FINAL PROJECT REPORT

ECE 385

By

Tianyang Wu

Yue Shi

I wanna Game, Spring 2024

TA: Jinghua Wang

1 May 2024

University of Illinois Urbana-Champaign

Abstract

Our project is centered around the development of an 'I wanna' series game on a Spartan-7 Urbana FPGA board. This platform serves as the foundation for a game that intricately weaves complex logic for game mechanics and control, as well as sound effects for an immersive experience. The game logic, a cornerstone of our project, is responsible for handling player movements, obstacles, and game state transitions, all of which are instrumental for gameplay dynamics. Equally significant is the sound effect logic, which operates in sync with the game logic, enriching player interaction and feedback.

Contents

1. Introduction **4**
2. Written Description of Final Project System **4**
2.1 Video output section 4
2.2 Game logic section - game states 6
The control of game state includes a finite state machine, its structure is shown as below: 6
2.3 Game logic section - game object movement 6
2.4 Game logic section - collision check 7
2.5 Audio effect logic 7
2.6 Audio PWM 8
3. Block Diagram **9**
Figure 2: Block Diagram of Final Project 9
3.1 MicroBlaze 9
3.2 Clocking Wizard 9
3.3 Processor System Reset 9
3.4 MicroBlaze Local Memory 9
3.5 AXI Interconnect 10
3.6 AXI Interrupt Controller 10
3.7 AXI Quad SPI 10
3.8 AXI Uartlite 10
3.9 AXI GPIO 10
3.10 Concat 10
4. Written Description of Modules **11**
4.1 Module: background_color_mapper 11
4.2 Module: background 11
4.3 Module: man_color_mapper 11
4.4 Module: man_left_palette 11
4.5 Module: man_dead_palette 12
4.6 Module: man 12
4.7 Module: spikes 12
4.8 Module: spike_color_mapper 12
4.10 Module: apples 12
4.11 Module: apple_color_mapper 12
4.12 Module: game_over_word 12
4.13 Module: over_color_mapper 13
4.14 Module: color_object 13
4.15 Module: game_state 13
4.16 Module: game_state_check 13
4.17 Module: spike_location_updator 13
4.18 Module: win_title 13
4.19 Module: win_color_palette 13
4.20 Module: pwm_top 13
4.21 Module: audio_rom 14
4.22 Module: pwm_core 14

5. Software Design **14**

The C workspace setup is exactly the same as LAB 6.2 as we only requires keycode passed from the keyboard. **14**

5.1 MAXreg_wr(BYTE reg, BYTE val) 14

5.2 MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) 14

5.3 MAXreg_rd(BYTE reg) 14

5.4 MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) 14

6. Simulation Waveforms **14**

7. Design Resources and Statistics **15**

8. Conclusion **16**

1. Introduction

Our project aims to develop and showcase a fully operational version of the "I wanna" game, implemented on an AMD FPGA board utilizing the Microblaze processor.

In Lab 6.2, we implemented a controllable ball on-screen using keyboard inputs, building upon the VGA controller and color mapper file. Our final project evolved this setup by expanding the scope to a full game environment. We introduced a character that players can control by keyboard and incorporated obstacles like spikes and apples. The game logic, developed on the FPGA, manages collision detection, movements, and colors, enhancing the interactive experience. We also integrated sound effects triggered by specific game events—such as jumping, hitting obstacles, or reaching the endpoint—to enrich the gameplay and provide real-time auditory feedback.

2. Written Description of Final Project System

In order to complete the whole project, we decide to develop based on Lab 6.2. The whole project will have three major parts: audio output section, video output sectioin and game logic section. The game logic part includes several important part that controls whether the game should freeze or move on, movement of the game objects, and collision check of game objects. Here the description focuses on more on the whole picture, some of the details for combinational logic(eg. various coordinate comparisons) are omitted because including them will make this description too verbose, those details can be check in code repository. The overall block diagram of the project will be shown as following:

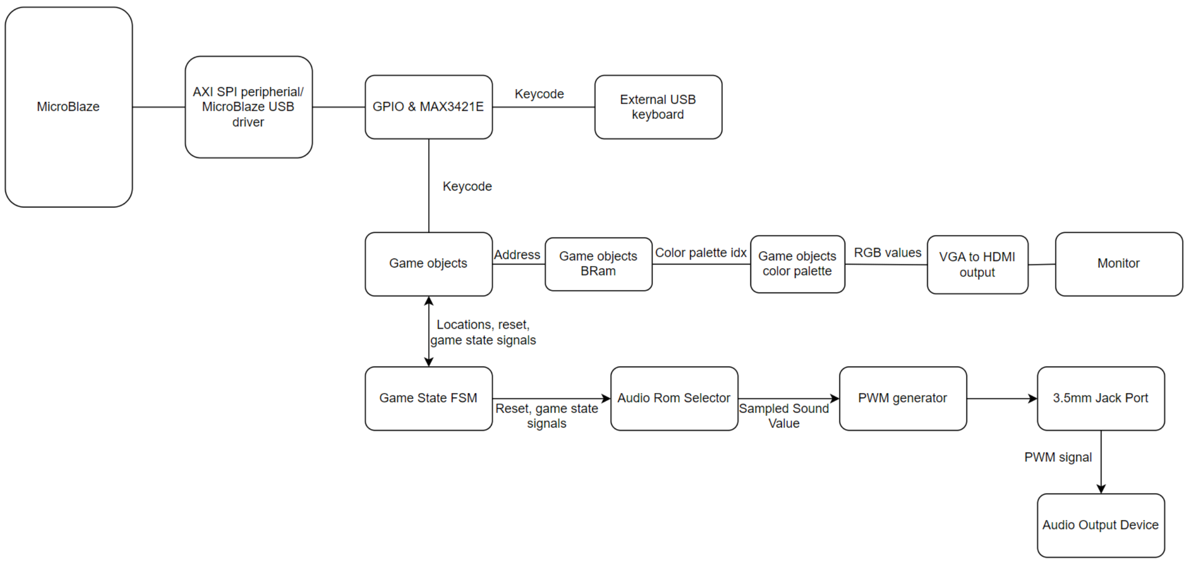


Figure 1: Overal Block Diagram

2.1 Video output section

From lab 6.2, we have the capability to assign video RGB(4 bit value per pixel) signal per pixel according to the drawX and drawY (drawing pixel location) to output video signal. We modified the provided VGA signal module to make the screen resolution to be 480*360 pixels. All we need to do is to design logic to output correct RGB values given the drawing location and the location of each game object.

Before discussing how to cooperate different game objects, we need to findout the RGB values corresponding to each game object. This requires us to findout both the relative pixel location(which pixel in the range of the game object) and overall drawing location(on the whole screen). Based on memory limitation, we restrict the whole screen resolution to be 480*360 pixels, and use a 16-color color palette(4 bit per pixel in memory, stored in little endian format), notice that for better graphics performance, each game object has its own color palette. This can be done by initializing several BRam modules and load in pre-processed image content(as .COE file processed by Image_to_COE github project), and then calculate the corresponding address and offset to get the color palette index(palette is also calculated by Image_to_COE github project), and finally transfer to RGB values. Address and index calculation will be shown in the following part.

Denote the top-left pixel as loction (0,0). Similarly, we denote the top-left pixel of each game object to be its own coordinate reference(denote as (object_x,object_y) here for simplicity). Also, denote the height and width of each game object as (object_width and object_height). We used a 32-bit memory bandwidth BRam block and store 4-bit information for a pixel(8 pixels in an address space), then the following calculation holds:

To check whether the drawing location is within the object:

object_x<=draw_x<object_x+object_width && object_y<=draw_y<object_y+object_height

To derive the address of pixel information if drawing location is whin the object:

pixel_idx=(draw_x-object_x)+(draw_y-object_y)*object_width

address=pixel_idx>>3

To derive the offset of the exact pixel information in the address space:

color_palette_idx=memory[address+4+8*pixel_idx%8:address+8*pixel_idx%8]

Then we just need to extract the corresponding R,G,B values in the object color palette with this color_palette_idx.

The game objects include background, game character(alive and dead state, alive state has facing left and right states), spikes(first type of trap), apples(second type of trap), win title and lose title. The drawing layer relation is as follows(from top layer to bottom layer):

win title=lose title>spikes>appples>game character>background

Those game objects have following dimensions:

win title:	266*64	pixels
lose title:	320*64	pixels
spikes:	20*20	pixels
apples:	16*16	pixels
game character:	20*20	pixels
background:	480*360	pixels

Given this relation, we can than easily design logic to correctly assign the RGB value according to the object layer. Notice that for some game objects(win/lose titles, game character, apples, and spikes), they have white or black background(background has different RGB values to all other colors), if we are drawing this background color, we should take the RGB value of the next layer.

2.2 Game logic section - game states

The control of game state includes a finite state machine, its structure is shown as below:

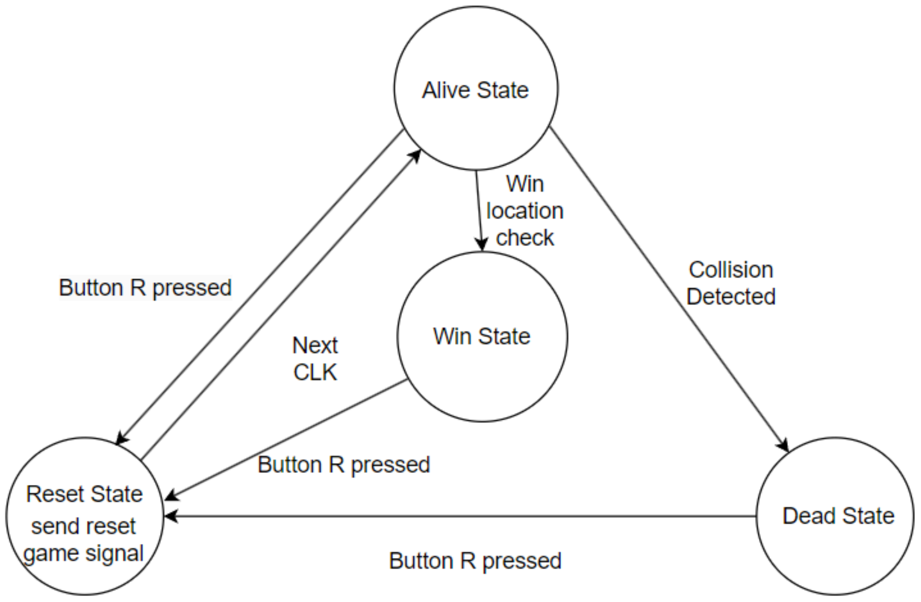


Figure 2: Diagram of Game Control State

This finite state machine simply freezes the game when game character collide with traps or moves to the winning location until button "R"(used for reset) is pressed. Once this button is pressed, the location of game objects will be reset and then shift back to alive state. Also, whenever we are in the win or dead state, the corresponding game title will be printed on screen and the correspond game sound will be triggered.

2.3 Game logic section - game object movement

The movement of game object includes two part: the movement of spikes and the movement of the game character. The movement of the game character needs to align with the input from keyboard. The character's movement is controlled by keyboard inputs and is subjected to game constraints such as boundaries, ground detection, and other environmental interactions like jumping and collision handling.In this project, we used two key input(pressing two keys simutaneously) to control the game character, the keycode is from LAB 6.2 setup(allowing at most 6 keycodes at the same time). For moving left and right, we just simply add the velocity to the character locationThe character moves left or right when 'A' or 'D' are pressed, respectively. The movement is controlled through incremental changes in the character_x location variable. Collision with walls prevents movement in the respective direction, ensuring the character does not pass through barriers.The 'W' key initiates a jump if the character is on the ground (on ground check is true). The jump is controlled by a counter (jumping_frame_count=40 frames) that modifies the vertical speed over time to simulate the physics of a jump. The character experiences gravity, represented by a constant downward adjustment to the character_y unless the character is on a surface or hits a ceiling (wall_above check is true), which stops the vertical movement.

The game logic includes conditions for landing on platforms at different heights and for bouncing off the edges of the game area. Specific X and Y coordinates check for landing platforms, and adjustments are made to prevent the character from falling through platforms. Also, there are combinational logic that determines whether there are walls above/right/left to the character based on current character position. There is also combination logic that checks whether the character is on the ground. Those environmental parameters check outputs signals that will be used in game character movement control. Also, The character's position and motion are reset when a Reset signal is triggered, setting the character back to a predefined center position. Additional checks for game states like winning or death halt movement and maintain the character's current position to reflect the game's end conditions.

To the movement of spikes(trap), we will use arrays to store and indicate the location and direction of spikes since there are multiple traps in the games(compare to only one game character). Since the spikes need to start moving when the game character moves to a specific location, we will have a triggered array that specify whether a spike is already triggered or not. Spikes are initially inactive (Triggered is set to false). They become active (triggered) based on the player character's position (character_x, character_y). For example, specific spikes are triggered when the player reaches certain X-coordinates, which simulates the spike activation when a player approaches or steps on a triggering mechanism. Once triggered, spikes move at a defined velocity (1 pixel per frame). The movement is typically vertical (up or down), and this is controlled by incrementing or decrementing their Y-coordinate (SpikeY). The direction of movement is governed by the initially set direction array. To prevent the spikes encountering overflow when moving out of the screen, we want to stop the spike movement once it reaches a boundary (e.g., SpikeY[i] > 359 or SpikeY[i] == 0), which prevents spikes from moving beyond the visible game area. This boundary handling ensures that spikes reset or stop at logical points, maintaining game integrity and challenge.The spike behavior is also influenced by the game states such as dead or winning states. When the player character is marked as dead or winning, the spikes will stop moving.

2.4 Game logic section - collision check

The collision between game character and spike involves checking specific points inside the game character and upward/downward spikes. When a spike is in upward orientation, the system checks a series of coordinates horizontally across the width of the character and vertically within the spike's height. The checks involve comparing the character's lower boundary (character_y+19, representing the bottom of the character sprite) against a series of positions extending from the base (SpikeY[i]) to the top (SpikeY[i]+19) of the spike.This is because that spike is a slit side with zigzags. Each check sees if the character overlaps with the spike at increasingly larger increments horizontally (character_x+15 to character_y+10 are additional checks positions for slight horizontal shifts, which means there are total 3 collision check positions on character). Similarly, The conditions invert when dealing with downward-oriented spikes. Here, the checks are similar but focus on whether the character's top part intersects with the lower parts of the spikes. Similarly, the collision between apple and character is realized but with different shapes for apples, the check position on character remains the same.

2.5 Audio effect logic

The audio system is designed to handle multiple sound triggers, specifically catering to game-related events, such as jumps, deaths, and wins. Each sound effect is stored in a dedicated Block memory within the FPGA, enabling quick retrieval and playback.

Memory usage:
Jump Sound: Stored using 8-bit depth in a memory block sized at 6,656 bytes, this sound effect is triggered by player jumps, adding a dynamic element to the gameplay.

Dead Sound: With an 8-bit depth and occupying 26,432 bytes, this sound signifies the player's failure or death in the game, providing immediate feedback on critical game events.

Win Sound: The largest of the three, using 69,888 bytes, this sound plays upon winning the game, marking significant achievements within the game.

Sampling rate: 44100Hz
Sampled sound is converted from a .wav file to a .coe file, which is browsed by a Block memory. The content in the .coe file is in Hex, which represents the pulse width.

The core of the sound system is the pwm_top module, which handles the playback logic. It includes mechanisms to prioritize which sound to play based on the input triggers. For instance, sound related to dead or win, takes precedence over the jump sound, ensuring that critical game feedback is delivered to the player without delay. This prioritization is managed through a combination of conditional logic(playback_complete) and state management, ensuring that once a special sound is playing, it is not interrupted until complete.

The prioritization system checks if a special sound related to death or winning is currently active through a flag(special_playback_active). If not, it checks for new triggers(Jump). If a sound is playing, it ensures this sound is not interrupted until it completes, allowing for a full, uninterrupted audio experience of crucial moments.

The state management involves managing the playback states using the playback_complete signal. This signal helps ensure that once a sound is triggered, it is played back entirely before another sound can interrupt unless it is overridden by another sound with a higher priority based on the game's logic and needs.

2.6 Audio PWM

The pwm_core, which is provided by Real Digit, within the system modulates the width of pulses in accordance with the audio data provided, effectively converting digital audio signals into analog signals through low-pass filtering on the output lines. The main component of the pwm_core module is a counter. This module operates by continuously comparing a counter value against a dynamically set pulse width value, which is derived directly from audio data. This mechanism allows it to generate a square wave signal with a duty cycle that directly corresponds to the audio signal's amplitude at any given moment.

Period: 8'd255

3. Block Diagram

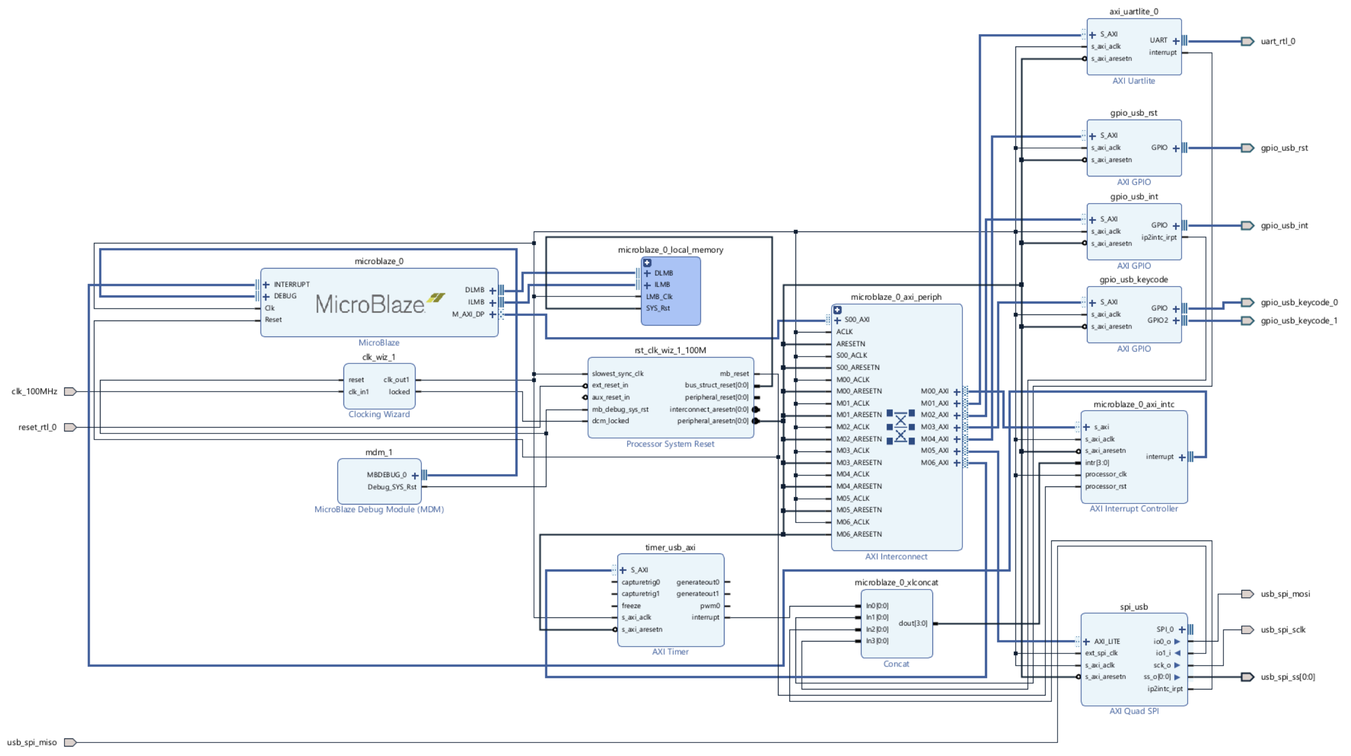


Figure 3: Block Diagram of Final Project

3.1 MicroBlaze

MicroBlaze is a 32-bit, soft-IP microprocessor utilizing a modified Harvard RISC architecture that is configurable to meet specific application needs through adjustable instruction and data cache sizes and various peripherals. Primarily serving as the system controller in our final project, it processes commands and handles non-high-performance tasks.

3.2 Clocking Wizard

The Clocking Wizard generates clock signals at specific frequencies and phase relationships to ensure synchronization among various components. It accepts a clock signal input and distributes it to connected components. It can also convert a 100MHz input signal into two outputs, 25MHz and 125MHz, specifically for HDMI synchronization.

3.3 Processor System Reset

The Processor System Reset module controls the reset signals for the MicroBlaze processor and other components, ensuring that the system boots up in a uniform state. It provides a feature for users to customize the reset function, allowing them to select between active high or active low reset configurations.

3.4 MicroBlaze Local Memory

The MicroBlaze processor uses its integrated on-chip local memory to store program code, data, and stack memory.

3.5 AXI Interconnect

The AXI Interconnect is designed in FPGA for fast and flexible communication between different parts like processors, memory, and other devices. It manages data flow, allows multiple devices to access data at the same time, and supports various sizes and types of data. Essentially, it allows multiple components to talk to each other at the same time.

3.6 AXI Interrupt Controller

The AXI Interrupt Controller manages interrupt signals within the system. This controller consolidates multiple interrupt requests from various peripherals and channels them to the MicroBlaze processor. It allows the processor to respond to external inputs effectively and simplifies the handling of interrupts, streamlining the system's operation.

3.7 AXI Quad SPI

The AXI Quad SPI module allows FPGAs to communicate quickly with SPI devices using the AXI interface. It supports fast data transfer over four channels simultaneously, making it great for accessing flash memory or streaming data at high speeds. This helps digital systems exchange data rapidly and efficiently using SPI protocols.

3.8 AXI Uartlite

The AXI UARTlite module is designed to facilitate UART communication within the AXI bus framework, functioning as a bridge between the FPGA and external peripherals. AXI UART offers a simplified solution for serial data transfer and enables asynchronous communication.

3.9 AXI GPIO

The AXI GPIO acts as a bridge connecting GPIOs within an FPGA system. It serves as an interface for external digital I/O devices, enabling the reading of inputs and control of outputs for interaction with the external environment. Through the AXI bus, this component enables direct alteration of pins, providing adaptable, software-managed interaction with external components or on-chip signals.

3.10 Concat

The Concat IP core is designed to merge multiple input signals, including data and control signals, into a single output. This is especially useful for simplifying interrupt signals into one line for easier management in a single bus or register.

4. Written Description of Modules

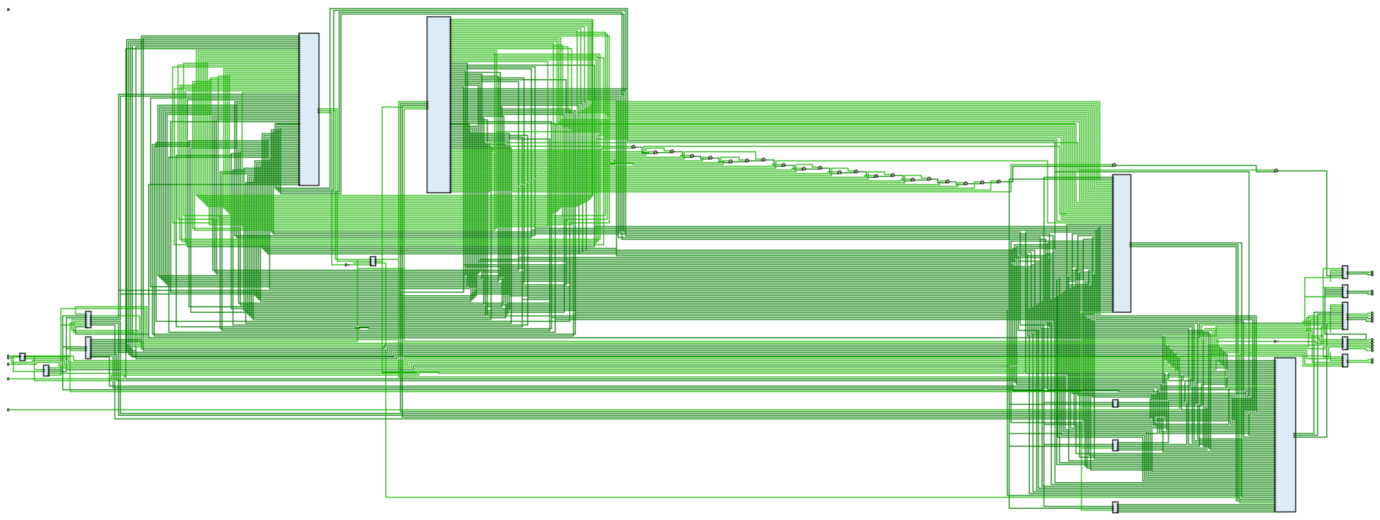


Figure 4: Schematic Diagram of Modules

4.1 Module: background_color_mapper

This module is designed for FPGA-based video applications. It maps packed background data to RGB color outputs based on pixel position. It accepts drawing coordinates and a 32-bit packed color data input, determining the specific color index for each pixel using the horizontal coordinate. The outputs, Red, Green, and Blue, are 4-bit values representing the pixel color at the given coordinates.

4.2 Module: background

This module generates the visual background for FPGA-based graphics applications. It calculates memory addresses from pixel coordinates, which are DrawX and DrawY, fetching color data from a block memory. The module outputs these colors Bkg_Red, Bkg_Green, and Bkg_Blue for display, operating at a 125 MHz clock frequency to ensure smooth visual updates.

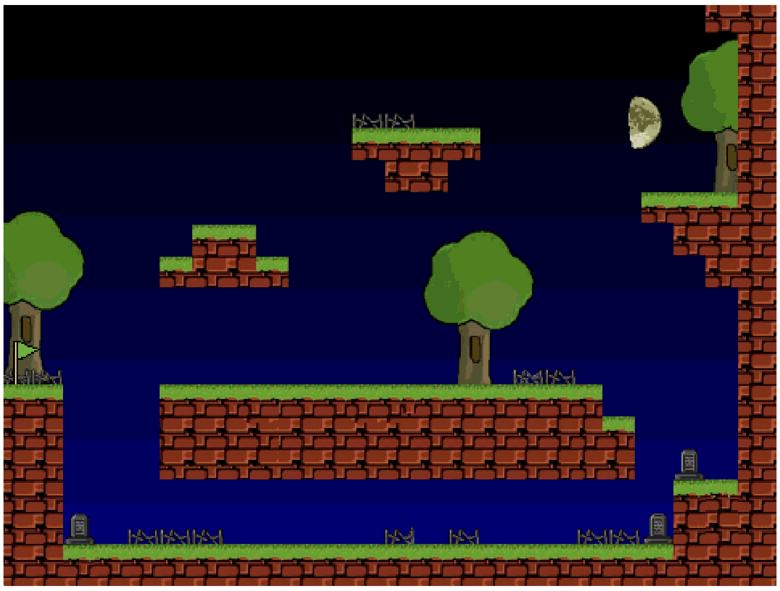


Figure 5: Background

4.3 Module: man_color_mapper

This module is designed for rendering a character in our game. It determines the correct color output for pixels within a defined area that represents the character on-screen. This area is specified by comparing DrawX and DrawY with ManX and ManY to calculate the pixel's index within the character's data. This module also reads the character's graphical data from a 32-bit input, man_data, and maps it to RGB values through a predefined color palette.

4.4 Module: man_left_palette

This module is designed to determine the appropriate color output for rendering a character facing left in our game. This module calculates which pixel of the character's image is being drawn based on its relative position within a 20x20 pixel block. This module ensures that the character's visual representation on the screen is accurate.

4.5 Module: man_dead_palette

This module is designed to render a character's appearance, specifically when dead, in our game. It assigns colors to a 20x20-pixel area representing the character based on its position relative to the drawing coordinates.

4.6 Module: man

This module is designed to handle a character's movement and rendering in a graphical user interface. It processes inputs such as directional controls from the keyboard and conditions like winning or dying to manipulate the character's position and state. The module calculates the character's position based on the current and next movement commands, adjusting for collisions and boundary limits. This module also manages jumping mechanics, ensuring that the character can only jump if it's on the ground and not already jumping.

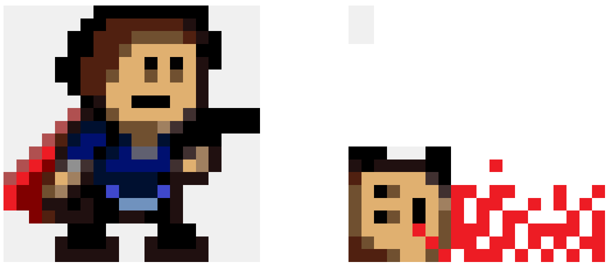


Figure 6 & 7: Character Picture

4.7 Module: spikes

This module is designed to control and display spike obstacles. It calculates the position of each spike relative to the screen and retrieves graphical data from memory based on whether the spike is facing up or down. The module uses two block memories to store the graphics for both orientations of the spikes, and it selects the appropriate color values to render each spike based on its orientation and position.

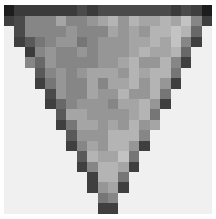


Figure 8: Spike Picture

4.8 Module: spike_color_mapper

This module is designed to generate the color output for spike obstacles based on their position and drawing coordinates. It takes the current drawing coordinates, spike coordinates, and spike data as inputs to determine which part of the spike is being drawn.

4.9 Module: spike_down_color_mapper

This module is designed to map color data for downward-facing spikes.

4.10 Module: apples

This module manages the display of multiple apples by assigning fixed coordinates to each apple and computing which part of an apple to render based on DrawX and DrawY. The module uses these coordinates to access pixel data from a block memory, which stores graphical data for the apples.

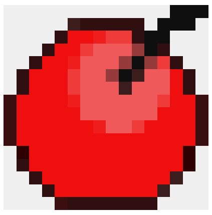


Figure 9: Apple Picture

4.11 Module: apple_color_mapper

This module calculates the specific pixel to display based on the current drawing coordinates relative to the apple's position. Using a predefined palette, it maps pixel data from a 32-bit value representing apple sprite data to 4-bit RGB color values.

4.12 Module: game_over_word

This module is used to display a "Game Over" message on a screen, utilizing graphical data stored in Block memory. It calculates memory addresses based on screen coordinates to retrieve graphical data, which is then converted into RGB color values using a color mapper.

Figure 10: Game Over Word

4.13 Module: over_color_mapper

This module is designed to map color data for a "Game Over" screen display. It determines the pixel index based on the position within the "Game Over" area and uses a predefined palette to assign RGB values.

4.14 Module: color_object

This module controls the color output for different game elements based on their coordinates. It prioritizes which element is visible at each pixel, managing overlays of the background, apples, spikes, the character, and messages like "Game Over" and "Win". The module checks each element's position and outputs the appropriate color, ensuring the correct visual representation on the screen.

4.15 Module: game_state

This module manages the state transitions within a game based on player interactions and game outcomes. It employs a state machine with four states: ALIVE, DEAD, RESET, and WIN. Transitions between these states occur based on triggers such as character collisions or completing a level. The module outputs signals corresponding to the current state, indicating whether the character is dead, the game should reset, or the player has won.

4.16 Module: game_state_check

This module is designed to detect collisions between the character and game elements such as spikes and apples. It checks if the player's position overlaps with these elements and sets a collide flag if a collision occurs. This flag can influence game states like ending the game or resetting the level. The module also checks for level completion and can trigger a reset based on specific key inputs.

4.17 Module: spike_location_updator

This module controls the positions and movements of spikes. Upon reset, it sets each spike's initial positions and states. During gameplay, it adjusts spike positions based on player interactions, triggering movements when the character reaches specific coordinates. This module ensures spikes stay within game boundaries, halting their movement at edges, and uses the game's frame clock to sync these updates.

4.18 Module: win_title

This module is designed to display a "win" title on the screen when a player wins a game. It determines the position of this title and fetches the appropriate color data from memory to be displayed.

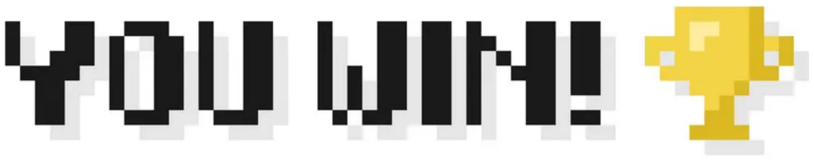


Figure 11: Win Title

4.19 Module: win_color_palette

This module is designed to assign colors to pixels for a win-title graphic on a display screen. It determines which part of the graphic each pixel corresponds to by computing an index based on the pixel's relative position within the graphic area.

4.20 Module: pwm_top

This module manages audio playback for different sound effects based on game events like 'jump', 'dead', and 'win'. It prioritizes 'dead' and 'win' sounds over 'jump' to reflect their importance in the game's state. The module controls a PWM generator to output audio, handles sound selection, and ensures uninterrupted playback of critical sounds. It uses a playback complete signal and modifies soundtracks dynamically.

4.21 Module: audio_rom

This module selects and outputs audio data from different block memories based on input signals. It supports three sound effects: 'jump', 'dead', and 'win', each stored in separate memory blocks.

4.22 Module: pwm_core

This module generates a PWM signal. It uses a counter that increments every clock cycle until it equals a predefined period, then resets to zero. The PWM output stays high as long as the counter value is less than a set pulse width and goes low otherwise.

5. Software Design

The C workspace setup is exactly the same as LAB 6.2 as we only requires keycode passed from the keyboard.

5.1 MAXreg_wr(BYTE reg, BYTE val)

This function is designed to write single-byte values for specific registers in the MAX3421E, configuring and controlling its operations for USB tasks. This includes setting up for USB communication, managing device interactions, and ensuring the USB protocol's proper functionality. It modifies the register value to indicate a write operation and utilizes a two-byte buffer for the transfer, handling the selection and deselection of the MAX3421E.

5.2 MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)

This function enables the transmission of multiple bytes to a specific register, catering to situations where data packets exceed a single byte in size. Similar to MAXreg_wr, it signals a write operation but differs by extending the buffer to accommodate the additional data bytes and adjusting the transfer length accordingly.

5.3 MAXreg_rd(BYTE reg)

This function fetches a byte from a target register, which is useful for acquiring new data, such as input keystrokes. It maintains the convention of signaling operation type through the register value and utilizes a two-byte buffer where the second byte is dedicated to receiving the read data.

5.4 MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)

This function facilitates reading multiple bytes from a register, suited for obtaining larger or more detailed information chunks from the device. It expands the buffer to include the additional bytes and mirrors the structure of MAXreg_rd while catering to multi-byte reads.

6. Simulation Waveforms

Notice that the game logic part includes interaction with keycode from keyboard and game interaction at different game object location, which makes designing a specific pattern of keycode in testbench inefficient and not comprehensive for testing, we directly test out game logic without a specific testbench. The Audio part includes a testbench and simulation waveforms.

Key Components in Waveform:

1. clk – Clock Signal
2. reset_rtl_0 – Reset Signal
3. en – Enable Signal
4. jump, dead, win – Control inputs
5. leftsound, rightsound – PWM Output

All configurations are set to default. All operational signals should stabilize by the end of this phase. The reaction time from when a sound trigger is activated (jump, dead, win) to when the PWM output changes should be minimal, demonstrating the system's responsiveness.

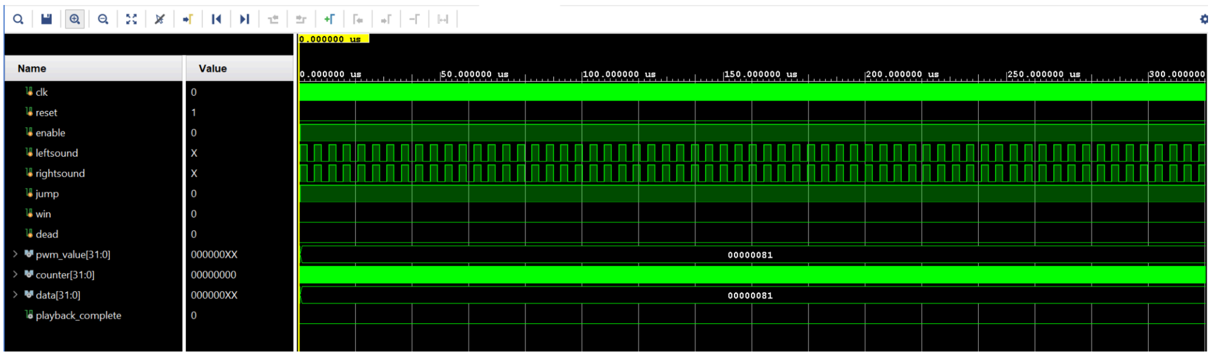


Figure 12: Simulation waveform for jump sound

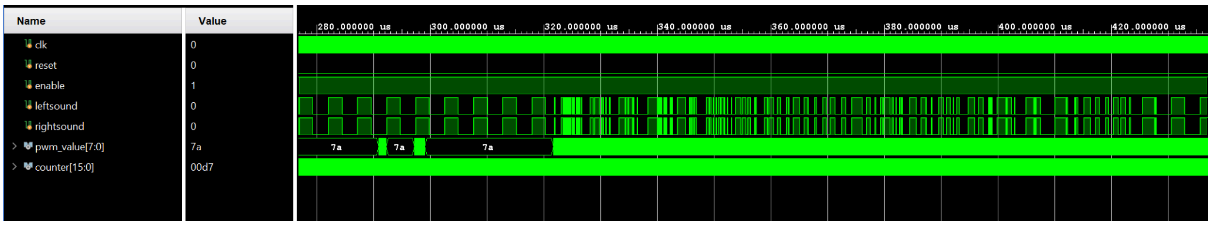
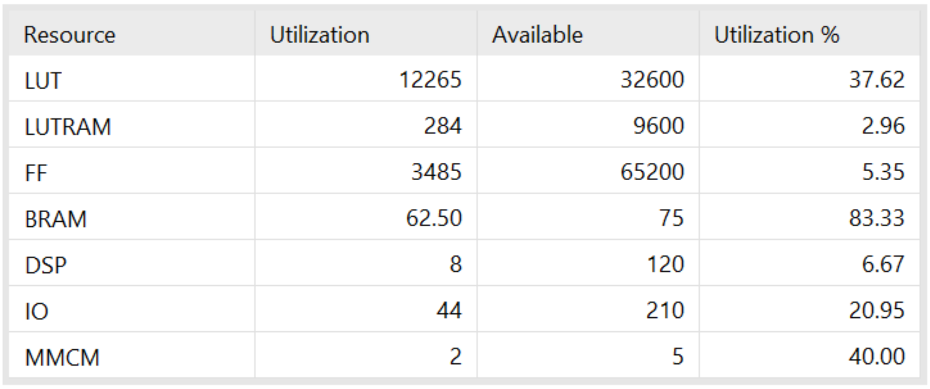


Figure 13: Simulation waveform for dead sound

7. Design Resources and Statistics

LUT	12256
DSP	8
Memory (BRAM)	83.3%
Flip-Flop	3485
Latches*	0
Frequency	20.67MHz
Static Power	0.079
Dynamic Power	0.507
Total Power	0.585



Resource	Utilization	Available	Utilization %
LUT	12265	32600	37.62
LUTRAM	284	9600	2.96
FF	3485	65200	5.35
BRAM	62.50	75	83.33
DSP	8	120	6.67
IO	44	210	20.95
MMCM	2	5	40.00

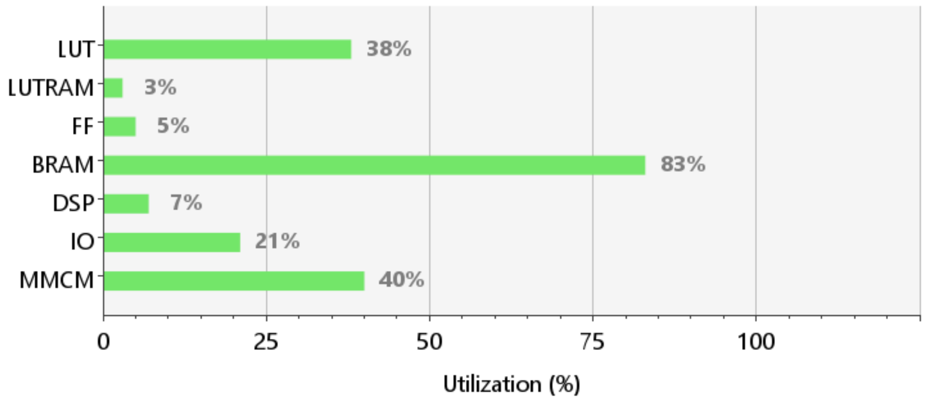


Figure 14: Project Resource Usage Report

8. Conclusion

We successfully developed an "I wanna" series game on the Spartan-7 Urbana FPGA board, incorporating advanced game logic, sound effects, and VGA visuals. the development of this FPGA-based game not only solidifies the understanding of digital design principles but also enhances practical skills.