Siran Shen - 304283584
Sahil Shridhar - 004281553
Dylan Hoang - 604268191

Creative Lab Report

Section 1: Introduction and Requirement

General Idea

In this lab, we implemented an FPGA version of a vertical scrolling platforming game, dubbed "Free Fall." The FPGA served as a controller for an object visible on the VGA console output. The object could maneuver left or right in order to fall through cracks between rising platforms and avoid hitting the top of the screen.

Design Requirements

The hardware resources used by this lab are the FPGA Board (Xilinx Nexys 3), the VGA port, and the monitor for console display output. Requirements for the implementation of the game itself can be seen in the following two tables.

Table 1.1: Game Logic Rules the Implementation Must Follow

Object must be able to move left or right based on user input
Object must be able to fall through platform cracks
Object must fall downward at a constant speed when not in contact with a platform or the bottom of the screen
Object must move vertically upward with the platform it is resting on top of
Randomly generated floors with cracks must ascend at a constant speed, that increases every 10 seconds to increase difficulty
The game ends when the ball comes into contact with the top of the screen
The score is linearly dependent on the time the user has lasted
The highest recent score must also be displayed for the user
The user can pause the game state
The user can reset the game state from the beginning

Table 1.2: FPGA Interaction Characteristics

C4 (BTNL) is the left button and D9 (BTNR) is the right button
The pause button is B8 (BTNS)
The reset button is A8 (BTNU)
The switch T5

Section 2: Design Description

The following figure shows a schematic for the modules and combination circuit implemented in this lab.

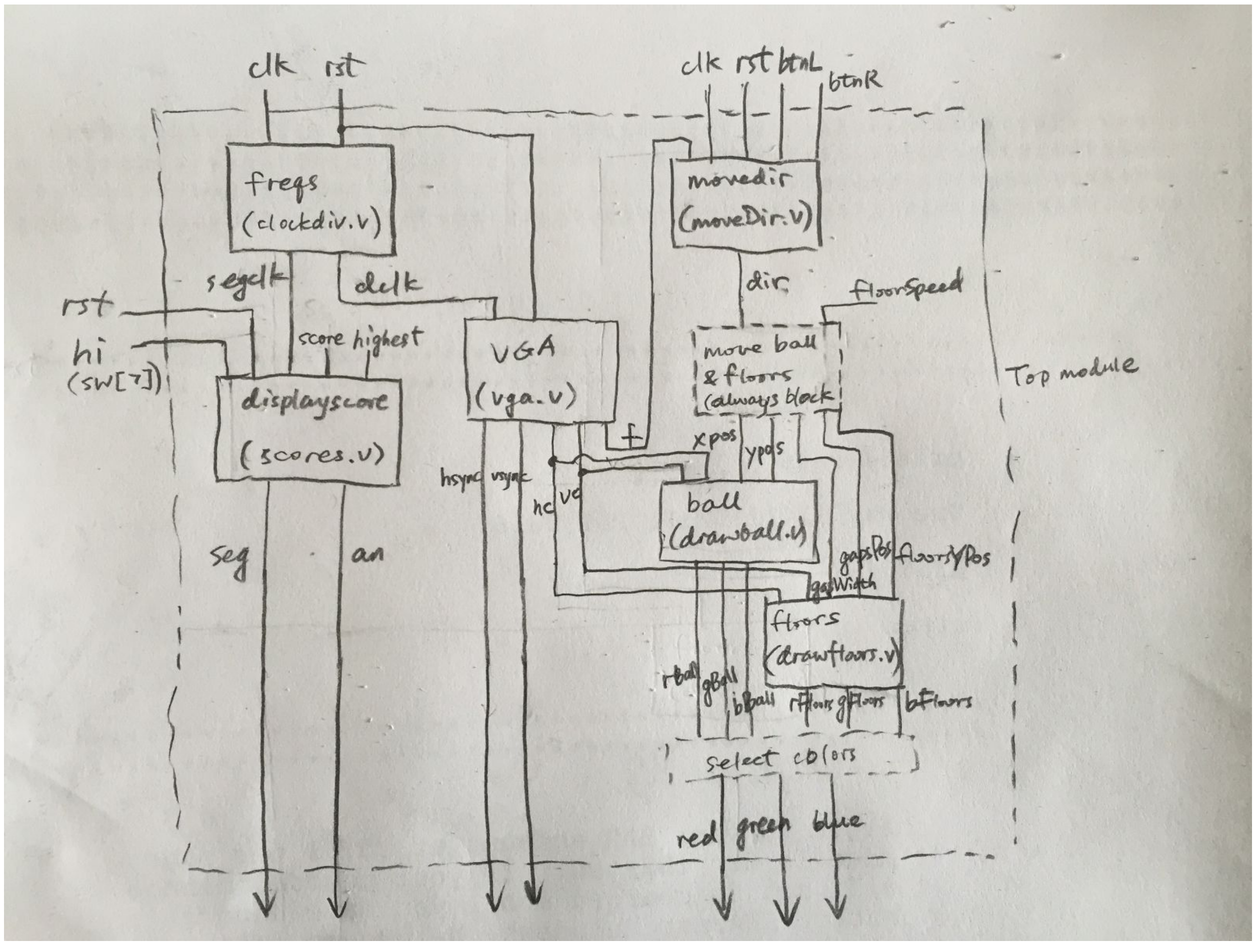


Figure 1. Schematic of Free Fall Game Circuit

freqs: *clockdiv.v*
Inputs for this module were clk - *master clock signal (100 MHz)* - and clr - *asynchronous reset*. Outputs for this module were dclk - *pixel clock signal (25 MHz)* - and segclk - *7-segment display clock signal (763 Hz)*.

The purpose of this module was to convert the incoming master clock signal into two output clock signals that were used for different purposes: dclk, which was used to output the pixels on the monitor console and segclk, which controlled the 7-segment display. The implementation used a 17-bit counter that incremented at a rate of 100 MHz using clk. The module asserted segclk every time its most significant bit was asserted (100 MHz / 2^17 = 763 Hz) and it asserted dclk every time its second-least significant bit was asserted (100 MHz / 2^2 = 25 MHz).

ball: *drawBall.v*
This module utilized various VGA constants in a *definitions.v* file (such as horizontal and vertical front and back porches, sync pulses, ball radius), along with input coordinates (xPos and yPos) on the console output to consistently draw the ball at the given position on the VGA console output. It colored the ball a specific shade of grey using the three RGB outputs, which were 2, 2, and 1 bit(s) large respectively.

floors: *drawFloors.v*
This module utilized various VGA constants in a *definitions.v* file (such as floor thickness and platform gap width), along with three inputs each of 30 bits: yPos, gapsPos, gapsWidth. Variable yPos was used along with the constants to consistently draw three platforms at three different, specified y-coordinates on the VGA console output. In addition, a gap was drawn at the specified gap position at the specified width on each platform. Each floor corresponds to 10 bits in yPos, and its gap also has two 10-bit long pieces of information in gapsPos and gapsWidth. It colored the platforms a specific shade of grey using the three RGB outputs, which were 3, 3, and 2 bit(s) large respectively.

movedir: *moveDir.v*
This module received user input from BTNL (“left”) and BTNR (“right”) on the FPGA and converted these into a 2-bit output dir. It sampled user input at the master clk frequency. If both or neither “left” and “right” are asserted, the output direction is 11 or 00, meaning that the ball should stay put horizontally. If only “left” is asserted, the output direction is 10, meaning the ball should be moving leftwards. 01 represents the ball moving to the right.

VGA: *vga.v*
This module uses the pixel clock signal along with various VGA constants in a *definitions.v* file to consistently output a display frame at a rate of 25 MHz.

displayscore: *scores.v*
The purpose of this module is to output the score of the game on the 7-segment display. It utilizes the implementation of Lab 4's stopwatch to generate the 8-bit seg signal at a frequency of segclk, generated by the clock divider. It also uses a register to store the highest recorded score during a single playthrough of the game. If the T5 switch is asserted, the output number was the register (the high score), else it is the current score, fed in by the top module.

Top module: *fDown.v*
This is the top module that pulled together all of the different circuit elements to properly create game functionality.

The module instantiates all modules and utilizes always blocks to consistently change the parameters of their inputs.

The connection between the FPGA circuit and the VGA connector-monitor aparatus is maintained by **vga**, which consistently outputs a frame at a rate of 250MHz, a signal generated by the **clockdiv** module. In addition, the **score** module's input score was constantly incremented.

The module instantiates three floors using **drawFloors** and consistently feeds this module new y coordinates, gap widths and gap positions. This same logic is applied to drawing the ball using **drawBall**.

The bulk of this code consists of mathematical computations to decide what these parameters should be at any point in time. For example, the calculated floor y positions increases every frame at a rate specified by floor speed, which is incremented every 2^8 frames displayed. The y positions of the three platforms are represented by 10 bits each of the 30-bit value that is passed into **drawFloors**.

The initial state of the game, which is produced at the press of a reset button or upon starting the game, uses hard coded values for these parameters. However, the new floors that are generated use values that are randomized. Specifically, the code generates random values for gap width and gap position. The three gaps in each platform are also represented using 10 bits for each one in a 30-bit quantity.

The module utilized the user input generated from **moveDir** to handle horizontal movement for **drawBall**. For example, if "left" was asserted by **moveDir's** output signal 10, the x position of the ball was to be decremented, to signify leftward movement.

To handle vertical movement, the code utilized conditional statements to see if the ball lay on top of a platform (had equivalent y position). If it did, the ball was to move upwards with the appropriate platform and **drawBall** was fed an appropriate value. This similar implementation was used to handle the freefall of the ball through air and the way it hangs at the bottom of the screen.

Final functionality handled by this top module was a debouncer that made sure the buttons used in this lab (left/right/pause/reset) were sampled at a lower frequency to improve accuracy of the controls.

Section 3: Simulation Documentation

freqs: clockdiv.v
We used very similar code to lab 4 when we did the stop watch and had different timings so we didn't create a testbench specifically for this. However to test out how fast the score was being incremented, we did play with the numbers to get an accurate representation of how fast the clock should move.

ball: drawBall.v
For the drawball module, we just had to see on the VGA that a shape that looked like a ball would be drawn. We were able to get the shape of the circle through known circle equations with a radius and then displayed it by specifying a color.

floors: drawFloors.v
We first had to hard code three floors using similar code that we had with the ball only now it should be extending over the entire screen. Once we got this to work and we could see basically a band on the screen, we then made a hole in a specific x position to see whether a gap would appear on the floor. Finally we then made the gap size and position random using LFSR and we tested it first through the top module test bench (see below) and then confirmed by seeing it on the monitor.

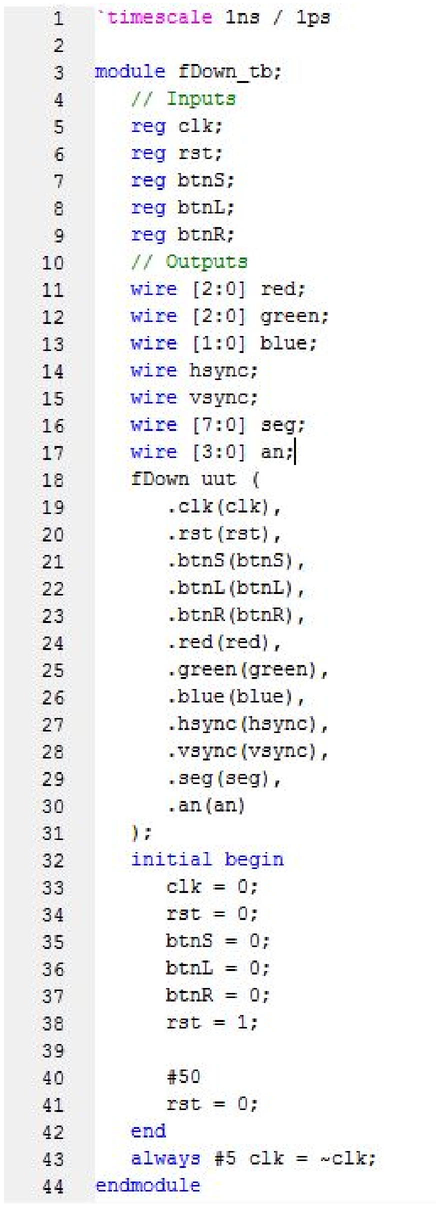
movedir: moveDir.v
This one was more tricky to test because when we saw it on the monitor, the movement was very glitchy/buggy. After making some changes, we were able to see the ball move. Then we had to enforce the direction capability depending on what button to press and the testing was done by using the Xilinx board buttons and test to see that the ball actually moved to the right if we held/pressed the right button and left when we held/pressed the left button

VGA: vga.v

We were able to get this to work by first using the NERP demo. Once we understood how it worked, we just had to implement our version which was easier because we were only using one color for the background while the balls and floors would be a different color. Again this was tested by looking at the monitor and making sure that it looked right.

displayscore: scores.v
This was implemented very similarly to the stopwatch module of lab 4 only know it didn't reset when it 60. It just keep incrementing until the very last number it can make is 9999. Because we used code very similar to the lab before (stopwatch), we did not create another testbench for it.

Top module: fDown.v
Our top module was tested using the test bench below. We had to make sure that the correct values were being piped into the module in order for it to work.



```
`timescale 1ns / 1ps

module fDown_tb;
   // Inputs
   reg clk;
   reg rst;
   reg btnS;
   reg btnL;
   reg btnR;
   // Outputs
   wire [2:0] red;
   wire [2:0] green;
   wire [1:0] blue;
   wire hsync;
   wire vsync;
   wire [7:0] seg;
   wire [3:0] an;
   fDown uut (
      .clk(clk),
      .rst(rst),
      .btnS(btnS),
      .btnL(btnL),
      .btnR(btnR),
      .red(red),
      .green(green),
      .blue(blue),
      .hsync(hsync),
      .vsync(vsync),
      .seg(seg),
      .an(an)
   );
   initial begin
      clk = 0;
      rst = 0;
      btnS = 0;
      btnL = 0;
      btnR = 0;
      rst = 1;

      #50
      rst = 0;
   end
   always #5 clk = ~clk;
endmodule
```

Figure 2. The Testbench for Top Module, *fdown_tb.v*

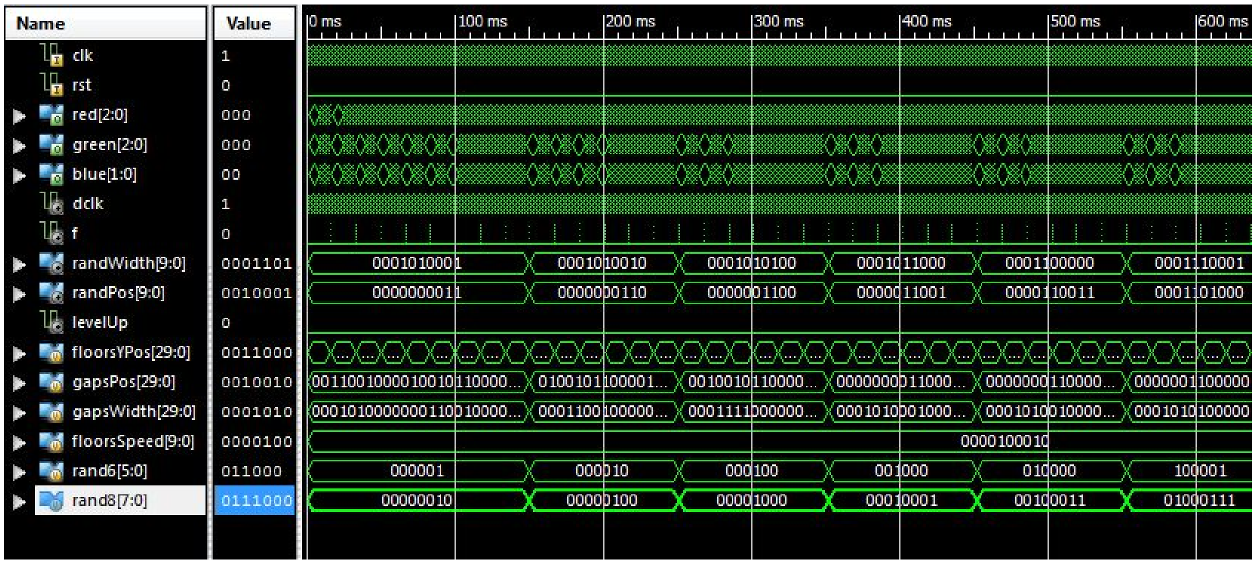


Figure 3. The Simulation of Top Module

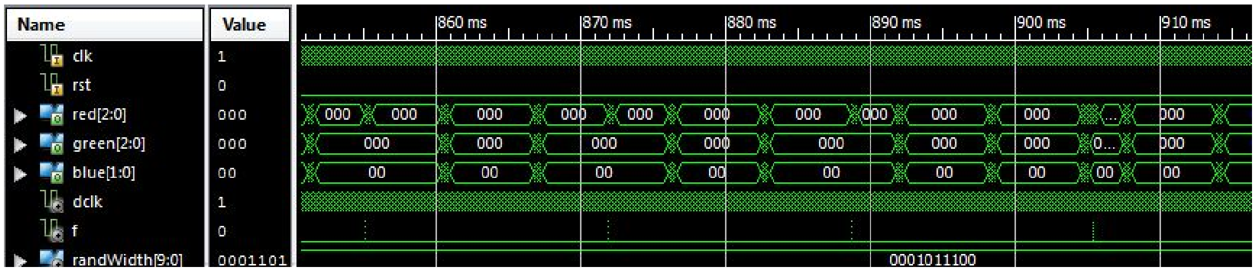


Figure 4. Zoom-in of the Simulation

The simulation is made by setting game not paused even after game over and setting a higher initial value for floors' speed. In the simulation, the variables rand6 and rand8 are LFSRs that have 6 bits and 8 bits, which changed values whenever a floor reached the top of the screen. Variables randWidth and randPos, derived from the above two LFSRs, were used for the gaps. If looking closely, e.g., in the zoom-in Figure 4, we can actually see how the colors (red, green, blue) were rendered in each frame (variable *f* turning 1 at the end of each frame). So it was possible test the VGA code without even looking at the monitor.

Section 4: Conclusion

In conclusion, the implemented design of the Free Fall game met all of the design requirements specified in our proposal. The modules were designed from scratch and most of the proposal's features were implemented, with the exception of the T5 switch being used for a high score button instead of a reset button.

We didn’t manage to get to accomplish any of our stretch goals, but overall this was a great learning experience and showcase our mastery of the Verilog and FPGA circuit design.

Our lab could have been improved by working more on the GUI, such as having a “GAME OVER” screen or adding more aesthetic appeal to the game.