ECE 385

Spring 2024

Final Lab - TA on Demo Day
(Puzzle-Platform Video Game)

Final Lab Report

Ziad AlDohaim (ziada2)
Mohammed Alnemer (mta8)
For: SH

Table of Contents

Introduction

For our final lab we created a two-player platform game using the Xilinx Spartan-7 FPGA and the MicroBlaze processor. The game features multiple levels, dynamic collision detection, sound effects, and graphical color mapping, offering a comprehensive application of FPGA programming skills in a real-world simulation. The demo can be found on the footnote.[1]

Key Features Implemented:

- Color Detection Collision Algorithm
- Character Collision
- Buzzer sound on death
- Sprites
- Multiple Levels and Players
- Advanced Finite State Machine (Moore)

Additional Features:

- Countdown Timer
- Death and Win Functionality
- Game Backgrounds (Start Screen, Death, Win, Game1, Game2)

[1] https://www.youtube.com/watch?v=YmarbLcq2cQ&ab_channel=ZiadAlDohaim

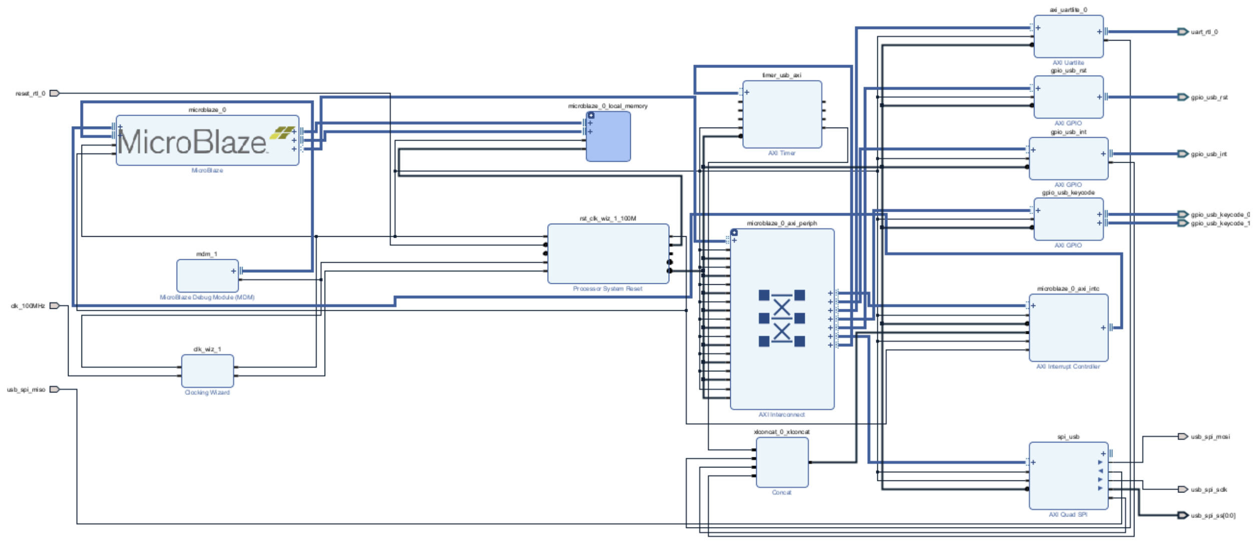


Figure 1: Circuit Design showing mb_block circuit design

.SV Modules

iii) Written Description and Diagrams of Microblaze System

Mb_usb_hdmi_top

Inputs:

Clk, reset_rtl_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, hdmi_tmds_data_n, hdmi_tmds_data_p, hex_segA, hex_segB, hex_gridA, hex_gridB, SPKR.

Description:

This top-level module includes subsystems including USB, UART, HDMI, and VGA for the FPGA hdmi system. It manages data between the FPGA and external interfaces, handles display outputs through VGA and HDMI, and processes user input from USB, UART, and SPKR (sound). Simply put, It controls the overall functionality of applications.

Purpose:

The mb_usb_hdmi_top module is designed as a central hub for managing different input/output interfaces and data streams in an FPGA system, facilitating complex interactions like video processing, audio output, and user input handling.

Changes:
Adapted from lab 6 to create sound, and designed to be highly modular, allowing easy updates and maintenance of individual components like USB handlers, video controllers, and audio outputs.

buzz.sv
Inputs**:**

Clk: Reset, isDead, isDead2

Outputs:
buzz_spkr
Description:
This module generates a sound signal through a speaker based on the game state, particularly when a player is dead. It uses a clock divider to generate an approximate frequency of 1.5 kHz for the sound signal, controlled by the isDead or isDead2 inputs. When either isDead or isDead2 is high, the buzzer is enabled, producing a square wave tone as an audible alert.

The module generates a tone using a simple clock division method, where the system clock (Clk) is divided down to a lower frequency suitable for driving a speaker.
The buzz_en signal is set based on the isDead input, which, when high, enables the generation of the sound. The output buzz_spkr toggles at the frequency determined by the clk_div_counter[12] bit, creating a square wave tone.

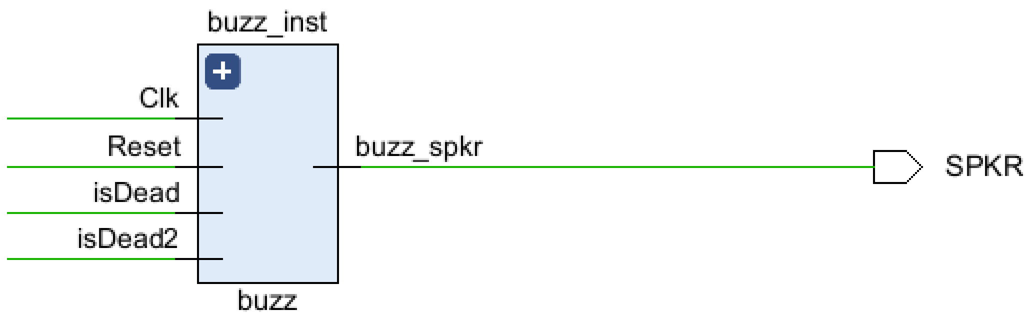


Figure 2: Buzz RTL Design Top View

Purpose:
The buzz module is designed to provide audio feedback during gameplay, enhancing the user experience by audibly indicating significant game events such as the loss of a life.

Changes**:**

This module was adapted to handle two inputs for the death conditions of two players (isDead and isDead2), integrating functionality to respond to events from either player in a two-player game setting.
Initially provided with a framework, the sound generation logic using the clock divider was refined to ensure clear and distinct audio output without affecting the performance of other system components.

color_mapper.sv
Inputs:

BallX, BallY, DrawX, DrawY, Ball_size
BallX2, BallY2, Ball_size2
BallXIntro, BallYIntro, Ball_sizeIntro
flappyX, flappyY, flappyS
BirdX_, BirdS_, SY_
frame_clk
Reset
pix_clk
vde
status

Outputs:

Red, Green, deathsignal
Description:
This module is responsible for determining the color output for each pixel based on the game's dynamic elements. It processes multiple inputs representing various game elements' positions and sizes and maps them to specific colors on the VGA display. The module uses several condition checks to determine which object occupies each pixel and sets the color accordingly.

For each pixel (DrawX, DrawY), the module checks its relation to the positions of the balls and other elements. Depending on which object the pixel aligns with, the RGB color is set to predefined values that represent different elements visually. The module also handles collision detection by setting the deathsignal when certain conditions are met, indicating a game event like a collision.

Purpose:
The color_mapper module enhances the graphical interface by dynamically adjusting the colors of pixels to represent various game elements visually. It provides a clear distinction between different objects and backgrounds, improving the game's visual appeal and player interactions.

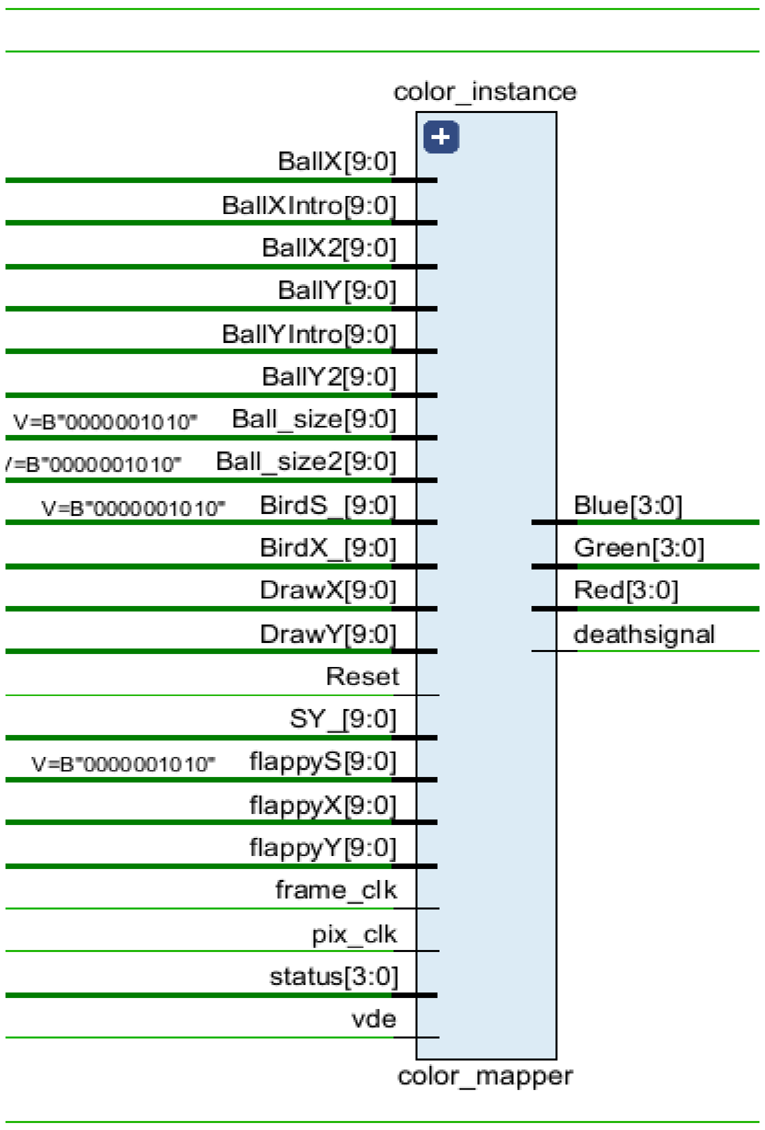


Figure 3: Color Mapper RTL Top View

Changes:

Originally designed to handle a single ball, the module was expanded to include multiple balls and dynamic objects like birds, enhancing the game's complexity and visual dynamics.
Added support for a 'death signal' output to integrate collision detection directly with visual feedback mechanisms.

FSM.sv
Inputs:

Clk,Reset, isDead, isDead2:, isWin, isWin2, isDeadFlappy, isWinFlappy, start,collisiondeath_,keycode[31:0].

Outputs:

status[3:0]: background, win, and lose.
Description:
This Finite State Machine (FSM) module controls the various states of a multi-game system, transitioning between game initiation, gameplay, and end states based on player inputs and game outcomes. It manages states for a platform game and a Flappy game, responding to user interactions and game conditions.

The FSM initializes in the START state and transitions based on user inputs and game conditions. keycode inputs are used to transition out of the start state, while the start input selects the game mode. Game progress, such as wins or losses in any game mode, triggers transitions to respective end states. The FSM updates the status output to reflect the current state, which is used to control game logic and display outputs elsewhere in the system

Purpose:
The FSM module orchestrates the overall flow of the game, managing transitions between different stages of gameplay such as starting, winning, losing, and switching between different game modes. This control is essential for maintaining the correct sequence of gameplay and ensuring that game rules are enforced.

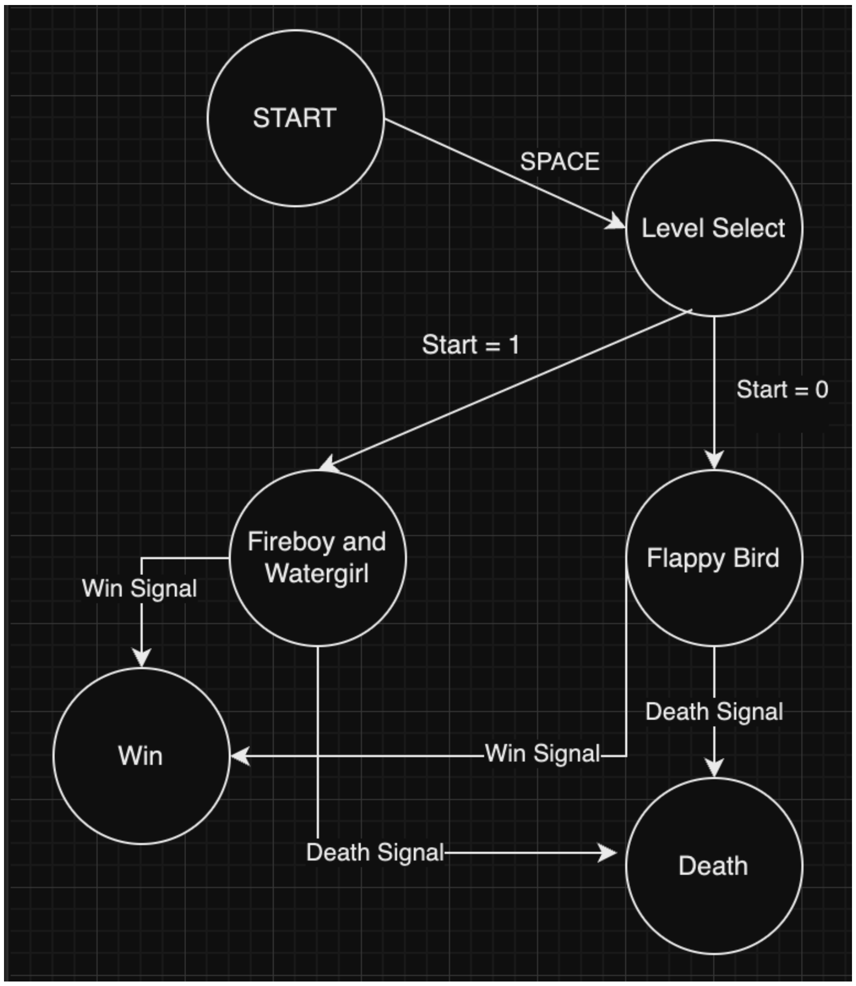


Figure 4: FSM of States

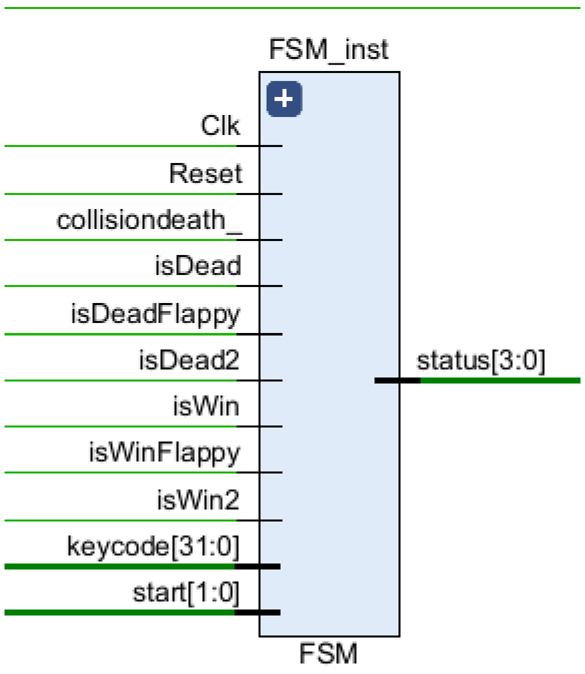
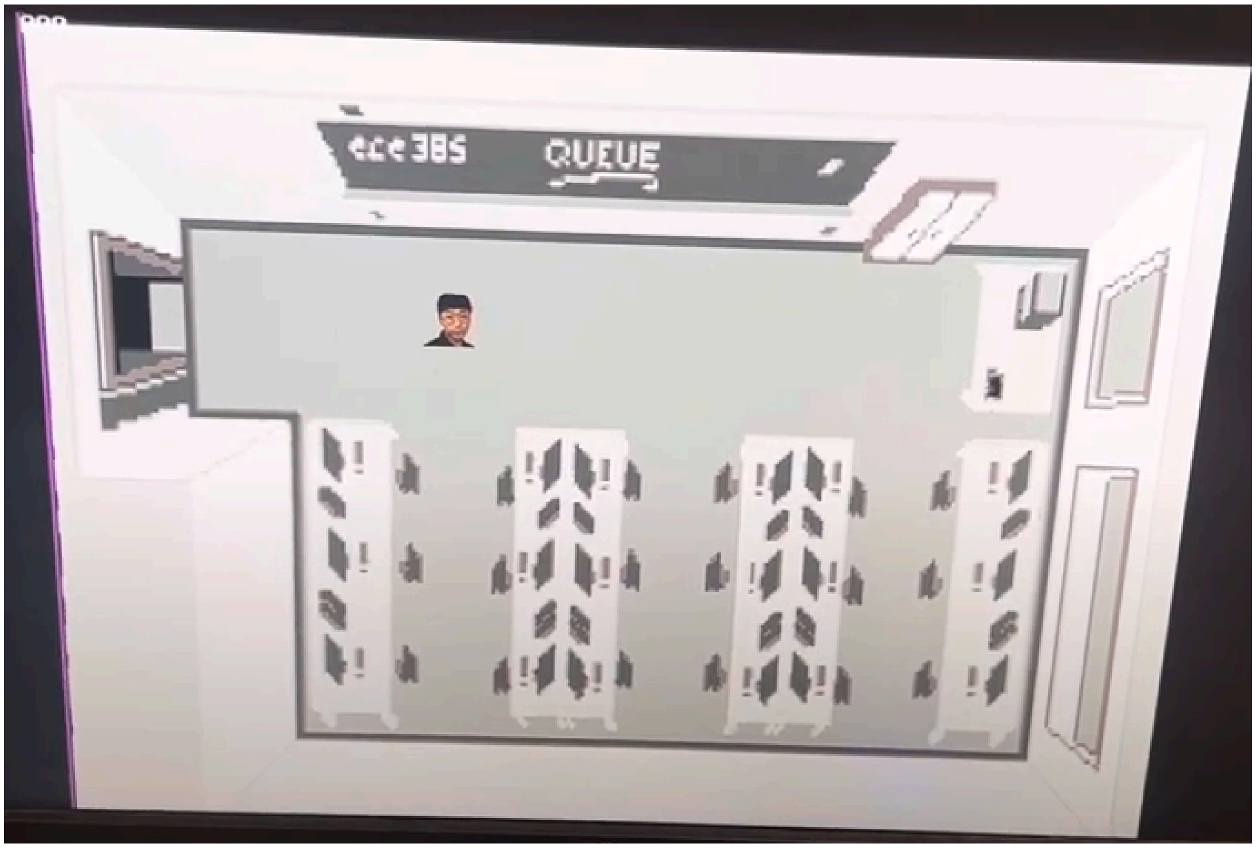


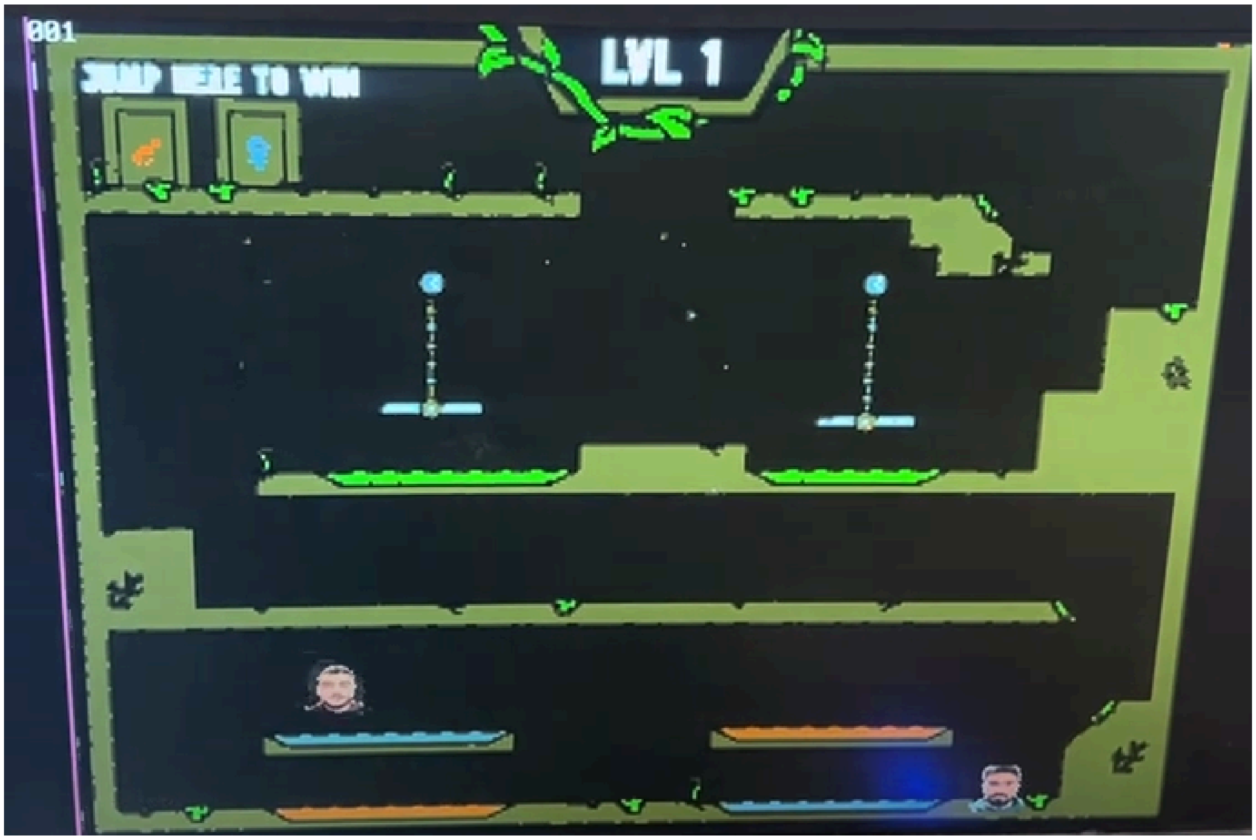
Figure 5: RTL Design of FSM Top View

Level Select:



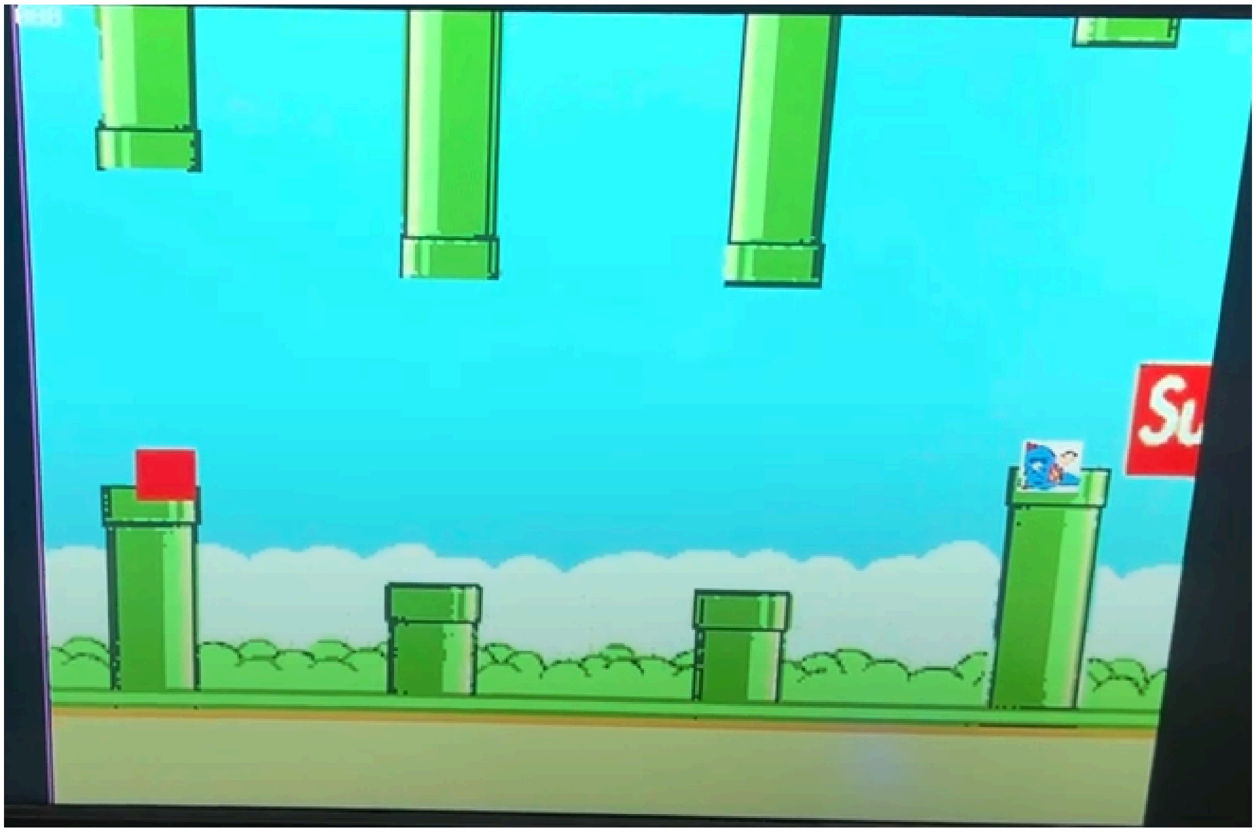
This shows the Level Select State, where the bottom computer goes to the main level (the computer we worked on in eceb 3022) and then the other level is on the top computer (random computer).

FireBoy and WaterGirl:



This shows the fireboy and watergirl level display and game. The two characters can move around with advanced physics and collision logic reacting to the level environment. The game is won by reaching the top left corner of the screen. (This image is captured seconds before fireboy touches water and dies, i.e just jumped up from the floor)

Flappy Bird:



This picture displays our second game display. The Red square can jump on the pipes to escape superman who can fly horizontally from right to left. The goal is to reach the end without being captured by superman and to get the supreme drip.

Changes:

Enhanced to support multiple games by adding specific states and transitions for each game mode.
Integrated collision detection and dual player mode considerations to enrich game dynamics and interaction.

Flappy.sv, Intro Ball
Inputs:

Reset, frame_clk, keycode[31:0], pix_clk

Outputs:

BallX[9:0], BallY[9:0], BallS[9:0],
SX[9:0], SY[9:0], SSize[9:0], BirdS[9:0]
ared[3:0], ablue[3:0], agreen[3:0],
Deadflap,
Winflap

<u>Parameters:</u>

Ball_X_Center, Ball_Y_Center, Ball_X_Min, Ball_X_Max, Ball_Y_Min, Ball_Y_Max, Ball_Size:

Bird_Y_Center, Bird_Size:
<u>Description</u>:
The Flappy module simulates a game where the player controls a ball (or similar object) that must navigate through obstacles. The module reacts to keyboard inputs to move the ball horizontally and jump vertically, checking for collisions with obstacles and boundaries. It also manages game states such as alive, dead, or victorious based on the ball's interactions within the game environment.

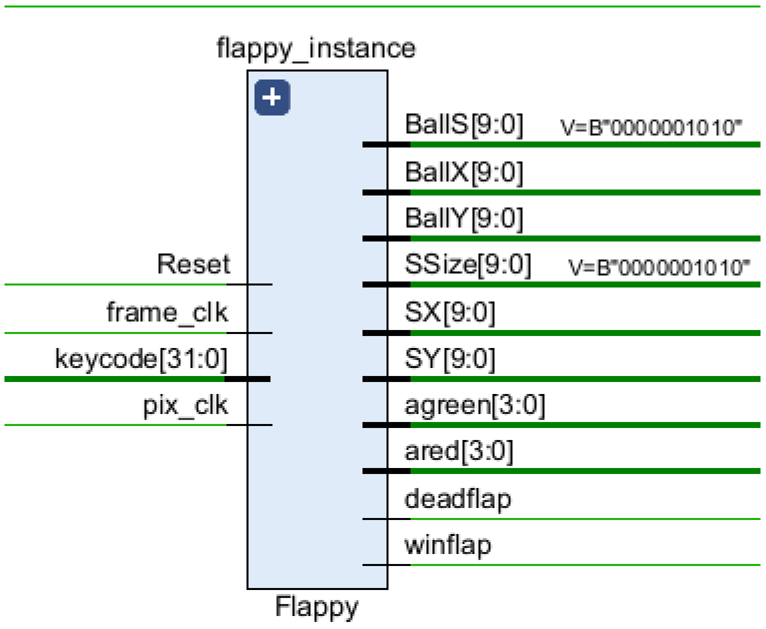


Figure 6: RTL Design Top View of Flappy

Changes:
Adaptations were made to incorporate color outputs directly linked to game events, enhancing visual feedback.
Enhanced collision detection to include more complex interactions with the game's environment and multiple types of obstacles.

ball.sv, ball2.sv, Superman.sv (exactly same description, only variation is naming, including inputnaming)

Inputs:

Reset, frame_clk, keycode[31:0]: pix_clk
Outputs:

BallX[9:0]:BallY[9:0], BallS[9:0]:
ared[3:0], ablue[3:0], agreen[3:0]:
dead2: win2:

Parameters:

Ball_X_Center, Ball_Y_Center:
Ball_X_Min, Ball_X_Max, Ball_Y_Min, Ball_Y_Max:Ball_Size:

Description:
The ball module simulates the movement and interactions of a ball within defined boundaries based on player input. It responds to keyboard inputs to move the ball horizontally and jump vertically, checking for collisions with boundaries to update game states such as win or loss.

On reset, initial positions and states are set for the ball.
The module calculates the ball's new position based on user inputs from the keyboard and the game's physics, adjusting its X and Y coordinates accordingly.
Collision detection checks if the ball has hit any boundaries or obstacles, updating the dead2 or win2 states based on these events.

The gravity of the balls worked using an a state diagram shown below, it seems simple however, a lot of logic goes on in each state to ensure correct collision checking (ground and/or ceiling), gravity speed, speed of jump, win and death signals.

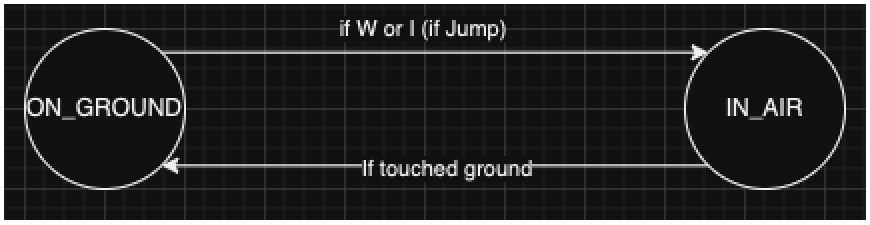


Figure 7: Finite State Machine of Gravity

Purpose:

The ball module is designed to manage the dynamics of a ball, handling physical interactions and game-related events within a confined space on an FPGA platform.

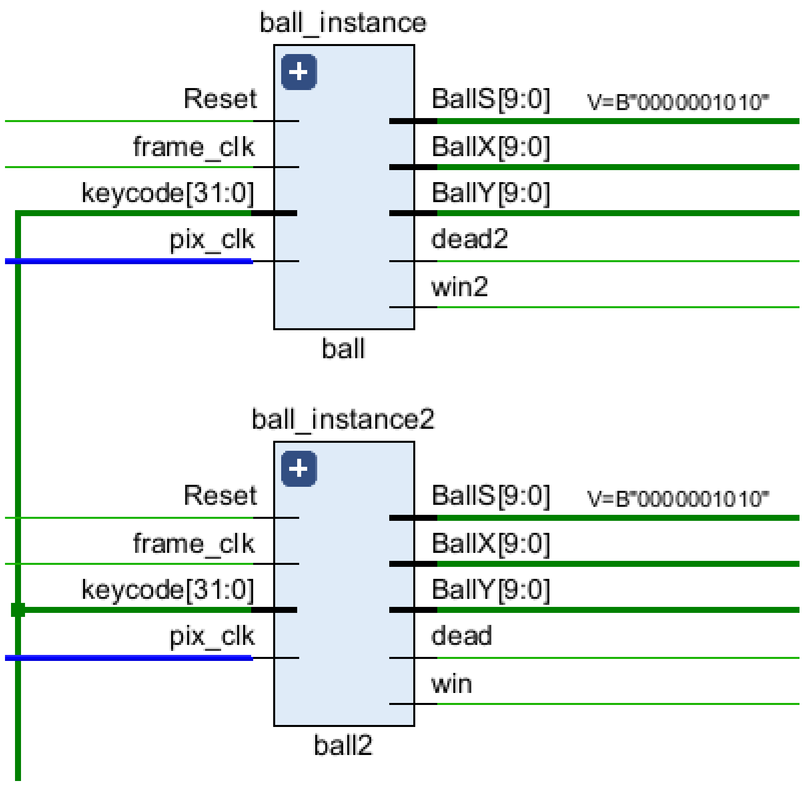


Figure 8: RTL Design Top View of Ball 1 and Ball2

Changes:

Taken from lab 6, however, we enhanced motion handling to include complex interactions with the environment, such as detecting collisions with multiple types of boundaries and updating game states. Adaptations were made to the module to handle additional game logic related to winning conditions and player deaths.

Intro_Ball.sv
Inputs:

Reset: Synchronous reset signal to initialize the module.
frame_clk: Frame clock signal used for timing and controlling the simulation.
keycode[31:0]: 32-bit input used to control the ball's movement via user input.

Outputs:

BallX[9:0]: X-coordinate of the ball, determining its position on the horizontal axis.
BallY[9:0]: Y-coordinate of the ball, determining its position on the vertical axis.
BallS[9:0]: Size of the ball.
enter_signal[1:0]: Signal used to indicate game transitions or specific actions triggered by the ball's position.
Parameters:

Ball_X_Center, Ball_Y_Center: Initial central position coordinates of the ball.
Ball_X_Min, Ball_X_Max, Ball_Y_Min, Ball_Y_Max: Define the minimum and maximum bounds of the ball's movement.
Ball_Size: The diameter of the ball, affecting its collision and rendering.
Description:
The Intro_Ball module is designed to manage the introductory interaction of a ball within a graphical display, particularly for initial screen animations or menu interactions in FPGA-based gaming or simulation applications. It uses input from a keycode to determine the ball's movement and can signal transitions such as starting a game or switching menus based on the ball's position and user input.

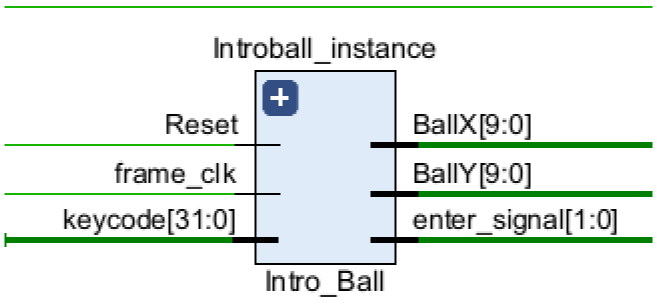


Figure 9: RTL Design Top View of introball

Purpose:
This module acts as an interactive introduction controller, providing users with immediate visual feedback through ball movement and helping them navigate the initial options or states of an FPGA-based system. It enhances user experience by integrating simple controls with dynamic visual elements.

Changes: This ball was taken from lab 6, and optimized to fit the intro screen, i.e. it just moves up and down within new constraints.

RTL DESIGN

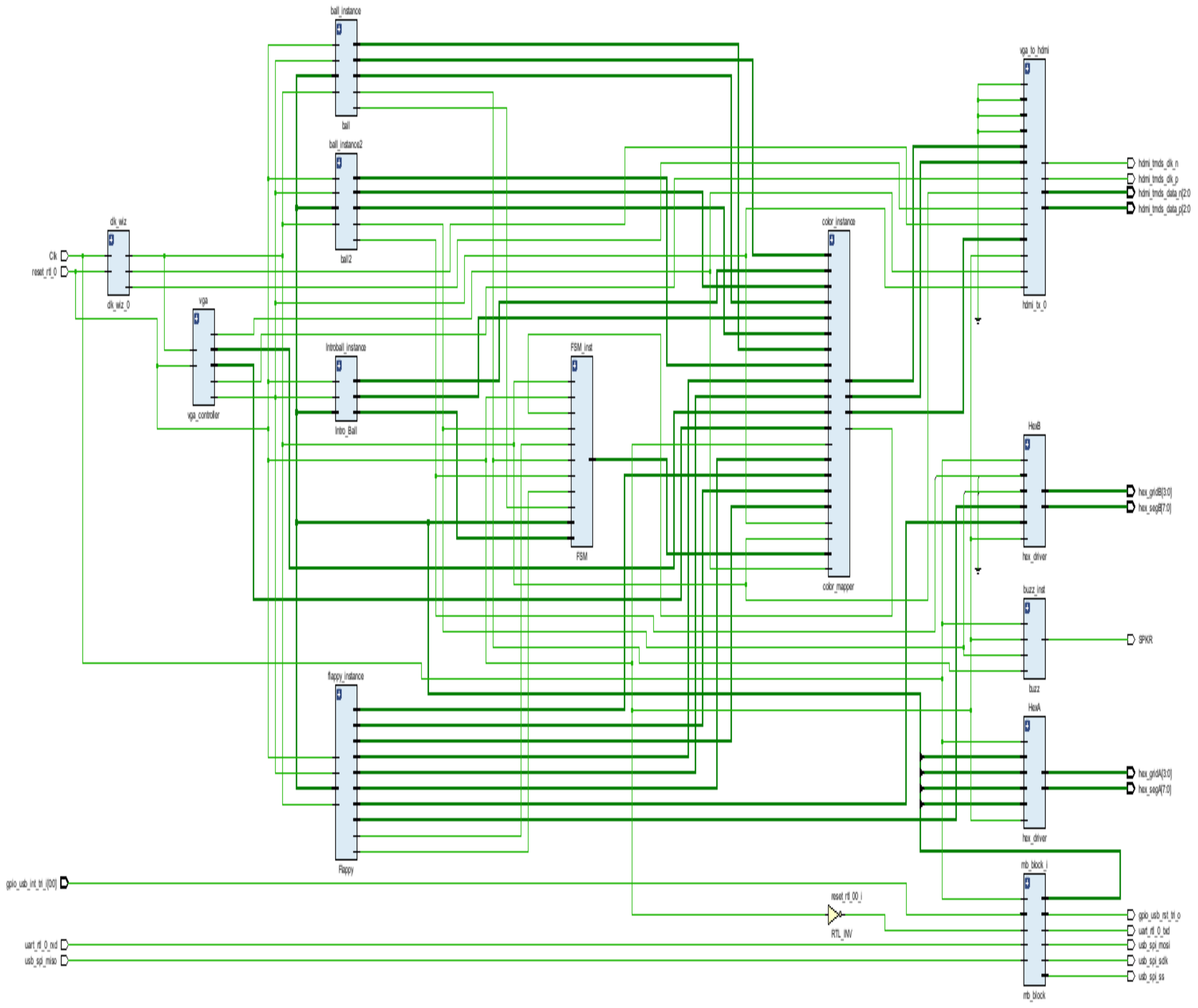


Figure 10: RTL Top View of Entire Lab.

Core Components:

<u>Microblaze:</u>

<u>Purpose</u>: Acts as the CPU within Xilinx FPGA systems, executing a wide range of software applications and algorithms.
<u>Operation</u>: It is configurable and can be tailored to specific needs, running everything from simple control tasks to complex processing algorithms directly on the FPGA.

<u>Microblaze Local Memory:</u>

<u>Purpose</u>: Provides fast, directly accessible memory for the Microblaze processor, critical for storing temporary data and instructions.
<u>Operation</u>: Integrated closely with the Microblaze core, it facilitates rapid data access and storage, ensuring high performance of applications by minimizing access delays.

<u>Microblaze Debug Module (MDM):</u>

<u>Purpose</u>: Allows to debug software on the Microblaze processor by providing mechanisms for inspection and control of execution.
<u>Operation</u>: Offers features like breakpoint setting, program stepping, and system reset, allowing for detailed observation and troubleshooting of software behaviors.

<u>Clocking Wizard:</u>

<u>Purpose</u>: Generates and manages the clock signals required for the operation of various components within the FPGA.
<u>Operation</u>: Utilizes external clocks to produce multiple derived clock signals, ensuring that different parts of the system can operate synchronously or at required frequencies.

<u>Processor System Reset:</u>

<u>Purpose</u>: Ensures that all components in the system are correctly initialized and synchronized by managing their reset sequences.
<u>Operation</u>: Can handle multiple reset signals, coordinating the reset process across various modules to ensure a clean start-up or recovery from errors.
<u>AXI Interconnect:</u>

Purpose: Acts as the main artery for communication between the Microblaze processor and peripheral devices, ensuring efficient data transfer and control signal distribution.
Operation: Manages data paths between master and slave devices, allowing for multiple concurrent data transfers and providing flexible bandwidth allocation.

AXI Interrupt Controller:

Purpose: Manages interrupt signals within the system, prioritizing and dispatching them to the Microblaze processor.
Operation: Aggregates multiple interrupt sources, prioritizing them based on system requirements and ensuring that critical interrupts are handled promptly.

AXI Uartlite:

Purpose: Provides a simple serial communication interface for the system, enabling data exchange with external devices or for debugging purposes.
Operation: Implements a UART protocol stack, managing data transmission and reception over serial connections.

AXI GPIO:

Purpose: General-purpose input/output interfaces, allowing the FPGA to interact with a range of external digital devices.
Operation: Can be set up as either input or output ports, enabling bidirectional communication with external hardware like sensors, switches, and LEDs.

Additional Components (Week 2):

Concat:

Purpose: Facilitates signal processing within FPGA designs by allowing multiple signals to be combined into a single, broader signal.
Operation: Merges several input signals into one wider output signal, simplifying signal management and supporting complex data structures.

AXI Timer:

Purpose: Provides timing and delay functions, crucial for operations requiring precise timing or scheduling.

Operation: Can be used for generating precise time delays, measuring time intervals, or scheduling events within the FPGA system

AXI Quad SPI:

Purpose: Enables high-speed serial communication with external memory and peripherals using the Quad SPI protocol.
Operation: Supports both standard and quad SPI modes, allowing for rapid data exchange with external SPI devices, enhancing system capabilities with additional memory or interfacing options

Background Color Detector collision method

Purpose:
The Background Color Detector Algorithm is designed to automatically identify and classify the color in the background of a visual scene. This functionality is crucial for the collision detection in our lab.

Operation:
The algorithm processes input image data to calculate the most frequently appearing color values, using nearest neighbor algorithm to detect colors above and below using a dual port rom. It supports real-time operation, allowing continuous assessment and adjustment in response to changes in the visual input. Essentially, this collision method could be used to make new levels on the spot, while also being RAM efficient due to the low resolution of the images.

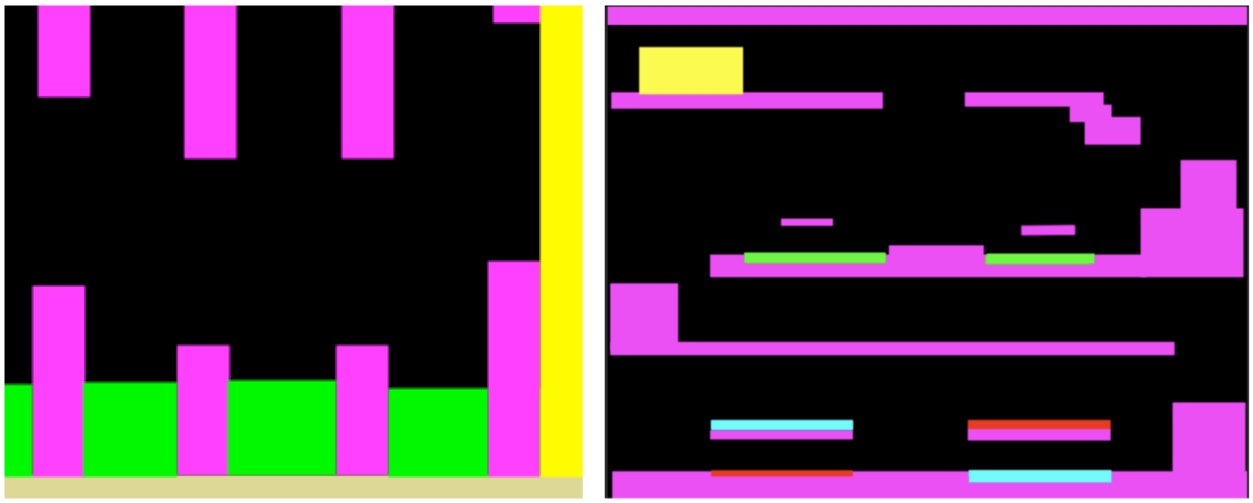


Figure 11: Images highlighting the Background color detector collision method.

<u>Countdown Timer</u>

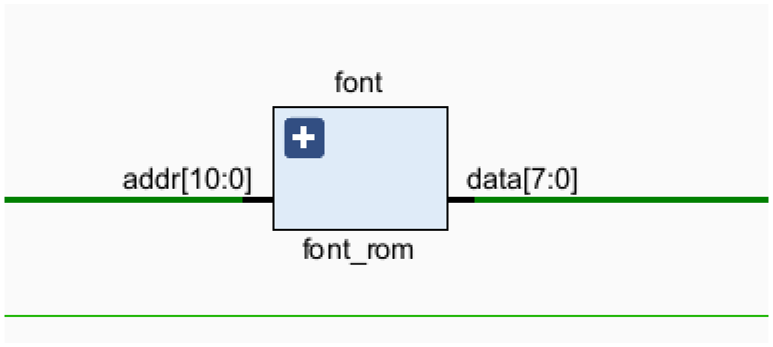
<u>Purpose:</u>

The Countdown Timer is integrated into the top left corner of our game, to provide players with a clear and constant update on the remaining time for a given level or session. When the timer reaches zero, the game transitions to a 'death state', indicating that the player has failed to complete the objectives in the allotted time, increasing the game's difficulty.

<u>Operation</u>:

This timer continuously counts down during gameplay, visually decrementing the displayed time at regular intervals. We display this through the font_rom.

Figure 12:



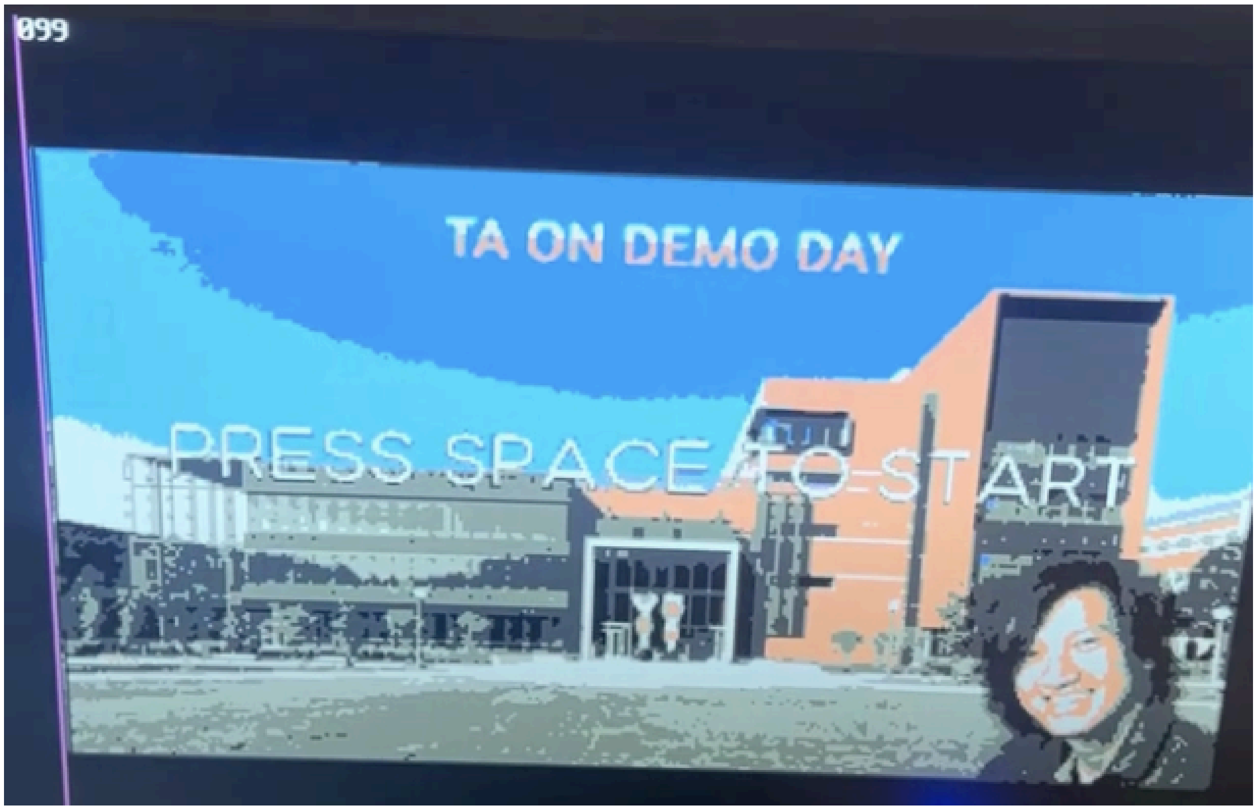


Figure 12: Intro screen showcasing timer on top left

Functions in the C Code (taken from lab 6 and not modified)

MAXreg_wr:
Parameters: BYTE reg, BYTE val
Functionality: This function writes a single byte val to a register reg on the MAX3421E chip. It's responsible for configuring specific settings or controlling various operations of the MAX3421E. The function selects the MAX3421E for communication via SPI, sends the register address offset by 2 to specify a write operation, writes the value to the specified register, checks for successful transmission, and then deselects the device.

MAXbytes_wr:
Parameters: BYTE reg, BYTE nbytes, BYTE* data
Functionality: This function writes nbytes bytes of data from the data array to the MAX3421E, starting at register reg. It's useful for sending a sequence of commands or a block of data. The function follows a similar process as MAXreg_wr but iterates through the data array, sending each byte consecutively after setting the register address.

MAXreg_rd:
Parameters: BYTE reg
Functionality: MAXreg_rd reads a single byte from a specified register reg on the MAX3421E. The function is utilized to check the current status or configuration of the chip. It performs an SPI transfer where it sends the register address and reads back the value into a buffer, which is then returned.

MAXbytes_rd:
Parameters: BYTE reg, BYTE nbytes, BYTE* data
Functionality: Similar to MAXreg_rd, but for reading multiple bytes. It reads nbytes bytes from the MAX3421E starting at register reg and stores them in the data buffer. This is particularly useful for reading out longer data structures like configuration settings or buffer contents from the MAX3421E.

Post Lab Questions:

Final Project Statistics table

LUT	5587
dsp	26
BRAM	51.5
Flip-Flops	3050
Latches	0
Frequency	90.596 MHz
Static Power	0.078 W
Dynamic Power	0.459 W
Total Power	0.537 W

Conclusion

This project not only reinforced our understanding of FPGA programming and the MicroBlaze architecture but also highlighted the importance of system integration and testing in embedded system design. Future improvements could include more complex levels, enhanced graphics, and multiplayer functionality over a network.

We faced many difficulties while implementing this final project. The biggest challenge we faced was getting the color tracking algorithm to work, we had many discrepancies within calculations, since we had to extend the ROM to a dual-rom port rather than the standard single port rom. Additionally, we faced issues getting the gravity to work properly, however, after creating a state diagram and with the help of the office hours we got it to work. Moreover, getting the sprites to move with the ball module was quite difficult and needed some math to get fixed. Furthermore, we did have some issues with BRAM optimization to deal with the shortage, we significantly reduced the resolution and scaled up the image to fit and be memory efficient. Lastly, an issue we faced was with the limitations of the keyboard, we could not use the arrow keys simultaneously with W A S D, so we decided to move from the standard arrow keys and use I J K L instead to ensure smooth gameplay. Thankfully, after a lot of effort and help from office hours we were able to get the final project to work smoothly.

Something we really found was important with this is debugging while using the hex drivers, it was the most important discovery we had and allowed us to complete this project, it's like a DebugCore in real time.