the ground is at y = 399. All locations end with the number "9", and all holes are 110 pixels tall.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

At every **slow clock pulse**, each of the 8 modules check to see if its pattern has been asked to be initialized. If so, the pipe pattern asked will appear on the very right of the VGA screen, with x-coordinates of 639 for the right side and 579 for the left side.

After every next **slow clock pulse**, if a pipe has been generated already, it will move 16 pixels to the left, meaning its x-coordinates are reduced by 16. If the pipes are successfully avoided by the player once a pipe pattern's x-coordinates coincide with the x-coordinates of the bird, the pipes will despawn from the screen once the left side of a pipe pattern is less than 4. When this occurs, a control signal is broadcast for a single **slow clock pulse** so that the score can be incremented by 1.

Pipe Manager Again:

When combined together, the adjustable clock, random number generator, and all eight pipe modules are used to generate and control the pipes on the screen, with it also incrementing the score by 1 once a pipe pair despawns if avoided. Figures 14 and 15 show two examples of gameplay, one on Difficulty 2 and the other on Difficulty 4:

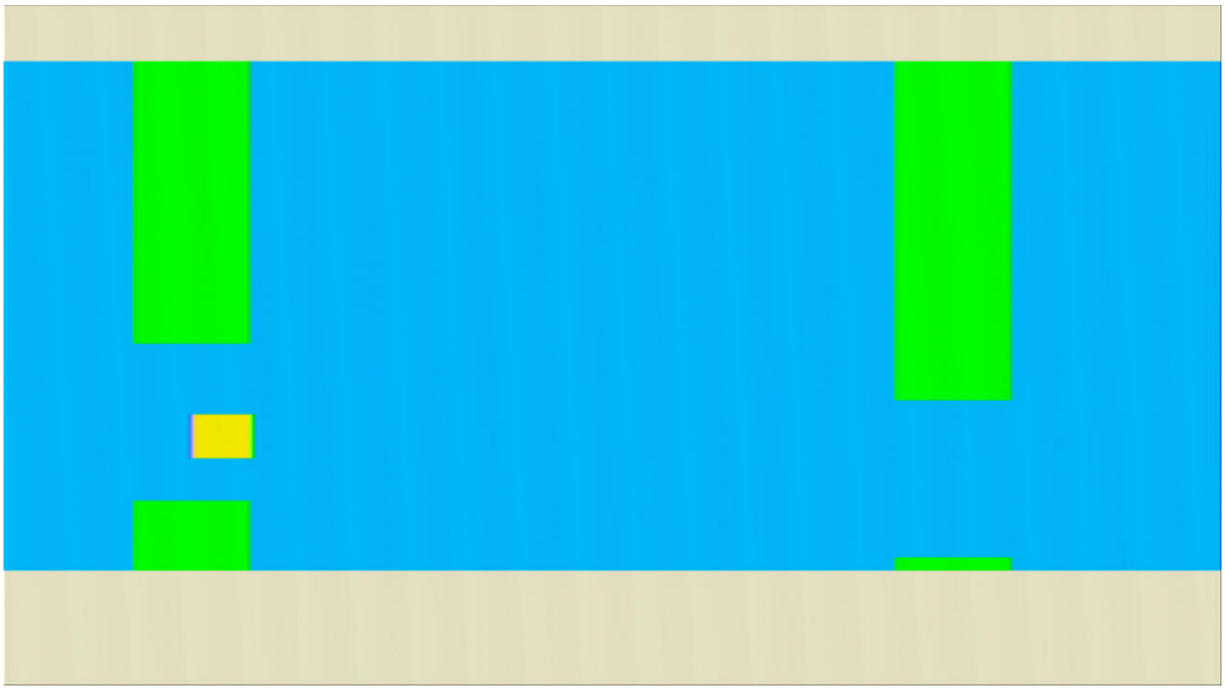


Figure 14: The VGA showing a snapshot of the Flappy Bird game when played on Difficulty 2. Note that only two pipes can appear on the screen at once due to the distance between them.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

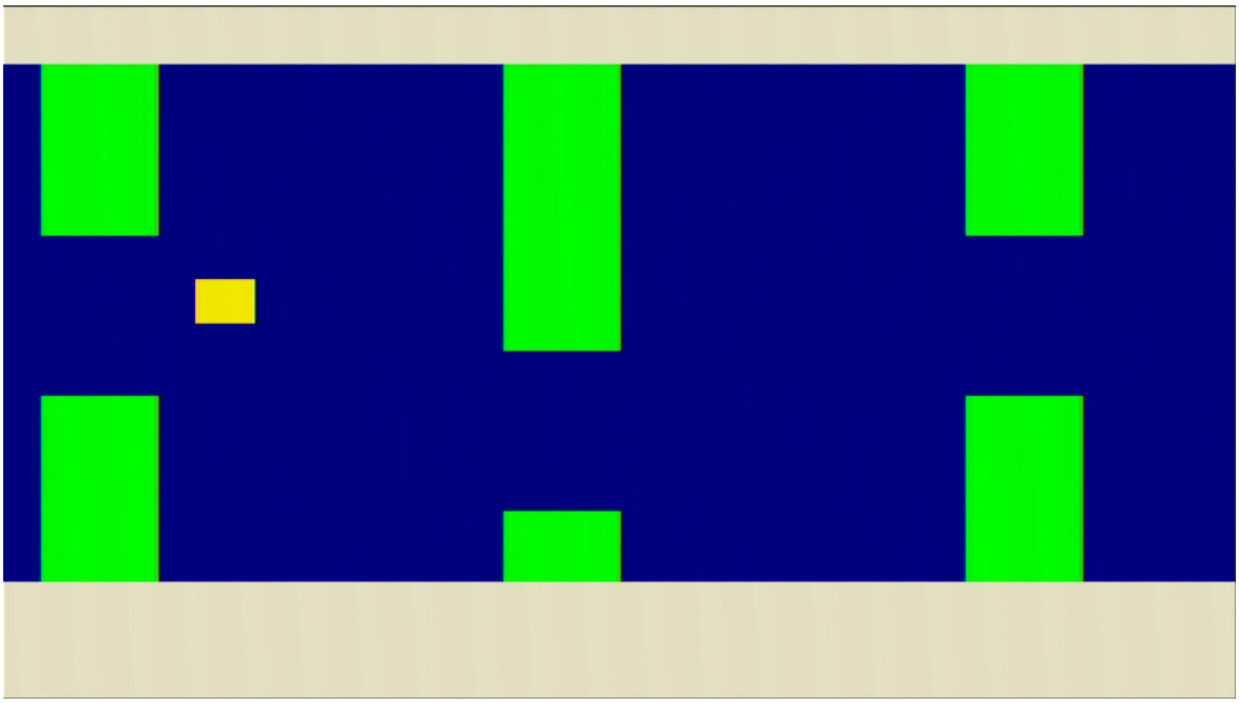


Figure 15: The VGA showing a snapshot of the Flappy Bird game when played on Difficulty 4. Note that since pipes spawn more frequently on this difficulty compared to the previous, the pipes are closer together and a maximum of three can appear on the screen at once.

Collisions:

Finally, the collision module is meant to determine whether the game should end or not. The game ends when either the bird flies into the ground or the bird flies into a pipe. This is done by the use of several if-else-if statements nested and/or stacked atop one another.

An always_comb block is used to check for collisions. By default, the game assumes that the game should continue. If the bird's bottom y coordinate ever becomes less than or equal to the y-coordinate of the ground (399), the game will end.

Likewise, if the bird ever runs into a pipe, the game also ends. To check for this circumstance, a function called check_collision is created that takes in a pipe pattern number as a parameter and checks to see if that pipe's x or y coordinates are overlapping with the x and y coordinates of the bird, and if so, it returns a high output.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

```
//use function to condense the collision logic in code
function logic check_collision(int pipe_index);
      check_collision = (end_x >= pipe_x_left[pipe_index]) &&
                        (start_x <= pipe_x_right[pipe_index]) &&
                        ((bird_top < pipe_y_opening_top[pipe_index]) ||
                         (bird_bottom > pipe_y_opening_bottom[pipe_index]));
endfunction
```

Figure 16: The check_collision function.

The use of a function to do this operation allows us to have an easier time writing the if-else statements that check to see if a specific pipe pattern overlaps with the bird, as all 8 pipe patterns must be checked.

VGA Color Assigner:

Finally, the VGA's pixels must be assigned colors.

The VGA operates by using an extremely-fast CLOCK_25 to cycle through all pixels on the VGA screen and assign them RGB values. This is done alongside CLOCK_50 clock cycles, so the x and y coordinates themselves do not have to be worried about. Instead, it is necessary to use the various x and y coordinates assigned by other modules to draw pixels on the VGA screen in specific colors.

For example, to assign a rectangle of pixels from (0, 0) to (20, 30) with the color red, we would use an if statement that checks to see if the current x and y coordinate being checked is within this region, and if so, its RGB values will be (255, 0, 0).

The following cases are checked while the game is in progress, and for each case the following colors are assigned:

- Y coordinate is less than or equal to 39, OR greater than or equal to 399
    - If so, RGB is (245, 245, 220) (beige)
- X coordinate is in between 129 and 189, and y coordinates are in between the current y positions of the bird
    - If so, RGB is (255, 255, 0) (yellow)
- X and Y coordinates are within a region that should be occupied by a pipe
    - If so, RGB is (0, 255, 0) (green)
- Game is over
    - If so, current pipe pixels remain (255, 255, 0), but all other pixels become (255, 0, 0) (red)

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

However, in the case of any pixels that are not assigned above, or if the game is at the start menu, the pixel color will depend on the current difficulty level selected.

- Difficulty is 1
    - If so, RGB is (176, 224, 230) (powder blue)
- Difficulty is 2
    - If so, RGB is (0, 191, 255) (sky blue)
- Difficulty is 3
    - If so, RGB is (65, 105, 225) (cobalt blue)
- Difficulty is 4
    - If so, RGB is (0, 0, 128) (navy blue)

If the game is at the start menu, all pixels will be of the colors above depending on difficulty, and if in game, any pixels not assigned as a ground, ceiling, bird, or pipe will be of the colors state above.

Overall system:

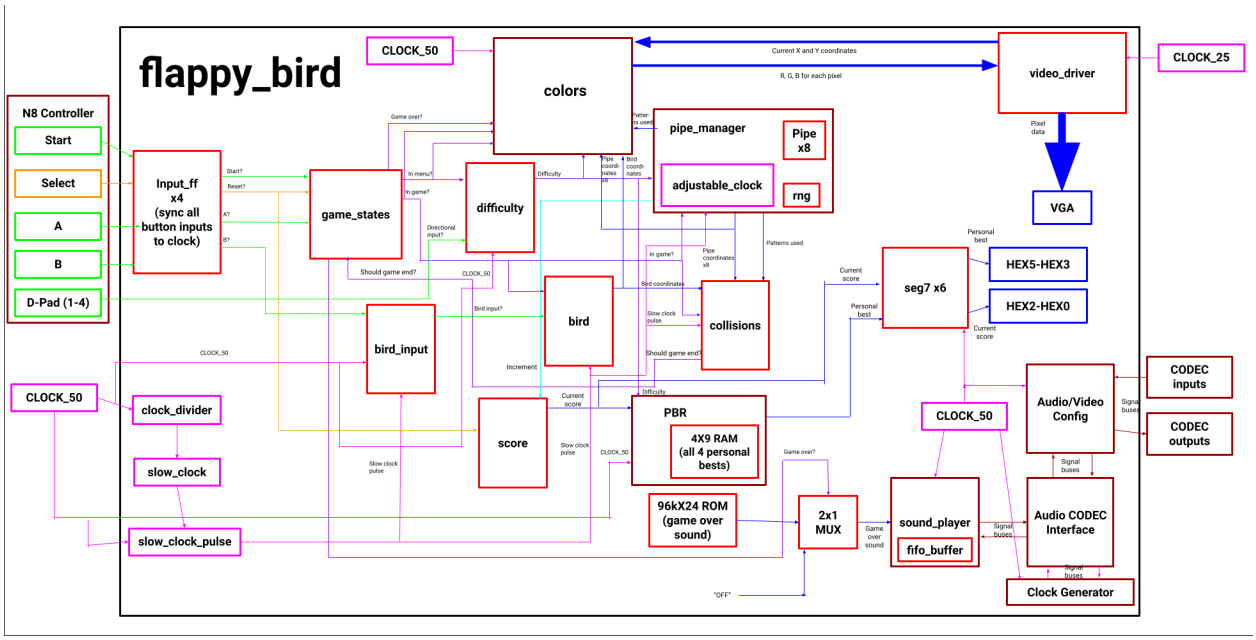


Figure 17: Top-level block diagram of the top-level DE1_SoC module.
Link to diagram for easier reading:
EE 371 Lab 6 Final Block Diagram

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Results:

Basically, this is a simple Flappy Bird game with adjustable difficulty, saved personal best records, and with audio functionality. If the game is at the start menu, a difficulty of 1 to 4 can be chosen, with each difficulty representing a different sky color and a frequency of pipe generation. Difficulties are selected via the N8 controller's directional pad

If the start button on the N8 controller is pressed, the game will begin. Once started, the bird, ground, and ceiling will appear. Pressing the B button will cause the bird to fly up, and not pressing it will have it gradually go down. If the bird hits the ground, the game ends, and if it hits the ceiling, the bird will remain at the ceiling.

Depending on difficulty, the frequency of pipes will differ. But at regular intervals, new pipes will appear on the right of the screen and gradually move to the left.

Pressing the select button will reset the game back to the start menu. Likewise, pressing the A button will immediately end the game if one has begun.

If the game is over, the screen will turn red except for the bird, and an audible "beep" sound will play until the game is reset.

The current score appears on the three rightmost HEX displays on the FPGA and the PBR for the current difficulty appears on the three leftmost HEX displays. If the current score ever exceeds the current PBR, the current score becomes the current PBR, but if not, the PBR will remain whatever achieved score was highest in the past.

The PBRs are stored in a RAM module, while the sound played on the game over screen is stored in a ROM module.

The maximum score possible in the game is 511 points, and if this value is ever exceeded, the score will remain at 511. Every time a pipe despawns if the player avoids a pair of them, the score will increment by 1.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Theoretically, the game could go on forever if someone is skilled enough to play for long enough to go past 511 points.

Some modules were considered novel or complex enough to warrant testbenches to ensure proper functionality. Here are each:

Testbenches:

Bird_tb:

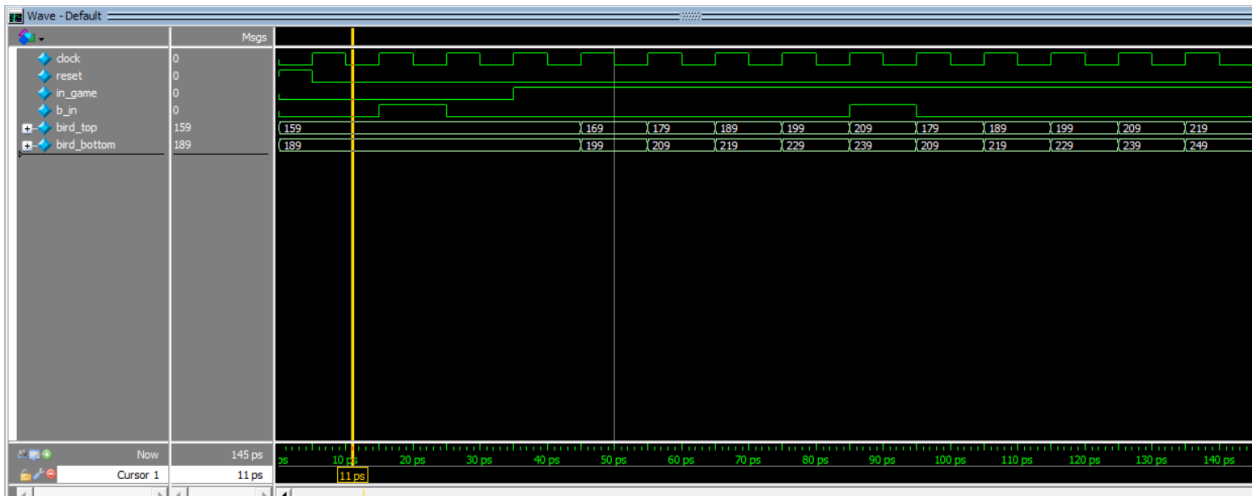


Figure 18: bird_tb testbench.

The bird module handles the upward or downward movement of the bird. After an initial reset, the input B is pressed while the game is not active yet; as can be seen, this does not do anything and the bird remains at its initial position. Once the game is turned on by making “in_game” active, the bird begins to go “down” by 10 at every clock cycle. However, in the clock cycle in which b_in is active, the bird instead goes “up” by 40. This continues until the simulation ends.

Bird_input_tb:

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

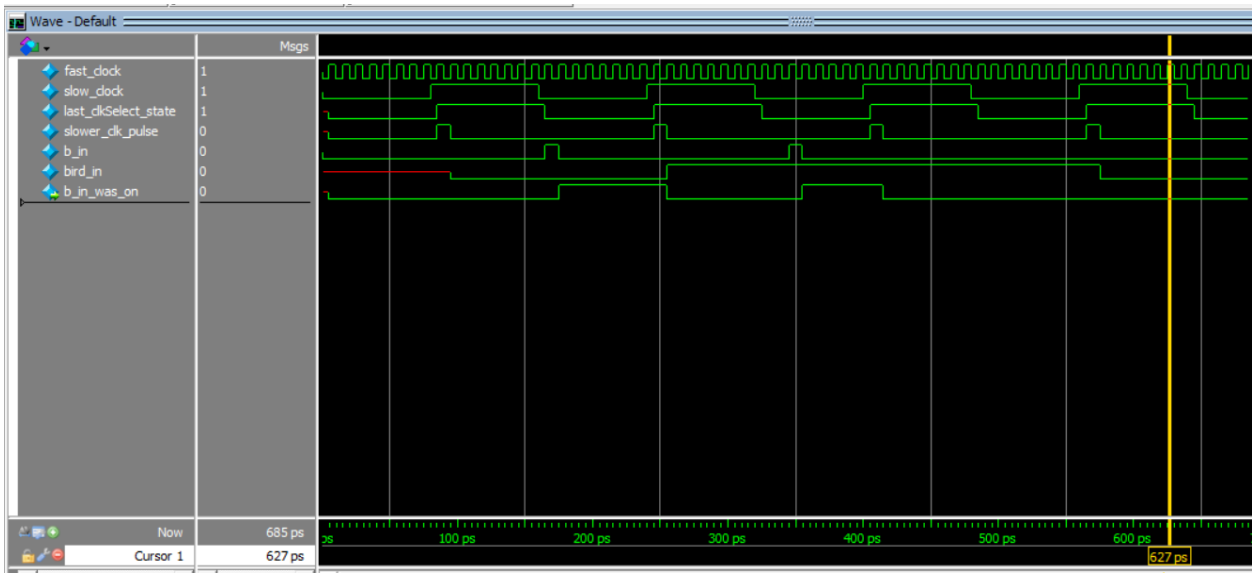


Figure 19: bird_input_tb testbench.

As can be seen, this testbench shows the functionality of the clock domain crosser between the CLOCK_50 (fast clock) and slower clock. When the slow clock has a rising edge, the module detects so and emits a single active fast-clock cycle long pulse. If the b_in is active at all in between rising edges of these pulses, the bird_input will be considered “on”. However, if no B input is detected the bird_input will remain off.

Random_num_generator_tb:

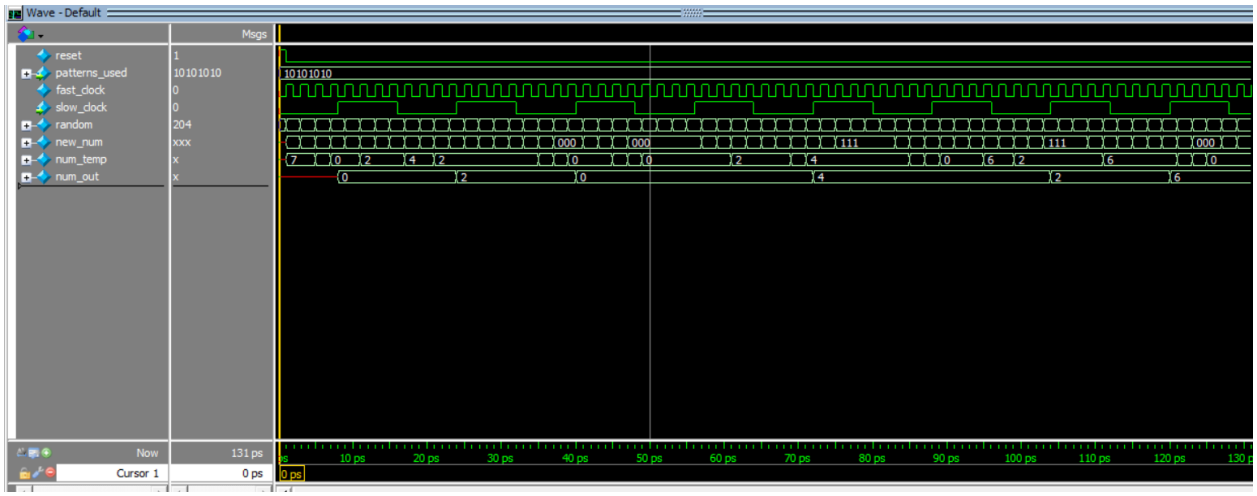


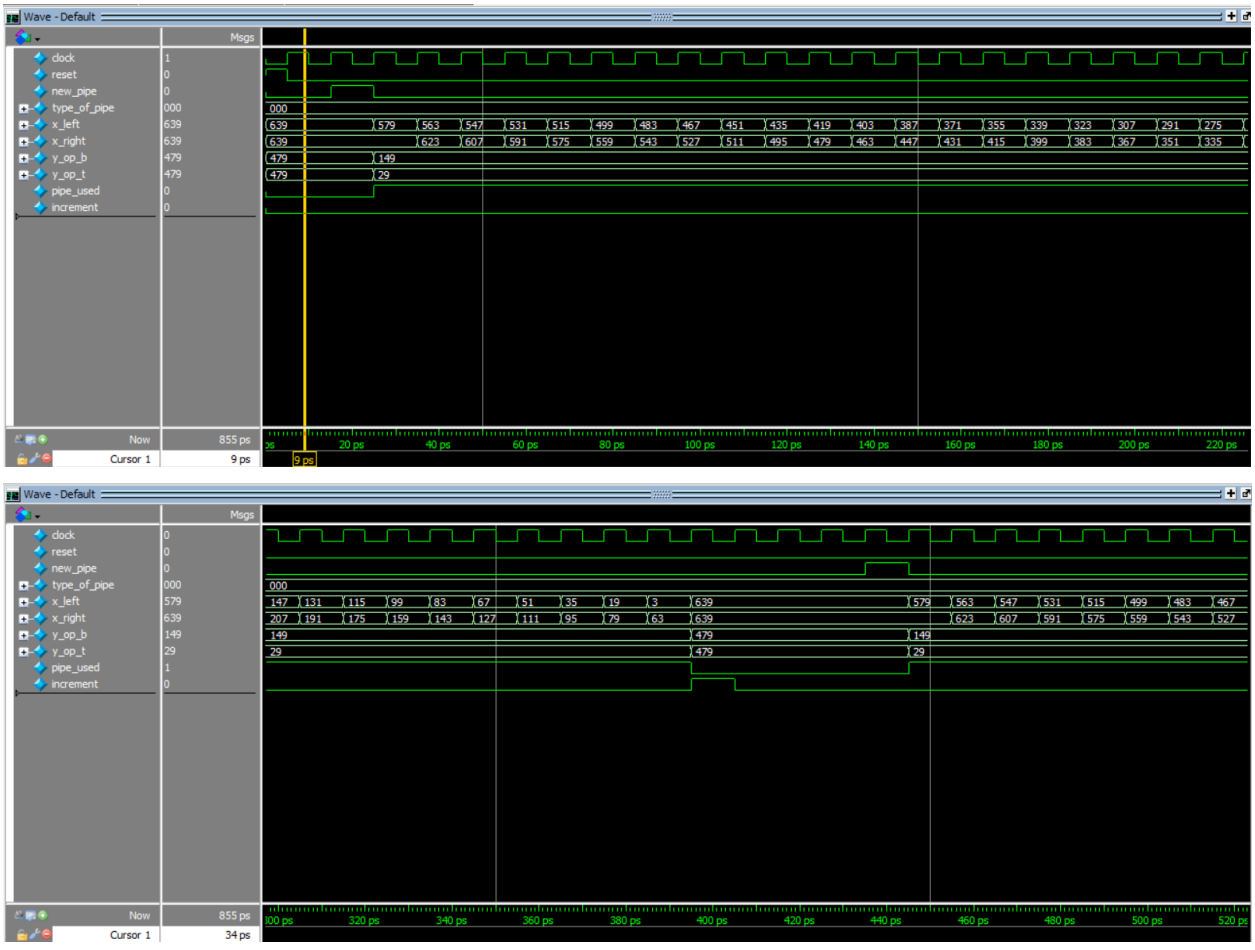
Figure 20: random_num_generator_tb testbench.

This module handles the random number generator for creating new pipes. As can be seen, at every fast clock cycle a new number is generated by the LSFRs. If this number’s remainder divided by 8 is associated with a bit that equals zero in the “patterns_used”

signal, it will be stored in the num_temp logic. At every positive edge of the slow clock, the number currently in num_temp will become the output of the random number generator, allowing it to create a new pipe with that index's pattern.

In this simulation, all odd numbered bits are currently "used", so the RNG module can only output even numbers.

Pipe_tb:



Figures 21A/21B: pipe_tb testbench.

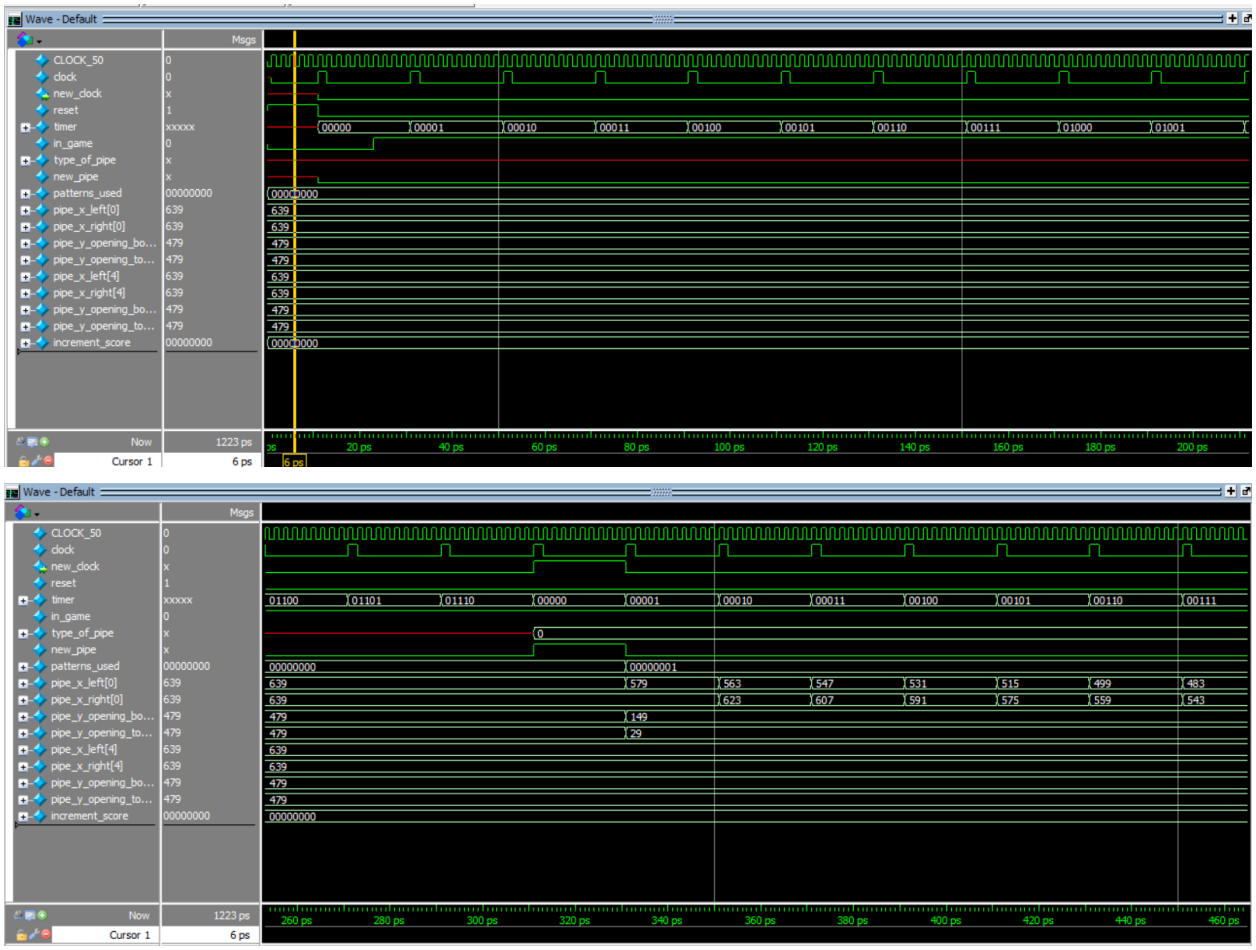
These two figures show the coordinates of a pipe over time. The pipe used in this test simulation is "pattern 0": in which the index is 0, the bottom of the hole is at y=149, and the top is at y=29.

If the pipe was not created in the game yet, its coordinates default to the bottom right corner of the VGA screen. However, once the pipe is created on the positive edge of a

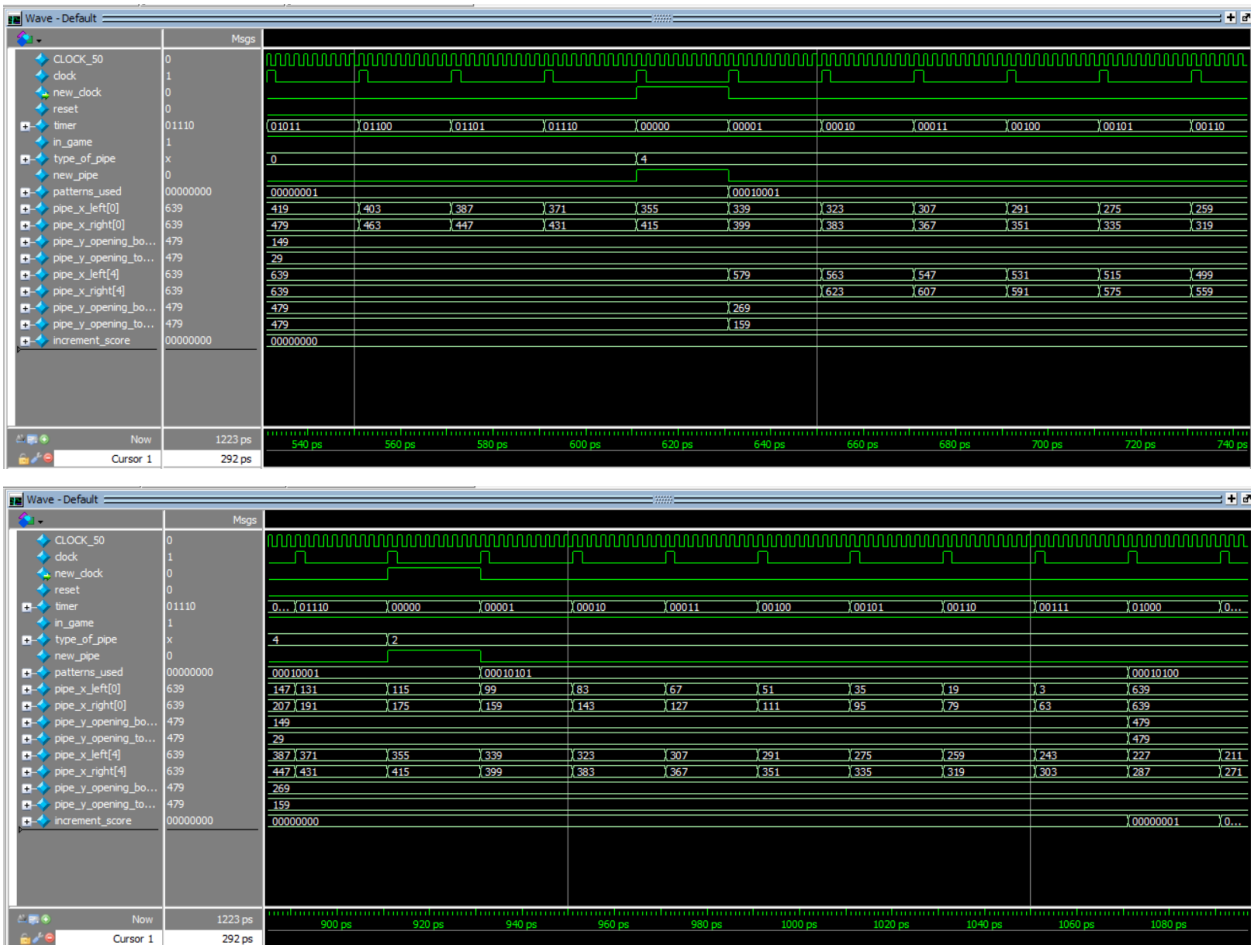
Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

new_pipe signal matched with the type of pipe desired matching this pipe pattern's index, a new pipe will be created at the very right of the VGA screen. At every clock cycle, the pipe moves 16 pixels leftward until it eventually reaches the end, where the left side is at x=3. At the next clock cycle after this, the pipe despawns and the increment signal is active for a single clock cycle.

Pipe_manager_tb:



Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report



Figures 22A-22D: pipe_manager_tb testbench.

This testbench simulates the full functionality of the pipe manager module, with the adjustable clock, random number generator, and all 8 pipe modules. It is assumed that the difficulty used for this simulation is the difficulty 4, in which a new pipe appears after 15 clock cycles of the slower clock. This adjustable clock is represented as “new_clock” by the simulation.

The simulation begins with the game being reset and turned on. In the first 15 slow clock cycles, no activity happens. However, after the 16th slow clock cycle, a new pipe with pattern 0 is requested. This pipe is then created by changing its coordinates to appear on the VGA screen on the very right, and after this the pipe begins moving leftward after every slow clock cycle. Pattern 0 also becomes used by setting patterns_used[0] to active.

After another 15 slow clock cycles, a new pipe with pattern 4 is requested. Like before, the module changes this pipe pattern's coordinates so they appear on the VGA screen, and the new pipe begins moving leftward after every slow clock cycle alongside pipe pattern 4. Patterns_used[4] is also set to active.

15 slow clock cycles later, a new pipe with pattern 2 is requested. This pipe is generated, but a few clock cycles after this pipe pattern 0 reaches the left of the screen. Once this happens, the pipe pattern vanishes from the VGA at the next slow clock cycle, and its pattern is removed from patterns_used by setting patterns_used[0] to low. At the same time, increment[0] is set to high, signaling that the score should be incremented at this clock cycle. If enough time passes, pattern 4 will eventually despawn and increment the score, followed by pattern 2 and whatever comes next.

Collision_tb:

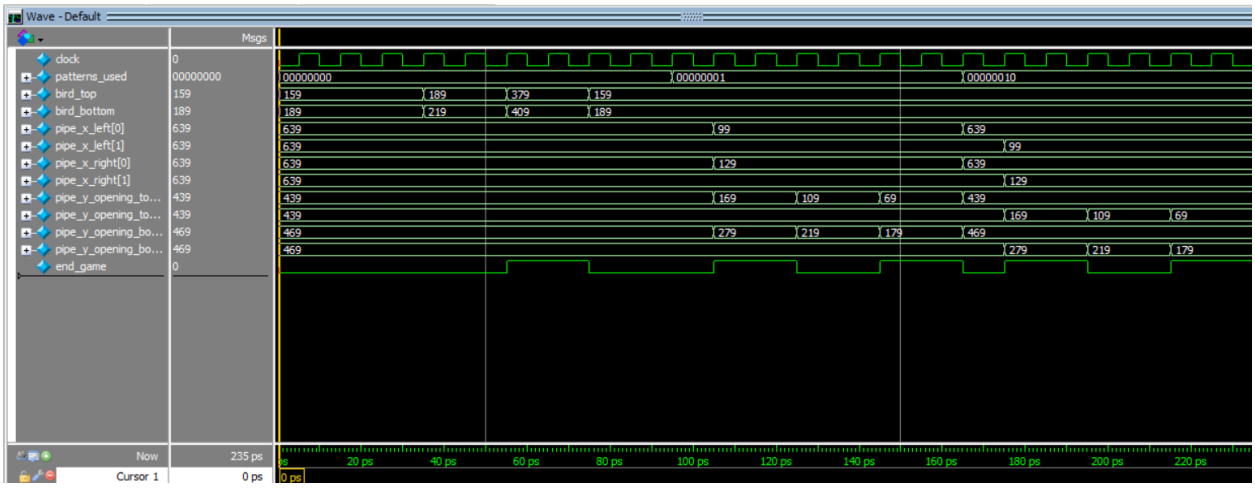


Figure 23: collision_tb testbench.

The collision testbench ensures that the game should end if the bird ever flies into the ground, or if the bird flies into a pipe.

Initially, the coordinates of the bird are at their default location and pipe patterns 0 and 1 are not on the screen yet. The location of the bird changes next so it is lower than before, but not low enough to hit the ground. This does not end the game.

Next, the coordinates are changed so that the bottom of the bird is below the ground (y = 399). This will end the game.

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Next, pipe pattern 0 is spawned in, and its location is set so that its x coordinates overlap with the x coordinates of the bird. Its y's are also changed over three cases: one where the top of the gap between the two pipes is greater than the top of the bird (bird hits top pipe), one where the bird's y coordinates are in between the gap (bird avoids pipes), and one where the bottom of the gap is less than the bottom of the bird (bird hits bottom pipe). The first and third case should end the game, while the second will not.

Finally, this process is repeated for pipe pattern 1 to make sure that more than one pipe pattern can end the game if its pipes collide with the bird.

Flow summary:

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	880
2		
3	Combinational ALUT usage for logic	1496
1	-- 7 input functions	22
2	-- 6 input functions	230
3	-- 5 input functions	253
4	-- 4 input functions	236
5	-- <=3 input functions	755
4		
5	Dedicated logic registers	613
6		
7	I/O pins	117
8	Total MLAB memory bits	0
9	Total block memory bits	2214180
10		
11	Total DSP Blocks	0
12		
13	Total PLLs	2
1	-- PLLs	2
14		
15	Maximum fan-out node	CLOC...nput
16	Maximum fan-out	637
17	Total fan-out	12497
18	Average fan-out	4.60

Figure 24: The Quartus Flow Summary of the compilation of the DE1_SoC module.

Experience Report:

Overall, this lab was a fun experience. The VGA used for this lab was much more intuitive to use and more useful compared to the one used in Lab 5, due to the change in

Brandon Tran (2164739), Chuyang Peng (2328964)
EE371, Aut 2023
December 8, 2023
Lab 6 Report

how the video drivers work and how we only have to specify which coordinates have which colors instead of needing to manually draw each pixel on the VGA. This made making objects for our game much easier, as I cannot imagine how hard it would've been to make Flappy Bird using the line drawing algorithm from the last lab.

Making Flappy Bird was surprisingly enjoyable. Making a side-scrolling game such as Flappy Bird is essentially just designing and implementing dozens of smaller, sub-programs and combining them all into one large program, which is a similar process to what people in real-world settings do to design and implement complex programs.

The hardest part of the project was implementing the pipe functionality, as due to SystemVerilog being a hardware-based language there is the need to make sure that all code written can be synthesizable as boolean logic, meaning that all timings must be constant throughout the FPGA. This forced us to create 8 versions of the same pipe module, one for each pipe pattern of the eight possible, and change our random number generator so that it cannot ask to create a new pipe that is currently already on the game screen. However, implementing this portion of the game was not as difficult as expected, especially after learning about the shortcuts and tips from EE371 compared to what was available in EE271.

All in all, this project was a very rewarding experience, as we can now confidently say that we recreated Flappy Bird on an FPGA and allowed for multiple difficulty settings for it.

Overall, we spent roughly **19** hours working on this lab over the course of two and a half weeks.

- Reading: 30 minutes
- Planning: 2 hours
- Design: 3 hours
- Coding: 7 hours
- Debugging: 5 hours
- Testing: 2 hours