ECE 385

Spring 2025

Final Report

Heet Chauhan

1. Introduction

a. Project Overview

The goal of our final project was to design and implement a playable version of the classic arcade game *Tetris* on an **FPGA** and rendering the graphical output in real time through **VGA display** on an external hardware to emulate the arcade experience to a degree. We handled all the game logic such as collisions, tetromino generation, and user signals from the keyboard into a **single monolithic hardware module** to maintain simplicity and efficiency which also handled the background generation.

Our project required careful coordination of game state tracking with pixel-level accuracy due to the collision logic and pixel-based drawing for the boundaries and the score and ensure that our design met within the real-time constraints of VGA synchronization and FPGA hardware. While the original proposal included support for a multiplayer version with 2-players playing simultaneously to provide a more interactive multiplayer experience synergistically with real-time audio playback. However, due to timing and memory constraints, our final implementation focuses on the core single-player Tetris experience with keyboard control and on-screen visual output.

b. Design Motivation

We choose to implement tetris for our final project due to its use of strong game logic and programming control as well as visual graphical representation. We had to learn to address key game logic such as shape rendering, user-controlled movement, line-clearling logic, rotation checks and collisions which helped us really hone our logic thinking and SystemVerilog implementation skills. We were especially drawn towards this project because it was the perfect balance of technical difficulty with the right level of appealing visual elements. It essentially presented a manageable yet non-trivial project that required a deep understanding of timing (always_ff vs always_comb) blocks for example or pixel mapping with draw_X and draw_Y from the VGA with the moving tetris blocks. These concepts overlap well and really enhanced our understanding of the core learning objectives of ECE385.

c. Differences from Previous Labs

One of the core differences from our previous labs was our overall implementation architecture. We intentionally developed our game using a self-contained single module which allowed us to have a stricter control over our game logic. This architecture allowed us to ensure that we used the hardware and resource on the FPGA as efficiently as possible since we were minimizing the additional overhead between inter-module connections which would require its own complex hardware in the top-level while making the process of combining the logic with the Tetris block coordinates along with the existing stationary block to provide a smoother, more-real time and interactive experience.

2. Written Description of the System

Our architecture is centered around a 21x40 cell grid to handle all the movement, collisions, rotatins and line-clearning logic which is enhanced with real-time player control via a USB keyboard and pixel-accurate VGA output required for the extremely precise movement required by tetris. We incorporated the MAX3421E USB controller and the supporting SPI logic and polling code from Vitis that we had previous implemented. The keycodes generated by the user from the USB peripheral was directly used on our module to control critical game logic signals including rotations, lateral and horizontal movements as well as start and end screen navigation. Our hybrid approach of allows us to efficiently use the control paths and modularize the overall architecture of the design to allow for a powerful game engine.

Our design reflects our intensive attention to detail over the timing synchronization across all aspects of the logic control with efficient memory rotation and dynamic rendering behavior.

Game Operation and Control Pipeline

The game operates based on a combination of user inputs from the keyboard which are decoded as well as additional logic elements (falling, collisions with old tetris blocks). However, our implementation can be abstracted into the following gameplay loop:

1) player-controlled movement

2) Automatic downward falling

3) Collision detection

4) Piece locking

5) Line Clearing

6) Score detectiokn

7) New tetromino generation using LFSR algorithm.

Keyboard input is received through a decoded keycode input where each keycode corresponds to a unique key. Our logic only uses the following keycodes:

- A (0x04) moves the active tetromino left.
- D (0x07) moves it right.
- S (0x16) increases the fall speed (soft drop).
- W (0x1A) triggers a clockwise rotation.

All input is gated using a move_delay counter to prevent rapid, unintended multiple movements due to sustained keypresses. Movement inputs are only accepted once move_delay reaches zero.

All of our inputs has an associated move_delay counter to prevent extremely quick and unintended movements due to sustained keypressed. Movement inputs are only accepted after the delay (move_delay variable) reaches zero which indicated to the system that the user is allowed to make another input.

For our tetromino control, we primarily use two coordinates:

- SHAPE_ROW_Y: Top Row of Tetromino
- SHAPE_COL_X: Leftmost Column of Tetromino

Additionally, we use these variables to define the placement of a 4×4 shape matrix over the global game_grid which contains all of the positions of previous tetromino blocks and overall state of the game. Similar to move_delay for user control of the tetromino, We add a delay to the falling motion using a drop_counter which is incremented each vsync. When drop_counter exceeds a threshold (20), the tetromino moves down pending approval from the collision logic. We use the same concept for move_delay where we increment it every clock cycle and then only allow movements after a certain threshold is reached before resetting it for the next movement.

Collision Detection and Locking

We have three dedicated always_comb blocks for collision detection. We used always_comb block due to their asynchronous nature since they do not rely on clock signals to update for instant update upon every change for accurate game logic signals. Our three blocks are as follows:

- collision_left and collision_right blocks which drive a dedicated collision_left and collusion_right indicator variable if the block is sharing a border with either the game boundaries of another tetromino block using both the pixel coordinates as well as game_grid for collisions with other blocks respectively.
- collision_down: This block drives the dedicated collision_down variable if either an existing block is below the tetromino or if the block is at the bottom of the game.

To accurately compute lateral bounds of the shape, the code calculates min_j and max_j, the leftmost and rightmost occupied columns of the current rotation. This allows for precise wall collision handling, especially for wide shapes like the I-block.

Additionally, to ensure the highest accuracy and to handle edge cases due to the non-rectangular and non-uniform tetrominos, we calculate a min_j and max_j variable as well which calculated the leftmost and rightmost occupied columns for the respective rotation to allow for precise collision handing and rotations.

Once a collision is detected, the tetromino is essentially ‘locked’ in the grid which is essentially acts as a matrix which stores which cells are occupied by a block and not. In our logic, each individual tetromino is a 4x4 matrix but only the cells that are considered part of the tetromino are 1 and these cells are written into the grid. Then a buffer delay is added after the locking and the generation of the next tetramine to mimic the soft-lock delay behavior of classic Tetris.

Line Clear and Spawning Logic

After locking, the design performs a full pass through each row in game_grid to check for line completions. If a row is entirely filled (full == 1), that row is cleared by setting all its columns to 0. Rows are not shifted down in this implementation—line clear simply blanks the filled row.

Once clearing is complete and the buffer expires, a new piece is spawned. The following initializations occur:

- locked is reset to 0.
- SHAPE_ROW_Y is reset to 0 (top of the screen).
- SHAPE_COL_X is set to 10 (rough center of the grid).
- drop_counter is reset.
- tetromino index is incremented modulo 2 (cycling through the two implemented shapes: I-block and O-block).

Edge-case logic for shape placement is also implemented. For example, when moving an I-block or O-block near the screen edge, special conditional statements correct rotation states to prevent illegal overlapping behavior.

Memory Organization and Shape Representation

The game board is stored as a two-dimensional logic array and can also be interpreted logically as a matrix where the cells represent the occupancy. We define it as follows:

```
logic [GRID_COLS - 1:0][GRID_ROWS - 1:0] game_grid;
```

This stores the memory of the locked blocks and is updated every time when a tetromino locks or a line is cleared. It is used in the tetromino rendering as well and is extremely important when we add the incrementing logic where all the rows above the line just cleared fall down. Each cell stores a single bit indicating whether it is filled and at the start of the program all the cells are initialized to 0 to indicate an empty grid.

We store the tetrominoes in a similar fashion but in a 4D array instead. We define it as follows:

```
logic [3:0][3:0][3:0] tetrominoes[6:0];
```

Each tetromino has up to 4 rotation states and we store each rotation in the 4×4 matrix representation. Here are two example rotations for the ‘I’ tetromino:

```
tetrominoes[0][0] = '{
  4'b1111,
  4'b0000,
```

```
  4'b0000,
  4'b0000
};

tetrominoes[0][1] = '{
  4'b1000,
  4'b1000,
  4'b1000,
  4'b1000
};
```

As you can see, the ‘1’ bits indicate the cells that the block would occupy in game_grid when falling. We define each cell to be 12x12 pixels. We use this representation in our rendering logic in loops where we check tetrominoes[tetromino][rotation][i][j] to access individual bits from the tetromino bitmaps in the rendering logic using the variables I and j.

Real-Time Rendering Logic

We synchronize the display output with the VGA controller using the dedicated vga_clk signal. The DrawX and DrawY signal provided by the vga_controller module determine the current pixel being drawn and they are mapped in our logic for the current grid coordinates using the following equations:

```
draw_col = DrawX / CELL_SIZE;
draw_row = DrawY / CELL_SIZE;
```

Each cell corresponds to a 12×12 pixel block on screen. We identify the pixel currently being drawn based on the grid that is currently being drawn using the following equations using the fact that the side length of each cell is 12:

```
pixel_in_cell_x = DrawX % CELL_SIZE;
pixel_in_cell_y = DrawY % CELL_SIZE;
```

Rendering is based on the value of occupied_cell which we used the tetrmoninoes bitmaps bits with the following logic:

- The pixel lies within the active tetromino (shape_cell == 1)
- OR it lies in a locked cell (game_grid[draw_col][draw_row] == 1)

In the always_ff @(posedge vga_clk) block, if occupied_cell is true and the screen is blanking, a static RGB color (Red: 0xD, Green: 0xA, Blue: 0xF) is written to the VGA output as a default but, the pixel defaults to our background color which we designed for our game.

Full Game Cycle Walkthrough

The game operates based on a combination of user inputs from the keyboard which are decoded as well as additional logic elements (falling, collisions with old Tetris blocks). However, our implementation can be abstracted into the following gameplay loop:

1. **Shape Falling:**

   On every vsync clock cycle we increment the drop_counter variable and when it exceeds a threshold (20) that we defined to control the pace of the game and the piece is not locked, the system tries to increment SHAPE_ROW_Y which controls the row of the tetromino, which represents the vertical position of the tetromino as long as there are no collisions.

2. **Collision Check:**

   Before falling or moving the system checks for collisions using three dedicated always_comb blocks. We have a downward collision (collision_down) variable that is flagged if any occupied cell in the 4×4 tetromino matrix would either:

   - Overlap an active cell in game_grid one row below, or
   - Exceed the grid boundary (r = SHAPE_ROW_Y + i + 1 >= GRID_ROWS) where GRID_ROWS are the number of rows in the entire game (40).

3. **Locking:**

   If the tetromino is unable to move down the piece is locked and each active 1 bit in the tetromino's 4×4 matrix is written into game_grid[SHAPE_COL_X + j][SHAPE_ROW_Y + i] which allows the tetromino to persist in the game even after another tetromino has been generated.

4. **Line Clear:**

   After locking, the system iterates through each row (y) of game_grid to check whether all columns are filled and if a full row is detected (full == 1) using a simple nested for loop. It is cleared by setting all bits in that row to 0 and then shifting all rows above it downward using another for-loop.

   New Piece Generation:

   After a short buffer delay (buffer) after a tetromino is locked we initialize a new tetromino with the associated parameters:

   - SHAPE_ROW_Y = 0
   - SHAPE_COL_X = 10
   - locked = 0
   - drop_counter = 0
   - Then, we use a pseudo random LFSR algorithm to "randomly" choose a tetromino index and spawn it at the bounds specified above. This algorithm ensures that the pieces generated do not repeat in sequence very often. To further explain the

algorithm, a Linear Feedback Shift Register (LFSR) is a shift register whose input bit is a linear function (typically XOR) of its previous state bits and it generates pseudo-random binary sequences by shifting bits through the register and feeding back a combination of bits based on a predefined polynomial.

5. **Score Update:**

   Our design of Tetris also includes a score counter which updates differently depending on the parameters of the game. To be more specific, if the user is choosing to play on the lowest speed (difficulty), then the score starts increasing by 10 after the first line is cleared and then by 10 based on each block that is places/locked. However, if the speed is greater and the round goes on longer the lines cleared award 100 points and each block placed also corresponds to a higher score than 10. Additionally, this score counter is also created in a similar manner to the tetrominoes, with a rom of digits 1-9. In addition, the same rom is also used to generate and display the current chosen speed of the tetrominoes.

Collision Detection Equations

Collision logic is computed using combinational logic within always_comb blocks due to its asynchronous nature allowing for immediate and updated logic signals which are essential for the games operation. The shape's current 4×4 matrix is iterated using nested loops with variables (i, j) to compute grid-relative coordinates r = SHAPE_ROW_Y + i and c = SHAPE_COL_X + j to get the respective coordinates in the game_grid to find out activate cells. The three key types of collision are:

- **Downward Collision:**

```
r = SHAPE_ROW_Y + i + 1;
if (game_grid[c][r]) collision_down = 1;
```

- **Lateral Collision (Left/Right):**

  First determins the shape's horizontal bounds using the true width and height for each individual tetromino:

```
if (SHAPE_COL_X + min_j <= 0) collision_left = 1;
if (SHAPE_COL_X + max_j >= GRID_COLS - 1) collision_right = 1;
```

- It also checks for adjacent locked blocks:

```
if (game_grid[c-1][r]) collision_left = 1;
if (game_grid[c+1][r]) collision_right = 1;
```

- **Rotation Validity:**

  To prevent illegal rotations, the design simulates a future rotated matrix and verifies using the same collision detection logic for above for the next rotation to always precompute the

signal using the variable new_r which just holds the incremented rotation index that is then put through a module operator to only cyucle through the possible rotations:

```
if (new_r >= GRID_ROWS || new_c >= GRID_COLS || game_grid[new_c][new_r])
    valid_rotation = 0;
```

3. Hardware/Software Interface (What we used from Lab 6 vs. What we created)

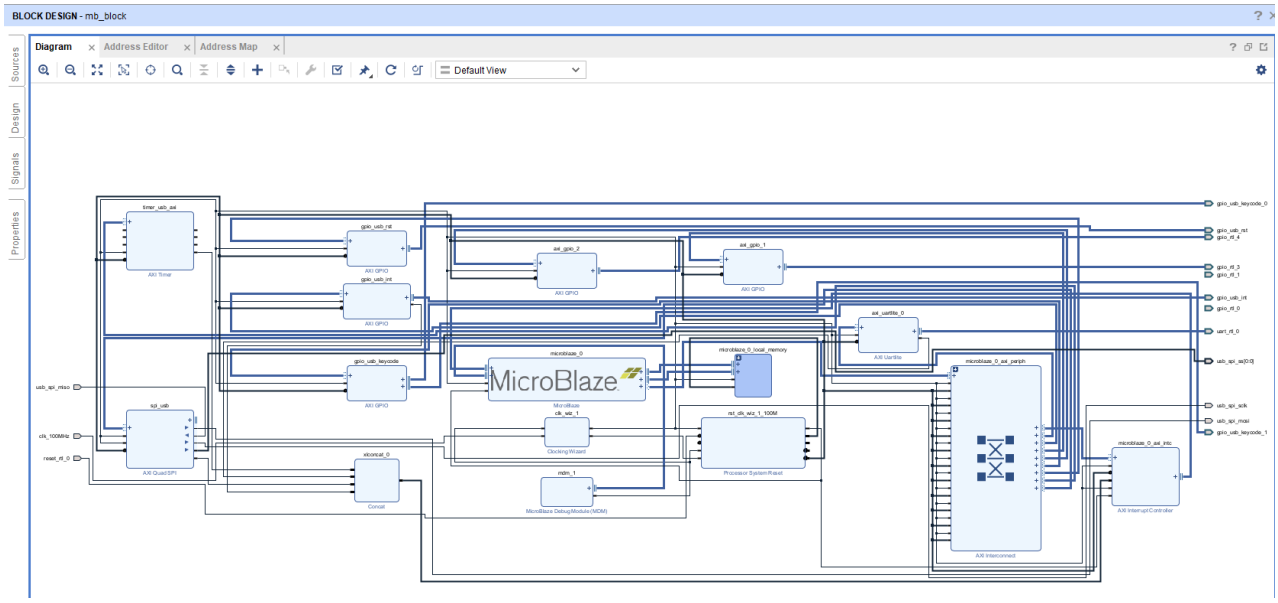
All the game logic, dynamic rendered as well as grid memory were implemented entirely using hardware using System Verilog. However, to enable real-time user control we incorporated the design elements from the Lab 6.2 setup. We included the MAX3421E USB interface as well as the necessary SPI peripheral connections to enable keyboard input and decoding for the tetromino movements and control.

Although our entire game logic was hardware based, we used the USB polling loop on Vitis to decode the keycodes from the USB keyboard where were then passed into the tetris module, similar to the Lab 6.2 setup. As we mentioned in our Final Project Proposal, we utilized the base design and setup of Lab 6 to build upon and create our game. However, we did not use the color mapper files or the ball logic for any of our sprites or game logic. However, we did utilize the VGA-HDMI IP and modules along with the VGA controller that was provided in Lab 6.2 for our vsync and displaying our RBG values onto the screen. However, how we generate and assign values to the RGB was part of our game logic build and was entirely unique.

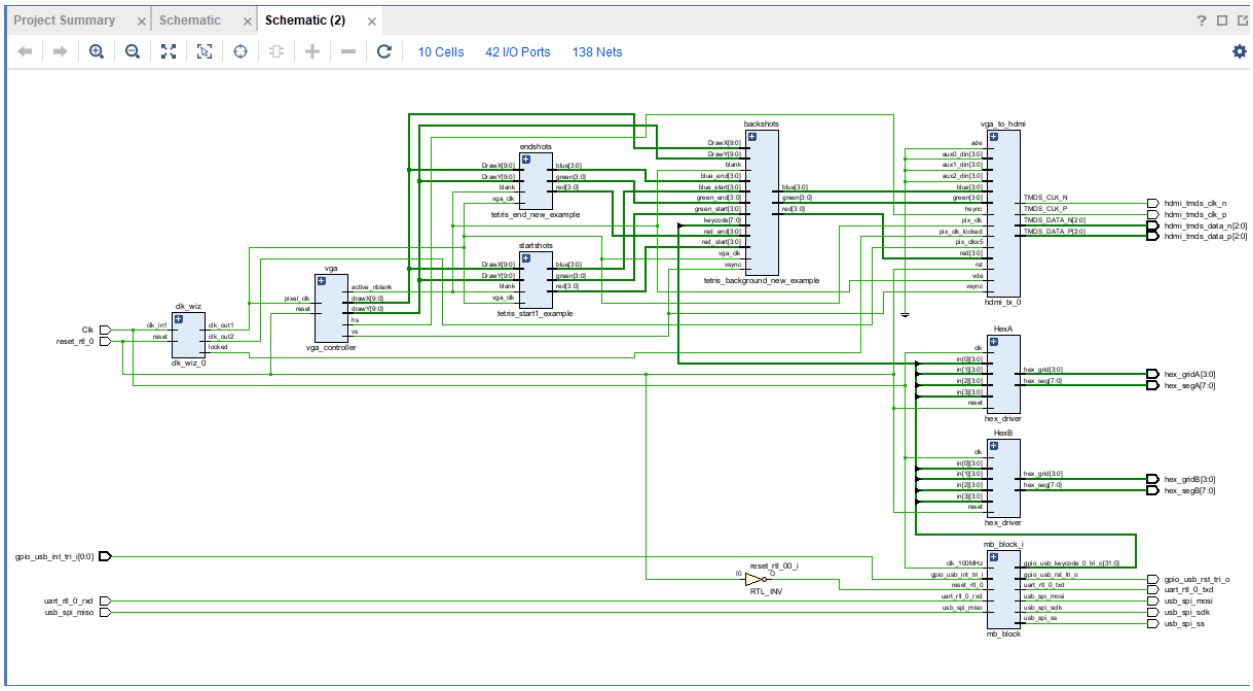
Additionally, we also utilized sprites in our design of the game, for example, the background sprite. The background sprite also provided us with the example file, which we treated as the main file for our entire game.

4. Block Diagrams and State Machines

a. Top-Level Block Diagram:



b. Top-Level RTL Diagram:



Create a diagram showing:

- Input ports (VGA clock, keycode, draw coordinates, etc.)
- Output ports (RGB channels)
- Internal grid array and control logic.

- **c. State Machines:**

Our Design did not consist of any major state machines; however, we included 3 states in our game. These states were the logic for the Start screen, Game Screen, and the End screen. This was a simple design as we created 2 different 1-bit signals, one for the start and one for the end. The start signals defaulted to 1 and changed to 0 when the user pressed enter key on the keyboard. After which we transition to the next state. Which is to display the game screen and run the game. When the game ends (the blocks reach the top of the screen), the end signal goes high and then the end screen is printed, and the game is stopped.

5. Module Descriptions:

Name: mb_usb_hdmi_top

Inputs: Clk (1-bit), reset_rtl_0 (1-bit), gpio_usb_int_tri_i (1-bit), usb_spi_miso (1-bit), uart_rtl_0_rxd (1-bit), hdmi_tmds_clk_n (1-bit), hdmi_tmds_clk_p (1-bit), hdmi_tmds_data_n (3-bits), hdmi_tmds_data_p (3-bits)

Outputs: gpio_usb_rst_tri_o (1-bit), usb_spi_mosi (1-bit), usb_spi_sclk (1-bit), usb_spi_ss (1-bit), uart_rtl_0_txd (1-bit), hex_segA (8-bits), hex_gridA (4-bits), hex_segB (8-bits), hex_gridB (4-bits)

Description: This module integrates USB, UART, HDMI, and VGA functionalities and it communicates through SPI for USB devices, handles UART data transmission, and drives HDMI output signals from VGA signals. It also generates the hex displays that can be used to monitor the keycodes as they are being pressed for ball movement. This is the top-level module that integrates all the other modules and parts of the project.

Purpose: The purpose of this module is to manage communication between USB devices, UART peripherals, and HDMI output for a system that integrates graphics onto an HDMI display.

Name: hex_driver

Inputs: clk (1-bit), reset (1-bit), in (4 x 4-bits)

Outputs: hex_seg (8-bits), hex_grid (4-bits)

Description: This drives the 8-bit hex_seg output and the 4-bit hex_grid output to control a 4-digit 7-segment display. We use this to monitor which keycodes are beign pressed to make the ball in the direction we want.

Purpose: The purpose of the hex_driver module is to convert 4 nibbles of input data into their 7-segment display equivalents and manage the display of these values on a 4-digit hex display.

Name: vga_controller

Inputs: pixel_clk (1-bit), reset (1-bit)

Outputs: hs (1-bit), vs (1-bit), active_nblank (1-bit), sync (1-bit), drawX (10-bits), drawY (10-bits)

Description: The module takes a pixel clock and a reset signal as inputs while it outputs the horizontal sync pulse (hs), vertical sync pulse (vs), and the active video signal. Additionally, It also outputs the drawX and drawY coordinates

Purpose: The purpose of the vga_controller module is to generate the necessary sync signals and coordinates for a VGA display with a set resolution, using a pixel clock and makes sure that proper timing is provided for the VGA display and that the correct pixels are getting displayed.

Name: tetris_background_new_example

Inputs: input logic vga_clk, input logic [9:0] DrawX, DrawY, input logic blank, input logic [7:0] keycode, input logic vsync, input logic [3:0] red_start, green_start, blue_start, input logic [3:0] red_end, green_end, blue_end,

Outputs: output logic [3:0] red, green, blue

Description: This is the file created by the python script we ran to get our background sprite (image to COE). Additionally, this file serves as the only file in which we wrote our entire code for the final project. This file contains everything including collision logic, red, green, blue signal generation, keycode handling, mini state machine for the star and end screens, game, logic, tetromino creation, text creation, and everything else that our game included.

Purpose: The module implements a full-featured, hardware-based version of the classic game Tetris, intended for display on a VGA output. It draws a grid-based playfield where tetrominoes (the falling shapes) move, rotate, and lock into place, while also detecting collisions, clearing full lines, updating the score, and increasing speed based on performance. Additionally, it supports dynamic rendering of digits for the score and speed display, uses ROM-based graphics for background visuals, and maintains game states like start screen and end screen with custom colors. It interfaces with user input via keycode and updates game behavior synchronously with vsync and vga_clk

Name: tetris_background_new_palette

Inputs: input logic vga_clk,

input logic [9:0] DrawX, DrawY,

input logic blank

Output: output logic [3:0] red, green, blue

Description: contains the color pallete of the game screen sprite so that it able to be drawn correctly. It uses the rom-address and inde to go through the colors on the pallete and draw the background

Purpose: The purpose is to draw the background sprite for the actual game.

Name: tetris_start1_example:

Inputs: input logic vga_clk,

input logic [9:0] DrawX, DrawY,

input logic blank,

Output: output logic [3:0] red, green, blue

Description: This is similar to the other background file that houed ur entire tetris game implementation, however, this file only creates the instance of the start screen sprite so that we can display it using the other background file.

Purpose: The purpose of this file is to create a start screen sprite and provide the correct red,green,blue values to the backgound file/game file and then display it when the start signal is one.

Name: tetris_start1_pallete

Inputs: input logic [2:0] index,

Outputs: output logic [3:0] red, green, blue

Description: the file only contains the pallete that is use to draw the start screen sprite when needed.

Purpose: This file's purpose is to use the rom address and index of the start screen to update the red,green, blue values so that the sctart screen sprite is drawn onto the screen.

Name: tetris_end_new_example

Inputs: input logic vga_clk,

input logic [9:0] DrawX, DrawY,

input logic blank

Outputs: output logic [3:0] red, green, blue

Description: This is similar to the other background file that houed ur entire tetris game implementation, however, this file only creates the instance of the start screen sprite so that we can display it using the other background file.

Purpose: The purpose of this file is to create a start screen sprite and provide the correct red,green,blue values to the backgound file/game file and then display it when the start signal is one.

Name: tetris_end_new_pallete

Inputs: input logic [2:0] index,

Outputs: output logic [3:0] red, green, blue

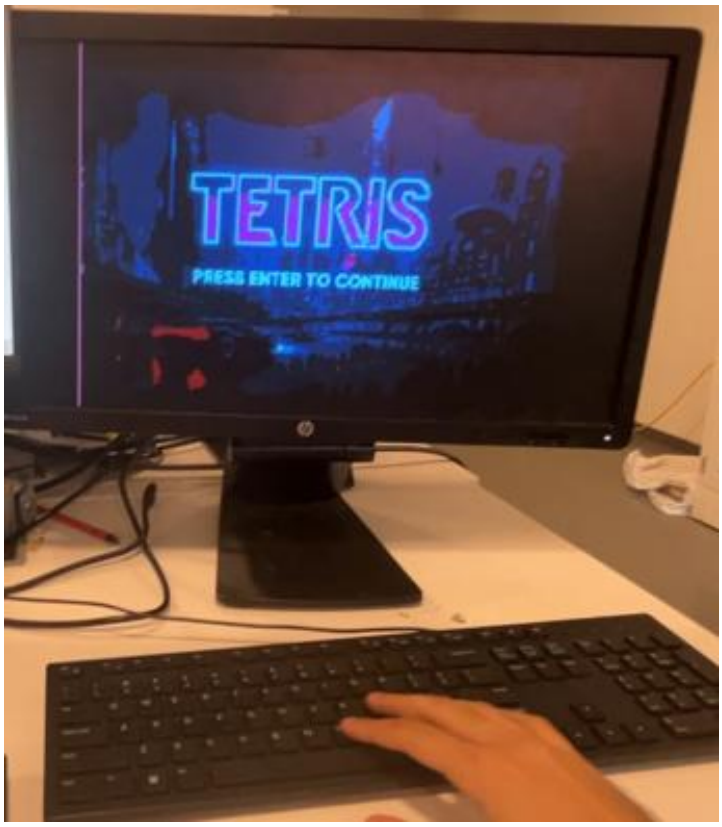
Description: the file only contains the pallete that is use to draw the end screen sprite when needed.

Purpose: This file's purpose is to use the rom address and index of the end screen to update the red,green, blue values so that the end screen sprite is drawn onto the screen.

6. Simulation Results: we did not have to utilize a test bench or simulate any results for completing this project because all the elements in this project were visual elements which used real-time keyboard inputs for testing.

7. Images and Gameplay

Start Screen:



Screenshot from real-time gameplay:

End Screen:

8. Resource Usage and Statistics

LUT	8967
DSP	9
Memory (BRAM)	12
Flip-Flop	5082
Latches*	0
Frequency	118.58 MHz
Static Power	.075 W
Dynamic Power	.386 W
Total Power	.461 W

- **b. Discussion:**

Our Design is not the most efficient as it could have been as we realized that we could have used a lot more of the BRAM and saved on a lot of FPGA resources like the FFs and LUTs. However, our implementation still works exactly as intended without any performance issues. Something that we would do differently next time is to definitely store our game's grid entirely in BRAM an then modify it, which would also reduce the number of resources we need to use and make generating bitstreams a lot faster which helps a lot for debugging and game testing.

9. Conclusion

a. Summary of Design Functionality:

Our final design successfully achieved a playable single-player version of Tetris on an FPGA platform using VGA output and keyboard input through USB and while we scaled back some of our original project goals, such as multiplayer support and real-time audio, we maintained core gameplay functionality including real-time rendering, randomized tetromino generation, accurate collision detection, scorekeeping, and screen transitions for game states (start, active, and end). Furthermore, all of our game logic and rendering were done entirely in hardware,

b. Lessons Learned:

One of the major lessons that we learned with this project is memory allocation with FPGAs. While doing the project, we realize that our synthesis took longer and longer due to the fact that it was so inefficient. This is why next time we will be sure to make the design more efficient and use both BRAM and the on-board resources like LUTs and FFs in balance.

c. Potential Extensions that we could have added:

Add soundtrack and sound effects

Multiplayer support