ECE 385

Fall 2024

Final Project

Doodle Jump

Krish Gandhi: krishjg2

Aashay Soni: asoni29

Introduction

Doodle Jump is a simple yet highly addictive mobile game developed by Lima Sky and released in 2009. Players control a character known as “The Doodle,” who continuously jumps upward across a series of platforms. The objective is to reach the highest possible score by avoiding obstacles and enemies while using power-ups like jetpacks and springs.

For our final project, we designed and implemented a simplified version of Doodle Jump on an FPGA board. The goal of the game is to collect as many points as possible by consecutively jumping on blocks without dying. The doodle is horizontally controlled by the player. Due to memory limitations, the maximum score a player can earn is 999 in our game but the limit in the original game is around 355,000,000. This project involves the creation of a real-time gaming environment, integrating interactive gameplay mechanics, dynamic graphics, and modular system design. The project leverages the capabilities of FPGA boards to simulate the game logic, render visuals on a VGA monitor, and accept user inputs through a keyboard or onboard buttons. Additionally, advanced features such as sound effects, scrolling backgrounds, and specialized platforms will be considered to enhance the gameplay experience.

Written Description

Game Functionality

We designed our game to be very similar to the original, given our memory and run time constraints. The game has a mix of essential and advanced features to balance complexity and functionality.

Core Features:

1. Character movement controlled through user input (e.g., ‘A’ for left and ‘D’ for right movements).
2. Continuous jumping of the character.
3. Random positioning and length of platforms, when generated.
4. Collision detection for character-platform interactions.
5. Score calculation.
6. Basic walls of screen (i.e.left and right walls keep the character inside, while top and bottom walls end the game).
7. ‘Game over!’ screen.

Additionally, we also added animation of the character based on the last movement. If the last movement was right, the character will stay facing right. This also applies to the left direction.

Basic Function of the MicroBlaze Processor on the Spartan 7 FPGA

To implement our design of Doodle Jump, we decided to build off of Lab 6, because the MicroBlaze was used to handle I/O with a keyboard input. In Lab 6, we familiarized ourselves with the soft-core Microblaze processor within a System-on-chip framework on the FPGA. We expanded the functionality by adding USB and VGA peripherals that enable interaction with a USB keyboard to control our game displayed on an HDMI monitor. Using SPI communication, the Microblaze processor interfaces with the MAX3421E USB host controller to retrieve keypress data. The MAX3421E enables this process by facilitating USB communication through SPI, allowing the Microblaze to communicate with the FPGA to manage keyboard inputs. The design also includes a VGA-to-HDMI IP block, converting VGA signals into HDMI format, allowing real-time visualization of the game state on an external HDMI monitor in response to user input.

How the I/O Works

The Microblaze processor within our FPGA configuration communicates with external components through memory-mapped I/O that allow software running on the Microblaze to access I/O peripherals as if they were memory locations. We established serial communication interfaces using UART for external data transmission. Each key press from a connected keyboard is processed through the Microblaze and sent to the

FPGA, so it can perform logic and control the game. The Microblaze translates these inputs into keycode signals, which are then sent to the FPGA. The MAX3421E USB host controller enables the Microblaze to send and receive data through the USB keyboard. When a data transfer is initiated, the MAX3421E's internal read and write buffers temporarily store the data, making it accessible for the Microblaze to process. This buffer system allows the Microblaze to update the display based on immediate user inputs. The Microblaze can continuously manage and synchronize input from the keyboard with the player's movement on the display.

VGA Operation and How the Modules Interact

The VGA module generates timing signals, determines pixel coordinates, and enables the display at a 640x480 resolution. Given inputs of platforms and character positions, we wrote logic to decide the color for each pixel in the Color Mapper. The Color Mapper module takes the pixel coordinates (DrawX and DrawY) from the VGA and returns the RGB value for that pixel. The Ball (name is unchanged from Lab 6) module manages the character's motion by updating its X and Y coordinates in response to keypress inputs—moving left or right by adjusting X. The Ball, Platforms, Scoreboard, Font_rom, Color Mapper, and VGA modules work in sync to control and display the game on the screen. As the VGA controller scans the display pixel by pixel, it collaborates with the other modules to identify the locations of the player and platforms, and with the Color Mapper to retrieve the correct color to display at that pixel. This setup enables dynamic and responsive interaction with visual feedback on the VGA display.

Block Diagrams

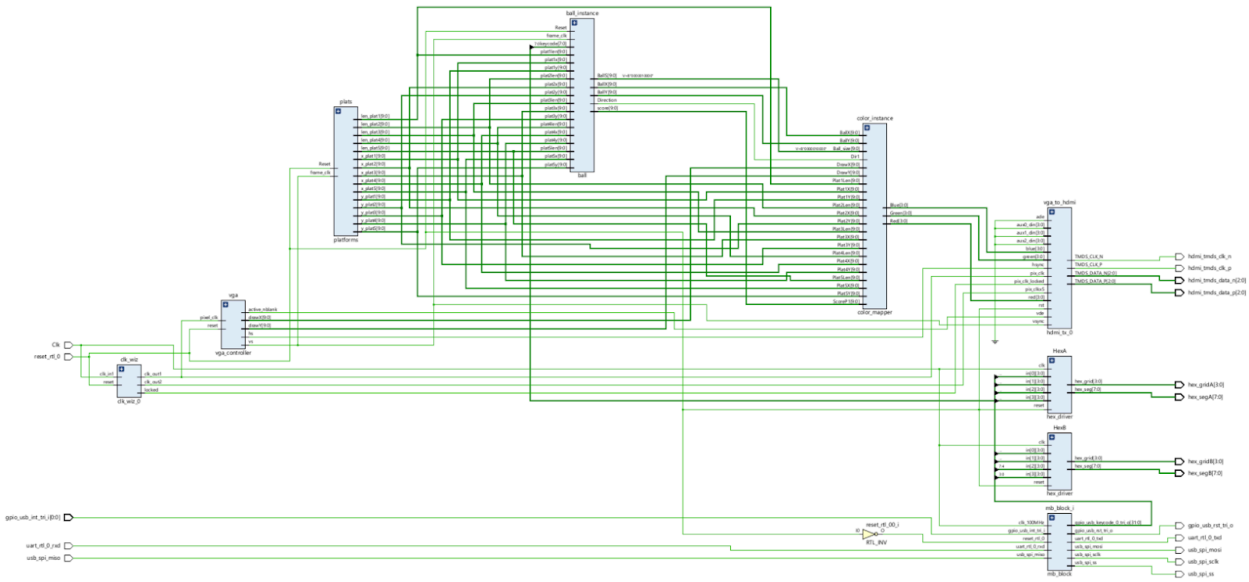

Figure 1: Top Level Block Diagram of Design

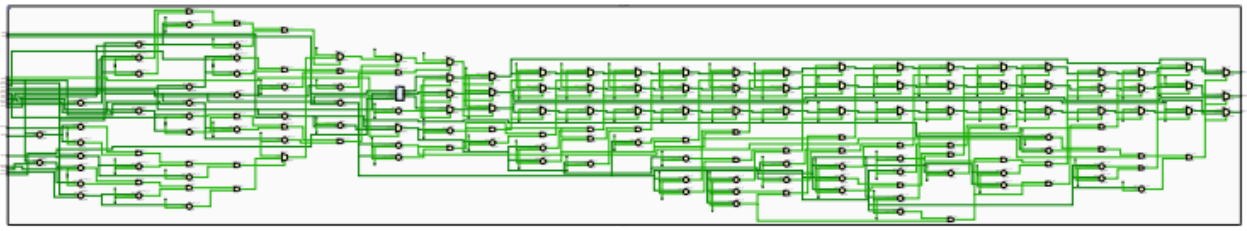

Figure 2: Block Diagram of Color Mapper Module

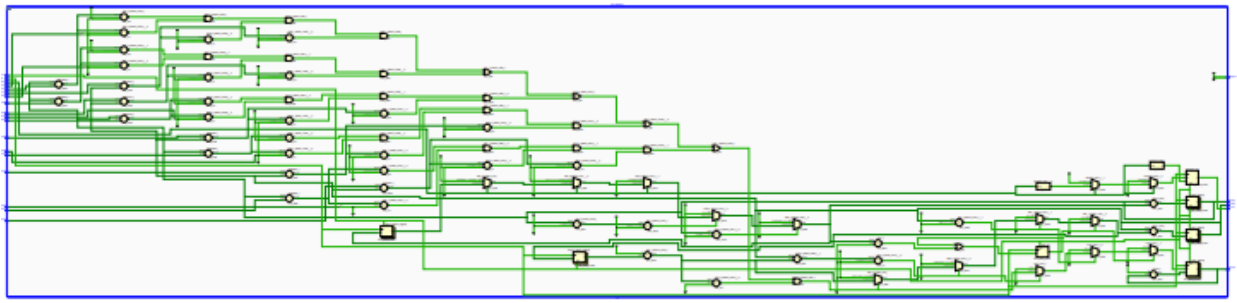

Figure 3: Block Diagram of Ball Module

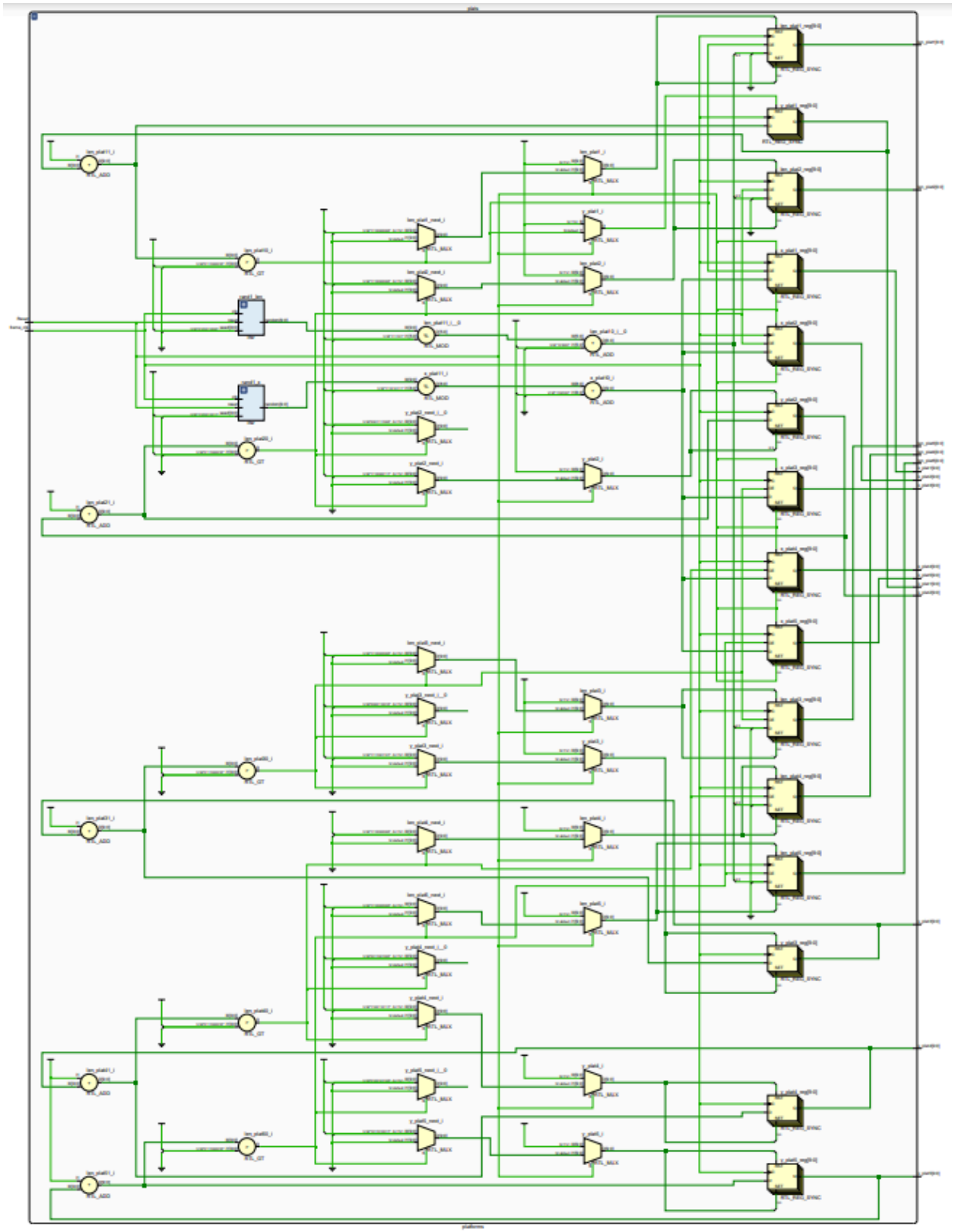

Figure 4: Block Diagram of Platforms Module

Module Descriptions

Module Descriptions

Module: ball.sv (name is unchanged from Lab 6, but functionality is adjusted)
Inputs: Reset, frame_clk, [7:0] keycode, [9:0] plat1x, [9:0] plat1y, [9:0] plat1len, [9:0] plat2x, [9:0] plat2y, [9:0] plat2len, [9:0] plat3x, [9:0] plat3y, [9:0] plat3len, [9:0] plat4x, [9:0] plat4y, [9:0] plat4len, [9:0] plat5x, [9:0] plat5y, [9:0] plat5len
Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS, [9:0] score, Direction
Description: The Ball module controls the position, animation, and movement of the player on the screen. Based on keyboard inputs (keycode and platform positions), it updates the X and Y coordinates, ensuring the character moves according to user commands (left or right) and bounces back upon hitting screen boundaries. This module also controls the jumping effect of the player. This module was edited from Lab 6.
Purpose: This module is called in the top level module. It takes in the keycode from the Microblaze processor and allows the user to control player movement across the screen.

Module: Color_Mapper.sv
Inputs: Dir1, [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] DrawY, [9:0] Ball_size, [9:0] Plat1X, [9:0] Plat1Y, [9:0] Plat1Len, [9:0] ScoreP1, [9:0] Plat2X, [9:0] Plat2Y, [9:0] Plat2Len, [9:0] Plat3X, [9:0] Plat3Y, [9:0] Plat3Len, [9:0] Plat4X, [9:0] Plat4Y, [9:0] Plat4Len, [9:0] Plat5X, [9:0] Plat5Y, [9:0] Plat5Len
Outputs: [3:0] Red, [3:0] Green, [3:0] Blue
Description: The Color Mapper determines the RGB value for each pixel based on its position. This module takes in many input values of game objects and outputs the RGB values for the requested pixel. This module was edited from Lab 6.
Purpose: This module is used in the top level module to assign the RGB values of pixels on the display. The output Red, Green, and Blue signals are fed to the VGA to HDMI converter to be sent to the screen.

Module: lfsr.sv
Inputs: clk, reset, [9:0] seed
Outputs: [9:0] random
Description: This module is a 10-bit Linear Feedback Shift Register for pseudo-randomness. Given an input seed, this register will output every 10-bit number in a pseudo-random order, before indefinitely outputting 10’b0000000000. We created this module specifically for this project.
Purpose: This module is used in the platforms module to “randomly” generate values to use for the horizontal position and length of a new platform.

Module: platforms.sv

Inputs: Reset, frame_clk
Outputs: [9:0] x_plat1, [9:0] y_plat1, [9:0] len_plat1, [9:0] x_plat2, [9:0] y_plat2, [9:0] len_plat2, [9:0] x_plat3, [9:0] y_plat3, [9:0] len_plat3, [9:0] x_plat4, [9:0] y_plat4, [9:0] len_plat4, [9:0] x_plat5, [9:0] y_plat5, [9:0] len_plat5
Description: This module generates 5 platforms for the game. It controls the generation and movement of all 5 platforms. It outputs the length of each platform, as well as it's X and Y position. We created this module specifically for this project.
Purpose: This module is called in the top level module. It's outputs are sent to the ball module and color mapper module to display and control gameplay.

Module: font_rom.sv
Inputs: [9:0] row, [9:0] col, [7:0] charCode
Outputs: out
Description: This module acts as a 4-bit-by-8-bit font rom. The only available characters (in order) are space, A-Z, 0-9, semicolon, and exclamation point. The module takes in the X and Y value of the character, along with the character code of the desired character. It then outputs a 0 or 1, depending on if that pixel should be filled in or not. We created this module specifically for this project.
Purpose: This is used in the scoreboard module to print out the player's score and other things.

Module: scoreboard.sv
Inputs: [9:0] DrawX, [9:0] DrawY, [9:0] p1_score, [9:0] BallY
Outputs: [3:0] Red, [3:0] Green, [3:0] Blue
Description: This module is used to print out the scoreboard of the game. Given the position of a requested pixel, the score of the player and the Y position of the player, this module will return the RGB value of the requested pixel. A "Game Over!" is displayed on the scoreboard if the player falls below the bottom of the screen. We created this module specifically for this project.
Purpose: This module is used in the Color Mapper module to display the scoreboard for the game.

<u>Module Descriptions from Lab 6</u>
Module: mb_intro_top.sv
Inputs: clk, [15:0] sw, [3:0] btn, uart_txd
Outputs: [15:0] led, uart_rxd
Description: This is the top-level module for the Week 1 design, connecting the Microblaze processor and handling inputs and outputs such as LEDs and UART. It coordinates components for a simple LED blinking demonstration. In our implementation of Doodle Jump, this was unchanged from Lab 6.

Purpose: This module introduces the Microblaze in Week 1, setting up the environment for basic control over LEDs and serial communication.

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: The VGA controller is responsible for generating timing signals that control the display's synchronization and scanning. It produces horizontal and vertical sync pulses to drive each line and frame refresh, allowing the screen to display images at a resolution of 640x480. The module also includes logic to blank areas outside the display boundaries and reset the sync signals at appropriate intervals. In our implementation of Doodle Jump, this was unchanged from Lab 6.
Purpose: This module is called in the top level module to sync the VGA output. It drives the drawX and drawY signals, which are then sent to ColorMapper.sv.

Module: mb_usb_hdmi_top.sv
Inputs: Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd
Outputs: gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB
Description: This top-level module integrates the Week 2 design components, including SPI communication, USB control, VGA-HDMI conversion, and hex displays. It manages signals between submodules and connects inputs and outputs like UART, SPI, and HDMI signals. In our implementation of Doodle Jump, this was unchanged from Lab 6.
Purpose: This module acts as the central hub for the Week 2 design, coordinating communication between different components and enabling the SoC to interface with various peripherals.

Name: hex_driver.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: This module converts the four 4-bit input signals in[4] from binary value to an 8-bit binary value that represents a hexadecimal character on the seven segment display. We made no changes to this module. In our implementation of Doodle Jump, this was unchanged from Lab 6.
Purpose: This module is used in the top level module to convert data from hexadecimal representations to seven segment displays. These representations are then outputted to the seven segment displays on the Urbana board. This was useful for showing debug information directly on the FPGA.

Block Design Components from Lab 6
Component: microblaze_0
Inputs: INTERRUPT, DEBUG, Clk, Reset
Outputs: DLMB, ILMB, M_AXI-DP
Description: This is the core processing unit, a 32-bit soft-core Microblaze processor that executes program instructions and manages data flow within the system. It interfaces with connected peripherals through the AXI interconnect. In our implementation of Doodle Jump, this was unchanged from Lab 6.

Axi_gpio_0: An AXI GPIO module that provides input and output functionality for simple external devices, such as buttons or LEDs. It enables the Microblaze to control and monitor digital signals through the AXI bus. In our implementation of Doodle Jump, this was unchanged from Lab 6.

Axi_gpio_1: Another AXI GPIO module, similar to axi_gpio_0, which offers additional input/output support for the system. It can connect to other external components like switches or additional LEDs. In our implementation of Doodle Jump, this was unchanged from Lab 6.

Axi_gpio_2: A third AXI GPIO module, allowing for more general-purpose I/O expansion. It interfaces additional peripherals with the Microblaze processor through the AXI bus, supporting specific design needs for extended I/O. In our implementation of Doodle Jump, this was unchanged from Lab 6.

Microblaze_0_axi_intc: An AXI interrupt controller that manages and prioritizes interrupt signals from various peripherals, ensuring the Microblaze processor can handle multiple external events efficiently by routing and handling interrupt signals. In our implementation of Doodle Jump, this was unchanged from Lab 6.

Axi_uartlite_0: A lightweight UART module for basic serial communication, facilitating data transmission and reception over a serial interface. It is useful for debugging or for simple data exchange between the system and external devices. In our implementation of Doodle Jump, this was unchanged from Lab 6.

Design Decisions and Future Expansion
Design Decisions and Thought Process

The design process for implementing Doodle Jump on an FPGA platform required careful planning, modular design, and creative thinking to address both gameplay

mechanics and system integration. Here, we outline the steps and thought processes that guided the creation of this project.

While designing this project, we implemented the following features:

1. Keyboard control
2. Jump mechanism
3. Collision detection
4. Scoring
5. Scoreboard
6. Random platform algorithm
7. Platform generation and player graphics generation
8. Basic walls (die when hit the bottom screen)
9. Rendering platforms (platforms moving down when the player jumping up)
10. Animation of Player

To implement keyboard control, we simply decided to build off of the Lab 6.2 code. This was a fairly easy and obvious choice, because we did not want to deal with the hassle of setting up the MicroBlaze and MAX3421E USB interface again.

To implement the jump mechanism, collision detection and scoring, we decided to edit the code for the ball from Lab 6.2 to fit our application, with some inspiration from physics. As it is known, the constant for gravity is measured as an acceleration, i.e. it is unit length per unit time per unit time. To imitate this, we created a new variable in the ball.sv module. This allowed us to always accelerate the character downwards, instead of simply moving the character at a constant velocity. The “jump” effect occurs when the character is moving downward and overlaps with a platform. At this moment, a “collision” occurs and the character’s vertical velocity is set to -21, to move upward on the screen. The acceleration variable will then bring the character back down. Everytime a collision occurs, the character gains one point.

To implement the platform generation, player graphics generation, and scoreboard, we edited the Color Mapper from Lab 6.2. We essentially inputted all of the objects we wanted to display, then wrote logic to determine what RGB value the Color Mapper should output. The scoreboard is made up of 10 columns and 39 rows of characters. The characters are 4 bits by 8 bits, so we also wrote a custom font rom. The scoreboard displays the player’s score and a “Game Over!” message when the game ends.

To implement the random platform generation, we decided to use pseudo-random algorithms. At first, we wanted to use the built-in $urandom_range() function in SystemVerilog, which returns an unsigned integer within a given range. However, we

quickly realized that this is not synthesizable. We then pivoted and designed a 10-bit Linear Feedback Shift Register (LFSR) module. In the platforms module, there is one to generate X values and one to generate lengths of the platforms. The seed for the X value generator is 10'd859, while the seed for the length generator is 10'd344. A platform can have an X value in [100, 570], inclusive, and a length of [50, 100], inclusive. LFSRs work by starting at a non-zero seed, then going to every possible number before finally reaching 0 and staying there. In our case, this works because the next output of the LFSR is set to {random[8:0], random[9] ^ random[6]} and updates every clock edge.

To implement the basic walls of the game, we simply wrote logic in the ball module. If the player approaches the left or right walls of the screen (at X = 61 and X = 639, respectively), Ball_X_Motion_next is set to 1 or -1. This creates a sort of bouncing effect when the player attempts to run into the left or right walls. To implement dying at the bottom wall, we wrote similar logic. If the character falls below Y = 479, the vertical velocity of the ball is held constant at 0 and the game ends.

To implement the rendering of platforms, we simply designed an entirely new module for platforms, instead of trying to re-tool an existing one.

To implement the animation of the player, we added another variable in the ball module to track the direction of the last horizontal movement of the character. This variable was 0 or 1, depending on the direction of the ball. This variable was sent to the color mapper, where it was used in the calculation of drawing the player sprite. Since the sprite was designed facing right, if the direction variable was 1, no change was made. If the direction variabel was 0, the character needed to flip 180°. To do this, we simply changed the requested X value in the sprite from DrawX - BallX to 15 - (DrawX - BallX). This would allow the color mapper to output the correct RGB values to flip the character 180°.

Possibility of Future Expansion

We intentionally designed our project to be easily updated, so we could easily add additional features. In future versions of our game, we can add a second player, a leaderboard, improve the graphics and add more sprites in order to make the game more detailed.

We can easily add a second player by making slight modifications to the ball module. First, we would need to change the keycode logic to look at all of the keys being pressed, instead of just the first one. Then, in the player 2 module, we would need to change the control keys from A and D to left arrow and right arrow. Finally, we would need to pass along the outputs from the player 2 module to the color mapper.

We could implement enhanced graphics by introducing sprite-based rendering using a dedicated sprite controller module. This module would handle the sprite data using a finite state machine and output the current state of the sprite. This output would then be passed along to the color mapper. This could also be utilized to add more character sprites to the game.

We can add dynamic platform behaviors so the game is more challenging, hence more fun for the player. To create dynamic platform behaviors, we would extend the platform generation module to include logic for state-based behaviors such as disappearing, moving, or breaking. Using parameters, these behaviors could be encoded into the platform attributes, and an FSM would manage their transitions during gameplay.

A leaderboard can be added by using BRAM, similar to the Lab 7 implementation. The high scores could be stored in BRAM, and then recalled and printed to the scoreboard. A new high score can be written to BRAM in a similar way as Lab 7.

Design Resource and Statistics

LUT	4319
DSP	3
Memory (BRAM)	8
Flip-Flop	2793
Latches	0
Frequency	113.063 MHz
Static Power	0.466 W
Dynamic Power	0.390 W
Total Power	0.075 W

Table 1: Design Resources and Statistics in IVT

Table 1 shows the Design Resources and Statistics for our implementation of Doodle Jump. Frequency was calculated using the formula $f = \frac{1}{\mathrm{T - WNS}}$, where the period T is known to be $10\mathrm{ns}$. The Worst Negative Slack WNS of our design was found to be 1.553 ns. Therefore, the frequency was calculated to be 113.062 MHz.

These findings are similar to the Design Resources and Statistics from Lab 6.2. This makes sense because our design was built on top of the code from Lab 6.2.

Conclusion

The Doodle Jump project successfully demonstrates the implementation of an interactive game on an FPGA platform, utilizing SystemVerilog for hardware modules and C for input handling. Key features include collision detection, dynamic platform generation, scoring, and two-player multiplayer support, achieved through modular design and efficient hardware-software co-design. The project overcame challenges such as smooth graphics rendering, real-time synchronization, and resource optimization, showcasing the FPGA's capabilities in handling complex systems. It also highlighted the educational value of digital design, emphasizing modular architecture, timing constraints, and resource management. Future enhancements could include dynamic difficulty levels, improved graphics, online multiplayer, and energy-efficient design for portability. This project serves as a strong foundation for exploring FPGA-based gaming systems and provides a benchmark for further innovation in interactive system design