ECE 385

Spring 2024

Final Project

FPGA-based Tetris Game

Name: Jie Wang, Shitian Yang

Student ID: 3200112404, 3200112415

Prof. Chushan Li,
Prof. Zuofu Cheng
ZJU-UIUC Institute
May 24th , 2024, Friday D-225
TA: Jiebang Xia
Demo Point: 25/30

1. Introduction

1.1 Project Overview

The objective of this project is to develop a real-time Tetris game on an FPGA using SystemVerilog and C. Based on the Avalon Bus for IP communication, the game features a VGA display for graphical output and a PS/2 keyboard for user input. The project includes the basic game logic, rendering graphics, handling user inputs, and managing game states. With our knowledge learned through *ECE385: Digital Systems Lab,* we provide a new illustration for this classical game.

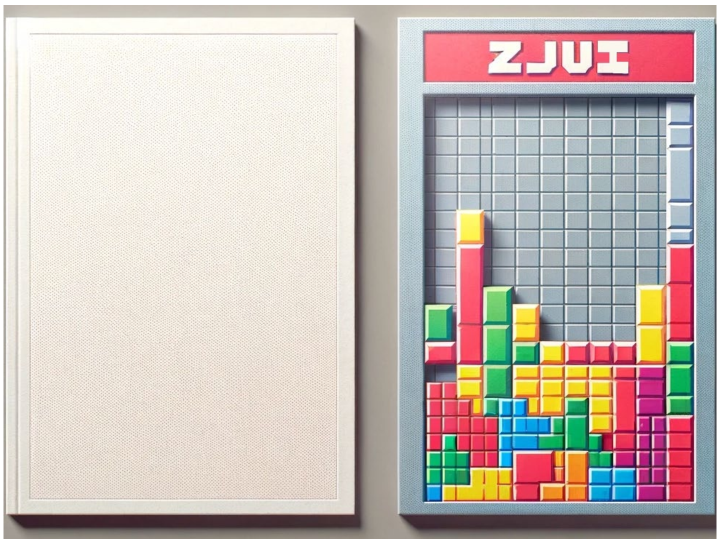


Fig 1: Original Taris Game Background, drawn by DALL-E. This image is too big for the FPGA to render, thus we need to compress and redraw it in 160 * 120 pixels size.

1.2 Motivation and Goals

This project was chosen because it provides a clear basis for the difficulty and development. With basis on 'Lab 8: SOC with USB and VGA Interface in S', we keep improving our hardware and software design skills. The goals include gaining practical experience in digital system design, understanding the integration of hardware and software components in embedded systems.

We also participated in ***"Eaton Cup"*** *ECE 385 FPGA Platform Digital Design Competition,* winning 'excellent design award' and received beautiful gifts from Eaton Co.

Fig 2(left): EATON, our course sponsor.
Fig 3(right): Demo to the Professors on our game.

2. System Design

2.1 Block Diagram

Figure 4 is the block diagram of our system, with Keyboard Handler, Game Logic Controller, Audio Handler, VGA Display Handler, Memory Module, Timer Module and Score and Level System.

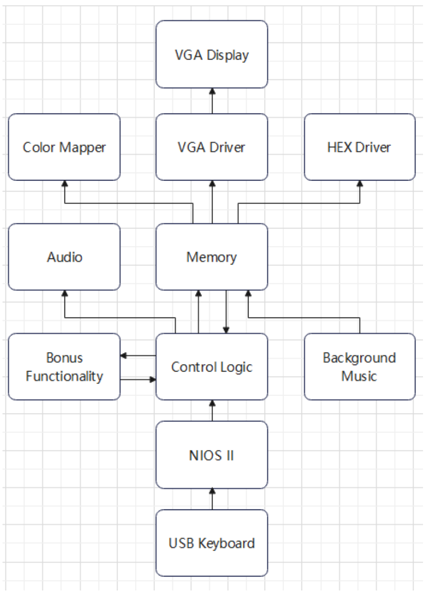


Fig 4: System Architecture with corresponding SV module and C code.

2.2 Description of Modules

- **Keyboard Handler**: Captures user inputs from the PS/2 keyboard to control the game. Including
- **Game Logic Controller**: Manages the game state, including piece generation, movement, rotation, and collision detection.
- **Audio Handler**: Manages background music and sound effects.
- **VGA Display Handler**: Renders the game graphics on the VGA display.
- **Memory Module**: Stores the game state, including the current position of pieces and cleared lines.
- **Timer Module**: Controls the game speed and progression by generating timing signals.
- **Score and Level System**: Tracks the player's score and adjusts the game difficulty level.

2.3 Hardware Setting: IP Cores and Avalon Bus

Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Family	Cyclone IV E
Device	EP4CE115F29C7

The hardware component is the FPGA, which runs the SystemVerilog modules for game logic and display rendering. The software component is a C program running on an embedded processor within the FPGA, handling game states, score tracking, and user input processing.

IP Cores Used:

1. **Nios II Processor (nios2_gen2_0)**:
   - **Address**: 0x0000_1000-0x0000_17FF
   - **Function**: Controls the overall game logic and coordinates between hardware and software.
2. **On-Chip Memory (onchip_memory2_0)**:
   - **Address**: 0x0000_0000 - 0x0000_000F
   - **Description**: Stores the state of the game grid and active pieces.
3. **SDRAM Controller (sdram)**:
   - **Address**: 0x1000_0000 - 0x17FF_FFFF
   - **Description**: Provides additional memory for game data storage, ensuring smooth gameplay.
4. **SDRAM PLL (sdram_pll)**:
   - **Address**: 0x0000_0090 - 0x0000_009F
   - **Description**: Generates the clock signals required by the SDRAM controller.
5. **System ID Peripheral (sysid_qsys_0)**:
   - **Address**: 0x0000_00A8 - 0x0000_00AF
   - **Description**: Provides a unique identifier for the system.
6. **JTAG UART (jtag_uart_0)**:
   - **Address**: 0x0000_00B0 - 0x0000_00B7
   - **Description**: Facilitates communication between the FPGA and a host computer for debugging purposes.
7. **PS/2 Keyboard Input (keycode)**:
   - **Address**: 0x0000_0080 - 0x0000_008F
   - **Description**: Handles inputs from the keyboard to control game pieces.
8. **Various OTG HPI Controllers**:
   - **Avalon Memory Mapped Slaves**
     - **Address**: otg_hpi_address_s1: 0x0000_0070 - 0x0000_007F
     - **Address**: otg_hpi_data_s1: 0x0000_0060 - 0x0000_006F
     - **Address**: otg_hpi_r_s1: 0x0000_0050 - 0x0000_005F
     - **Address**: otg_hpi_w_s1: 0x0000_0040 - 0x0000_004F
     - **Address**: otg_hpi_cs_s1: 0x0000_0030 - 0x0000_003F
     - **Address**: otg_hpi_reset_s1: 0x0000_0020 - 0x0000_002F
   - **External Connections**: Conduits for each OTG HPI module
   - **Description**: Interfaces for handling OTG communications and control signals.

Function Description of Each IP

- **VGA Controller**: Drives the VGA monitor, updating the display based on the current game state. It interprets the game grid data and renders the appropriate colors and shapes.
- **PS/2 Keyboard Interface**: Handles user inputs, translating key presses into game actions like moving or rotating tetrominoes.
- **Memory Controller**: Ensures efficient storage and retrieval of game state information, supporting real-time updates and smooth gameplay.

3. Features and Functionality

- **Piece Generation, Movement, and Rotation**: Random generation of tetrominoes, with movement and rotation controlled by the player.
- **Basic Scoring System**: Tracks and displays the player's score.
- **Simple Color Graphics**: Different colors for different tetrominoes.
- **Custom Background Image**: Colorful background with Sprite Logo
- **Audio Background Music (BGM)**: Plays background music during the game.

4. Implementation

4.1 SystemVerilog Implementation

For the detailed SystemVerilog module description, please refer appendix A.

4.1.1 Game Logic Controller

Implementation

- **Module Name**: game_logic.sv and ball.sv
- **Description**: Implements the core mechanics of the Tetris game, including piece generation, movement, collision detection, line clearing, and scoring.
- **Interface**:
  - **Inputs**:
    - clk: System clock.
    - reset: System reset signal.
    - key_input: Signals from the PS/2 keyboard.
  - **Outputs**:
    - grid_state: Current state of the game grid.
    - score: Current game score.

FSM Design

- **States**:
  1. **INIT**: Initializes game state.
  2. **IDLE**: Waits for user input.
  3. **MOVE**: Updates position of the active tetromino.
  4. **ROTATE**: Rotates the active tetromino.
  5. **COLLISION_CHECK**: Checks for collisions.
  6. **LINE_CLEAR**: Clears completed lines and updates the score.
  7. **GAME_OVER**: Ends the game when conditions are met.

Implementation in Ball.sv

- **Moving Block Definition**: Initial the moving block memory by choosing the block-type, then it will generate “L”, “I”, “T” and so on.
- **Moving Block Memory**: Using 25 registers to memory x and 25 registers to memory y for each block, which means a block can contain 25 squares at most.
- **Memory Block on Ground:** Mapping the moving block memory to a ground, and the ground will save it to prepare display the block.
- **Score Definition:** Using seven edges and six points to represent the number. Receive the number needed to print and output the squares need to be one.
- **Score Memory**: Using 34 registers to memory x and 34 registers to memory y for each block, which means a block can contain 25 squares at most.
- **Score on Ground:** Mapping the Score memory to a ground, and the ground will save it to prepare display the score.

- **Ground Display:** Using Draw-x, Draw-y and try to calculate the function transfer pixel coordinates to block coordinates. Then output is-ball to decide the color.
- **Initialization Controller Module:** Generate the first block of the game and allow operations and some other functions**.**
- **Conflict Detection Module:** Detect whether the movement is legal or illegal.
- **Touch Ground and Update Module:** Detect whether the block has touched the ground. If touched, the send signal to generate new block, update background.
- **Background Module:** Will memory the accumulated blocks on black, will update if one line is full.

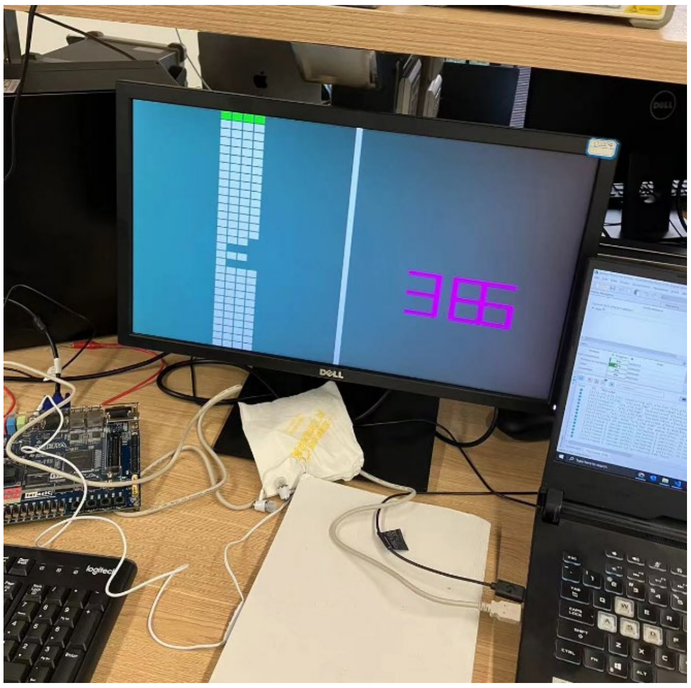


Fig 5: Left side are the accumulated blocks, moving block marked with green. Right side is the score, testing extreme cases for debug.

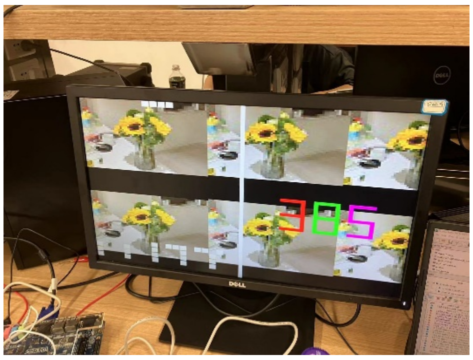
4.1.1 VGA Display Handler

Implementation

- **Module Name**: VGA controller.sv, Color_Mapper.sv
- **Description**: Manages the rendering of the game state on a VGA display.

Rendering Logic

- Uses double buffering to avoid flickering.
- Converts game grid data into VGA-compatible signals.
- Supports different colors for each tetromino type.



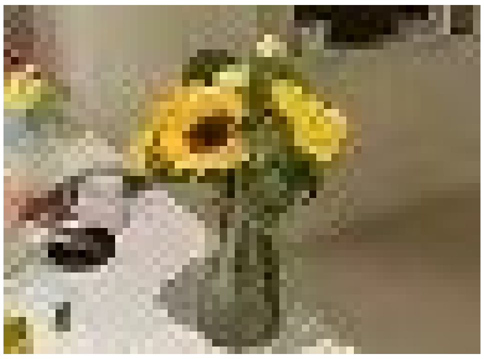


Fig 6 left: Background image test and Block debug mode, score display test.
Fig7 right: 80*60 pixel test image, a flower

4.1.1 Keyboard Handler

Implementation

- **Module Name**: keyboard_handler.sv
- **Description**: Interprets PS/2 keyboard inputs and converts them into control signals for the game logic.
- **Interface**:
    - **Inputs**:
        - clk: System clock.
        - ps2_data: Data from the PS/2 keyboard.
    - **Outputs**:
        - key_input: Control signals for game actions (A- left, D- right,W-rotate,S- drop).

Key Mapping

- WASD keys for movement.

4.1.4 **Audio background music**

Implementation

- **Module Name:** music_rom.sv, audio.sv
- **Description:** Play the BGM for the game via a pre-stored audio rom file in a loop.
- **Interface:**
    - **Inputs:**
        - **clk:** System clock.
        - **enable:** Signal to start or stop the BGM.
    - **Outputs:**
        - **audio_out:** Audio signal output to the speaker or audio hardware.

Audio Handling

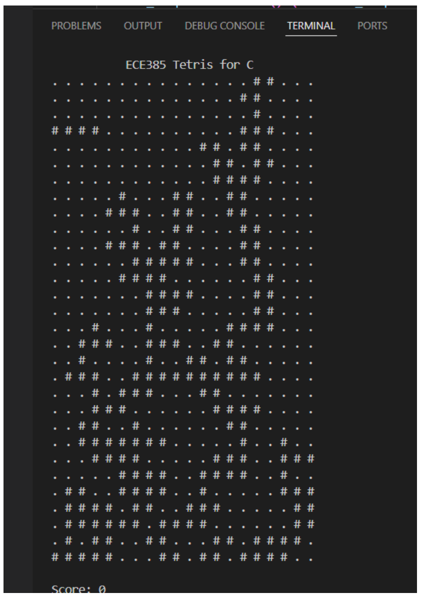
- **Music ROM:** The music data is stored in a ROM module(*resource/Tetris.txt)* which is read by the *music_rom.sv.* Transmitted to
- **Playback Control:** *Audio.sv* establishes hardware WM8731 Audio driver from the course website[12, 13, 14]. We use an open-source audio driver to read the data.
- **Looping:** Once initiated in the top-level file, the classic 'Tetris' BGM is played in a continuous loop to ensure uninterrupted background music during gameplay.

4.2 C Code Implementation

Initially, we plan to let the C program handles the overall game state, user input processing, and score tracking. This required communications with the SystemVerilog modules via memory-mapped I/O.
Based on najibghadri@Github's 200 lines of C implementation[12], we wrote a C program that manages each frame's Tetris game logic. As shown below, we achieved full functionality of moving, rotation, line clearing, and scoring

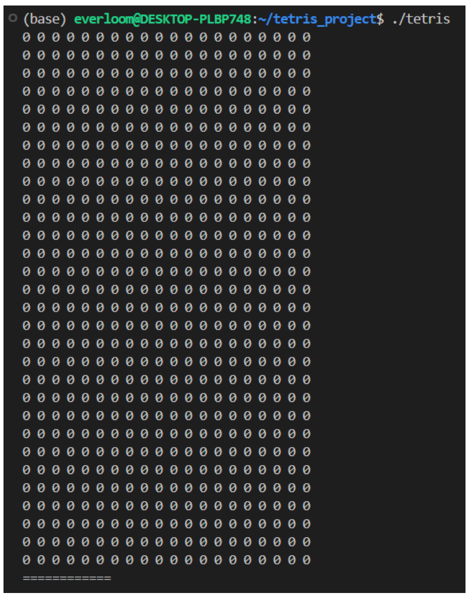
Game Logic Implementation in C

- **Move:** Handle left, right, and down movements.
- **Rotation:** Handle piece rotation.
- **Line Clear:** Detect and clear full lines.
- **Score:** Update the score based on cleared lines.



```
        ECE385 Tetris for C
. . . . . . . . . . . . . . . # # . . .
. . . . . . . . . . . . . . # # . . . .
. . . . . . . . . . . . . . . # . . . .
# # # # . . . . . . . . . . # # # . . .
. . . . . . . . . . . # # . # # . . . .
. . . . . . . . . . . . # # . # # . . .
. . . . . . . . . . . . # # # # . . . .
. . . . . # . . . # # . . # # . . . . .
. . . . # # # . . # # . . # # . . . . .
. . . . . . # . . # # . . . # # . . . .
. . . . # # # . # # . . . . # # . . . .
. . . . . . # # # # # . . . # # . . . .
. . . . . # # # # . . . . . . # # . . .
. . . . . . . # # # # . . . . # # . . .
. . . . . . . # # # . . . . . # # . . .
. . . # . . . # . . . . . # # # # . . .
. . # # # . . # # # . . # # . . . . . .
. . # . . . . # . . # # . # # . . . . .
. # # # . . # # # # # # # # # # . . . .
. . . # . # # # . . . # # . . . . . . .
. . . # # # . . . . . . # # # # . . . .
. . # # . . # . . . . . . # # . . . . .
. . # # # # # # # # . . . . # . . # . .
. . . # # # # . . . . . # # # . . # # #
. . . . . # # # # . . # # # # . . # . .
. # # . . # # # # . . # . . . . . # # #
. # # # # . # # . . # # # . . . . . # #
. # # # # # # . # # # # . . . . . . # #
. # . # # . . # # . . . # # . # # # # .
# # # # # . . . # # . # # . # # # # . .

Score: 0
```



```
(base) everloom@DESKTOP-PLBP748:~/tetris_project$ ./tetris
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
============
```

Fig 8 Left: C based Game Logic Implementation
Fig 9 Right: 0, 1 transmission for the 30*20 frame data

Challenges in Transferring Data to SystemVerilog

However, when we tried to turn that frame into the SystemVerilog, we met significant problems. We posted a query on Piazza . We learned that transferring large amounts of data from C to SystemVerilog via PIO is inefficient due to the 32-bit limitation and the high number of transfer operations required. The recommended approach for real-time applications is to use a dedicated memory copy unit to handle data transfers automatically.
Based on the feedback and the limited time before demo, we decided to skip the hardware linkage and implemented all the game logic in SystemVerilog.

5. Testing and Debugging

5.1 Testing Strategy

The system was tested using unit tests for individual modules, integration tests for combined modules, and system tests for the entire game. Both simulation and on-hardware testing were performed.

5.2 Design Resources and Statistics

Resource	Utilization
LUT (Logic Elements)	55,708 / 114,480 (49%)
DSP (Embedded Multiplier 9-bit elements)	10
Memory (BRAM bits)	1,346,832 / 3,981,312 (34%)
Flip-Flop	5091
Frequency	130.01MHz
Static Power	109.11mW
Dynamic Power	0.74mW
Total Power	180.90mW

Table 1: System Information of the full compilation.

5.3 Test Results

The test results confirmed the correct functionality of the game logic, input handling, and VGA display. Issues such as timing mismatches and synchronization errors were resolved during testing.

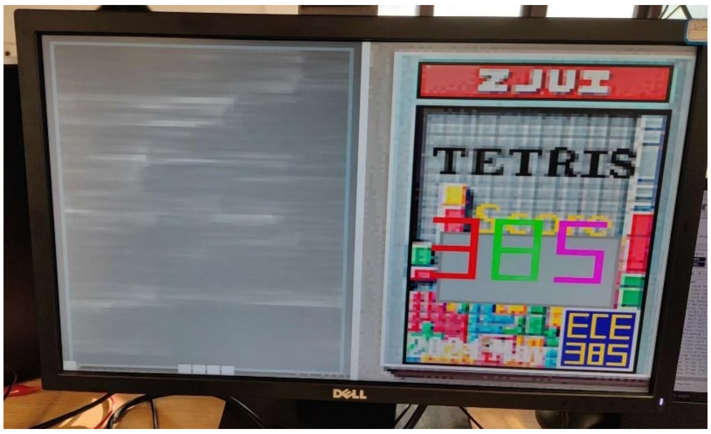


Fig 10: Final Demo image, with colorful background image, sprite icon, dynamic score, and game display. It is sad that we didn't took more photos / videos for our game, because we are hurry to write senior design thesis and take final exams (And we have to return the FPGA board).

5.4 Debugging

Score Image Mirror Display Issue: There was an error in calculating the x-axis coordinates, which led to incorrect pixel and block x-coordinate mapping. This has been resolved by rewriting the function that maps the x-coordinates of pixels and blocks.

Full Screen Color Flickering, No Blocks Visible: When adding more layers, there was an oversight in not properly checking whether the pixels belonged to a block before rendering. This caused all pixels to be rendered indiscriminately. The issue was resolved by implementing proper checks to determine if a pixel is part of a block before rendering.

Block Initialization Not Displaying: This was a timing issue. For certain specific sequences, the absence of synchronization caused the buffer to output directly without first reading the current block data, resulting in a blank display. This has been corrected by ensuring that the buffer properly reads the current block data before output.

Ineffective Boundary Collision: The timing discrepancy in the collision detection system led to transient collision signals that disappeared too quickly. This issue has been addressed by adjusting the timing mechanism to sustain collision signals for a sufficient duration.

Ineffective Boundary Collision: There was a logic error in the boundary check during shifts; it used a less-than-19 condition instead of less-than-or-equal-to-19. Additionally, certain special illegal blocks were not checked, which led to incorrect updates upon collisions. This has been corrected by adjusting the boundary check conditions.

Inability to Rotate Blocks: The issue with block rotation was due to the sequence of conditional checks in the rotation process. Illegal blocks were not preemptively excluded from the rotation updates, which caused them to be misidentified as out-of-bounds during collision checks, thus preventing rotation. This has been fixed by modifying the conditional logic to exclude illegal blocks from participating in rotation updates.

Simulated Keyboard Commands Not Working: This was caused by a mismatch in timing frequencies, resulting in keyboard commands not being detected at the appropriate time due to the absence of a rise signal. The issue has been resolved by adding necessary delays and implementing a release function to ensure commands are captured correctly.

Generation of Blocks Appeared Monotonous
Issue: Block generation was predictable monotonous due to reliance on a counter tied to the block generation frequency.
Solution: Enhanced the system by incorporating user input commands and additional counters. We should use random library in C to generate blocks. This introduced randomness and made the sequence of blocks more dynamic and engaging.

6. Challenges and Solutions

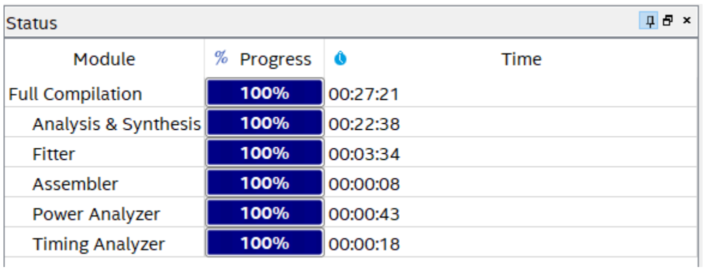
6.1 Technical Challenges

Timing and Synchronization:

Timing Constraints:
Tetris is not a game that requires real-time display updates; however, the system's automatic block descent and user key response frequencies are significantly lower than those of the initial clock (CLK) or even the frame clock (Frame clk). Operating checks or updates directly under these two frequencies can greatly reduce efficiency and may even lead to synchronization issues.
Adjusted Refresh Rates:
For the display refresh and automatic block descent, a refresh frequency of 0.5 seconds has been implemented. This slower rate ensures that the gameplay remains smooth without overwhelming the processing capabilities of the system. For user inputs, a more responsive frequency of 0.05 seconds has been adopted to ensure that the game accurately captures and responds to player actions promptly.
Sequence and Update Considerations:
Certain modules that perform real-time scanning require at least double the frequency of input-output modules to prevent synchronization problems. If not adequately managed, this discrepancy can lead to signal instability, flickering, and other unusual bugs. The always_comb blocks in high-frequency and latency-sensitive modules must be carefully managed, as they may not be able to update simultaneously without causing operational issues. It is advisable to optimize the logic by moving critical sections to always_ff blocks or by implementing locking mechanisms to ensure updates occur in an orderly and rule-abiding manner.
High-Frequency Access to Low-Frequency Operations:
Given that user input frequency can be relatively high and the desire is for the system to be highly responsive to such inputs, it is essential not to rely solely on inputs captured at the rising edge of the CLK.A locking structure has been designed that triggers and holds inputs for a set duration until they are read and processed. This lock can be released or managed based on specific rules after the update, ensuring inputs are neither missed nor excessively delayed.
These solutions address the core challenges of managing a game like Tetris where timing and synchronization are critical for gameplay integrity and user experience. By adjusting operational frequencies and enhancing control over data processing sequences, the game becomes more stable and responsive, significantly improving player interaction and satisfaction.

Extra Long Compilation Time:

- For the final version of our game, it took 27.21 minutes to complete one full compilation. This is mainly caused by the music module loading and background image rendering. This posed great difficulty during debugging.


Status

Module	Progress	Time
Full Compilation	100%	00:27:21
Analysis & Synthesis	100%	00:22:38
Fitter	100%	00:03:34
Assembler	100%	00:00:08
Power Analyzer	100%	00:00:43
Timing Analyzer	100%	00:00:18

Fig 11: It took nearly half an hour to compile one run, extremely painful for debugging. Acceleration and task segmentation is very necessary for FPGA programming.

- **Incremental Compilation:** Compiled only modified portions of the design to reduce subsequent compilation times.
- **Task Segmentation:** Divided the design into smaller, manageable modules for independent testing and validation.
- **Optimized Synthesis Settings:** Fine-tuned synthesis and placement settings in Quartus Prime for efficient resource utilization.
- **Parallel Processing:** Utilized maximum available processors for parallel processing to speed up compilation.
  These strategies effectively mitigated the impact of long compilation times, enhancing productivity and efficiency in the FPGA-based Tetris game development.

6.2 Project Management Challenges

- **Time Constraints:** One of the primary challenges was managing time effectively to meet project deadlines. The overlap between ECE385 and ECE445 Senior Design significantly limited our available time. As team leaders, we found it challenging to allocate sufficient time to complete all aspects of the code, despite our best efforts on the last week.
- **Workload Management:** Balancing the workload between team members and ensuring consistent progress was another major challenge. With the dual responsibilities of leading our teams in both courses, we had to carefully manage our tasks to maintain steady progress on the project while fulfilling our other obligations. Despite these challenges, we strived to achieve the best possible outcomes.

7. Future Work

7.1 Planned Enhancements

When we have time, we would like to add the following functionality, and we greatly advise future students work on these directions:

- **Increasing Levels of Difficulty**: Game speed increases with each level.
- **High Score Table**: Stores and displays high scores using non-volatile memory.
- **Enhanced Graphical Effects**: Includes animations for line clears and other events.
- **Game Loading from a CD**: Inspired by console game loading mechanisms.
- **Multiplayer Mode**: Allows multiple players to play simultaneously.

- **Improved Graphics**: Adding more detailed and visually appealing graphics.
- **Network Multiplayer**: Implementing a networked multiplayer mode.

7.2 Potential Improvements

- **Performance Optimization**: Enhancing the performance of the game by optimizing the hardware and software design.
- **User Interface**: Improving the GUI user interface for a better gaming experience.

8. Conclusion

Despite time constraints, our project successfully implemented 80% functionality of a real-time Tetris game on an FPGA, including features such as piece generation, movement, rotation, collision detection, background image, background music and a scoring system. This enables us to win the second prize in final competition.

The final project provided a comprehensive learning experience in digital system design, allowing us to construct a game independently. We believe it would be beneficial if the course were taken in the junior year or fall semester, providing more time for project development!

This report outlines the design, implementation, and interfacing of the FPGA-based Tetris game. By leveraging the Avalon Bus for IP communication and integrating SystemVerilog and C components, we achieved a real-time, interactive gaming experience on the FPGA Development Board. Our project not only demonstrates our technical understanding in ECE385: Digital Systems but also showcases creativity and innovation, positioning us well for the Eaton Competition. This project represents a modern reimagining of a classic game, blending traditional gameplay with digital design techniques.

9. References

[1] OpenAI, "DALL·E 3," OpenAI, 2024. [Online]. Available: https://openai.com/index/dall-e-3/.
[2] TinyVGA, "VGA Signal Timing for 640x480 @ 60Hz," TinyVGA, 2024. [Online]. Available: http://tinyvga.com/vga-timing/640x480@60Hz.
[3] weixin_44830487, "基于 fpga 的俄罗斯方块," CSDN, 2023. [Online]. Available: https://blog.csdn.net/weixin_44830487/article/details/115972133.
[4] FPGA4Fun, "Pong Game," FPGA4Fun, 2024. [Online]. Available: https://www.fpga4fun.com/PongGame.html.
[5] Terasic Inc., "Altera DE2-115 Development and Education Board," Terasic, 2024. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=502&PartNo=4#.
[6] Lan Tian, "Cyclone IV FPGA Development Bugs Resolve," Lan Tian @ Blog, 2024. [Online]. Available: https://lantian.pub/article/modify-computer/cyclone-iv-fpga-development-bugs-resolve.lantian/.
[7] H. Ye, "FPGA Tetris Game," GitHub, 2016. [Online]. Available: https://github.com/hanchenye/FPGA-tetris?tab=readme-ov-file.
[8] KTTECH. (2017, January 31). ECE 385 Lab 8: SoC with USB and VGA Interface in SystemVerilog. Retrieved from https://kttechnology.wordpress.com/2017/03/10/ece-385lab-8-soc-with-usb-and-vga-interface-in-systemverilog/. Teaching Assistant Blog
[9] ECE385 Faculty. (n.d.). Lab 8 description
[10] ECE385 Faculty. (n.d.). Introduction to SystemVerilog (pdf)
[11] ECE385 Faculty. (n.d.). Introduction to Quartus Prime in the lab manual.
[12] N. Ghadri, "Tetris in 200 Lines of C Code," GitHub, 2024. [Online]. Available: https://github.com/najibghadri/Tetris200lines/.
[13] Z. Cheng, "Sound on the DE2 and SDRAM," ECE 385 – Digital Systems Laboratory, University of Illinois at Urbana-Champaign, Spring 2024. [Online]. Available: https://courses.engr.illinois.edu/ECE385/sp2024/.
[14] K. Roy, "audio_interface VHD Driver for DE2 WM8731 Audio Codec," University of Illinois at Urbana-Champaign, April 23, 2010. [Online]. Available: https://courses.engr.illinois.edu/ECE385/sp2024/.

10. Appendices

Module Name:
ball
Inputs:
Clk: 50 MHz main clock signal.
Reset: Active-high reset signal.
frame_clk: Clock signal indicating a new frame (~60Hz).
DrawX, DrawY: Current pixel coordinates.
keycode: 8-bit keycode from input device.
Outputs:
is_ball [4:0]: Signal to indicate whether the current pixel is part of a ball or the background.
Internal Modules and Components:
Memory Arrays: Used for storing block and score data.
choose_block: Selects and positions blocks based on the type and control signals.
choose_num_score: Dynamically generates numbers based on inputs for score displays.
memory_block_on_moving_block_ground: Manages memory for moving blocks.
memory_num_on_score_ground: Manages memory for numerical scoring.
gravity_moving_block: Calculates new block positions based on gravity simulation.
keypress_generator: Generates signals based on keypress inputs.
draw_ground: Draws blocks on the VGA display.
draw_score: Draws scoring information on the VGA display.
Description:
The "ball" module manages a Tetris-like game where blocks move according to user input and gravity. It handles various game functions including block movement, rotation, and scoring, displaying these on a VGA screen. The module integrates multiple custom submodules for specific tasks like handling user inputs, updating and managing block positions, generating numerical scores, and rendering graphics.
Key Functionalities:
Block Movement and Rotation: Controlled by user inputs translated through keycodes.
Score Display: Numerical scores are generated and updated based on game progress.
Graphics Rendering: Visual representation of blocks and scores using VGA outputs.
Collision and Boundary Checks: Ensures blocks stay within game boundaries and handles interactions.
Detailed Operation:
Initialization:
Loads initial block positions and setups.
Resets game states and memories on a high reset signal.
User Interaction:
Processes key inputs to rotate and move blocks.
Updates positions and states based on these inputs.
Game Logic:
Gravity affects the block positions periodically, simulating falling blocks.
Collision detection prevents blocks from overlapping and guides stacking.

Score calculation based on line completion.
Rendering:
Draws the current state of blocks and scores on the VGA display.
Updates colors and positions based on the game state.
Challenges and Considerations:
Efficiency: Handling multiple game functions efficiently within the constraints of FPGA timing.
Responsiveness: Ensuring the game responds quickly to user inputs without lag.
Scalability: Structuring the code to allow easy modifications, such as changing game rules or adding new features.

Module Name:
timer
Inputs:
wire clk: Clock signal.
wire Reset: Reset signal.
already_touch_ground: Indicates whether the ground has been touched.
Outputs:
reg enable_choose: Control signal to enable choosing.
reg enable_control: Control signal for enabling control operations.
block_type: Indicates the type of block.
Description:
The "timer" module is designed to manage timing and control operations based on a clock signal and a reset condition. It features a counter for tracking time-related events and outputs signals to enable certain functionalities based on the ground contact condition and internal timing.
Operation:
Reset Condition: When the Reset signal is high, the counter is reset to zero, the enable_choose output is set to 0, and block_type is set to 0.
Counter Operations: The module increments the counter on each rising edge of the clock unless a reset condition is met.
Block Type Setting: The block_type is determined by the modulo 7 of the current counter value.
Control Enablement:
On the first clock cycle after the counter is reset, enable_choose is set to 1.
If already_touch_ground is true after the first clock cycle, enable_choose remains 1 and enable_control is set to 0.
If already_touch_ground is false, enable_choose is set to 0. If the counter is less than 63, it continues to increment, and enable_control is set to 1 to manage overflow conditions.

Module Name:
gravity_moving_block
Inputs:
logic frame_clk: Frame clock signal.
block_x [24:0]: Array of X coordinates of the blocks (25 blocks total).
block_y [24:0]: Array of Y coordinates of the blocks (25 blocks total).
background [29:0]: A 20-bit array representing background information for 30 vertical positions.

Outputs:
conflict: Conflict signal, indicates if there is a movement obstruction.
block_try_x [24:0]: X coordinates of the blocks after applying gravity effect.
block_try_y [24:0]: Y coordinates of the blocks after applying gravity effect.
Description:
The "gravity_moving_block" module processes the movement of blocks based on a gravity simulation. It takes as input the current positions of blocks and calculates their potential new positions after gravity has been applied. It checks for conflicts such as block collision or boundary limits, using a background matrix to determine allowable movements.
Operation:
Initial Settings: The rotation center block (index 0) maintains its X coordinate but is adjusted vertically to simulate falling due to gravity.
Conflict Detection:
The module checks each block's position against the background matrix. If a block is at the lowest possible Y coordinate (block_y[i] == 0) or if the background directly below the current position is occupied (background[block_y[i]-1][block_x[i]] == 1), the conflict signal is set to 1, indicating an obstruction.
Position Adjustment:
For blocks not on the boundary (within Y coordinate 0 to 29), the module updates the Y coordinate to one position lower to simulate falling, while keeping the X coordinate the same.
If a block's Y position is out of bounds or an invalid position is detected, the coordinates are set to an invalid state (111111 binary).

Module Name:
keypress_generator
Inputs:
clk: Clock signal.
reset: Reset signal.
Outputs:
keypress: 10-bit output representing the keypress code.
Description:
The "keypress_generator" module simulates keyboard operation for testing purposes. It cyclically generates keypress codes from a predefined list at a specified interval, using a clock signal to pace the generation. The system clock frequency is assumed to be 50 MHz, and the module manages keypress output timing using a counter.
Operation:
Initialization: On reset, the counter, index, and keypress outputs are zeroed, and the module is set to a generating state.
Counter Logic: With each clock pulse:
If the counter has not reached its maximum value (20), it increments, and no keypress code is output.
Once the counter reaches 20, it resets to zero, the keypress code from the current index in the KEYS array is output, and the index is incremented.
The keypress codes are all preset to the value 10'd4 for simplicity in this setup.

Index Cycling: The index cycles through the KEYS array:
If the index reaches the last element (index 10), the module resets the index to continue generating keypresses cyclically.
If not, the module proceeds to increment the index, allowing the sequence to repeat continuously.

Module Name:
allow_control_check
Inputs:
block_x [24:0]: Array of X coordinates of the blocks (25 blocks total).
block_y [24:0]: Array of Y coordinates of the blocks (25 blocks total).
enable_check_conflict: Control signal to enable the conflict check.
Outputs:
conflict: Logic signal indicating if there is a conflict.
Description:
The "allow_control_check" module is designed to verify the placement of blocks within defined boundaries. It checks if each block's coordinates fall within allowable ranges, using an enable signal to initiate the checking process. The module aims to ensure that no block is positioned outside predefined boundaries, thus preventing conflicts within a controlled environment.
Operation:
Initialization: Initially, the conflict signal is set to 0, indicating no conflict.
Conflict Checking:
Upon the positive edge of enable_check_conflict, the module resets the conflict signal to 0 as a default state.
The module iterates through each block's coordinates:
It first verifies that the coordinates are valid and not set to the invalid state (6'b111111).
It then checks if both the X and Y coordinates of each block are within the valid ranges (X should be 0 to 19 and Y should be 0 to 29).
If any block is found outside these ranges, the conflict signal is set to 1, indicating a conflict, and the checking loop is exited immediately.

Module Name:
control_block
Inputs:
block_x [24:0]: Array of X coordinates of the blocks (25 blocks total).
block_y [24:0]: Array of Y coordinates of the blocks (25 blocks total).
control_command: A 3-bit input to control block movements: 00 for rotation, 01 for moving right, 10 for moving left, 11 for moving down.
ask_control: A logic signal that triggers the control sequence.
background [29:0]: A 20-bit wide array representing background state for collision detection.
Outputs:
block_try_x [24:0]: X coordinates of the blocks after attempted movement.
block_try_y [24:0]: Y coordinates of the blocks after attempted movement.
conflict: A logic output indicating if a movement results in a conflict.
Description:

The "control_block" module is designed to control movements and rotations of Tetris blocks based on input commands. It adjusts the position of each block according to the specified movement direction while checking for collisions or boundary exceedances using the background state. This module plays a critical role in game dynamics by handling the logic for block interactions.

Operation:

Rotation (control_command = 00):

Rotates blocks around a center by 90 degrees clockwise. Checks for boundary exceedances and collisions with the background. If any block after rotation falls outside the allowed area or collides, it sets the conflict signal.

Move Right (control_command = 01):

Attempts to move all blocks one unit to the right. It checks for right boundary exceedance and background collisions. If a move is not possible without exceeding the boundary or causing a collision, it sets the conflict signal.

Move Left (control_command = 10):

Moves blocks one unit to the left. Similar checks as moving right are performed for left boundary and background collisions.

Move Down (control_command = 11):

Moves blocks one unit downwards. Checks for bottom boundary exceedance and collisions with the background. Sets conflict if a move is not possible.

No Movement (default):

If an unrecognized command is received, no change in position occurs.

Module Name:

draw_ground

Inputs:

frame_clk: Frame clock signal.

DrawX, DrawY: Current pixel coordinates.

moving_block_ground[29:0]: Memory array storing the state of moving blocks, where each row is 20 bits wide.

is_ball_color: A 5-bit input used to determine the color of the ball.

Outputs:

is_ball: A 5-bit output signal indicating whether the pixel is part of a ball.

Description:

The "draw_ground" module is responsible for determining if a given pixel corresponds to a part of a ball based on its location within a grid of blocks. It utilizes pixel coordinates, a memory array of moving blocks, and a color identifier to perform this operation. The module calculates block indices from the pixel coordinates and checks if the pixel lies within the interior of a block (excluding borders) to decide the output.

Operation:

Block Index Calculation:

block_index_x: Calculated by dividing DrawX by the width of each block (PIXELS_PER_BLOCK), to identify the horizontal block position.

block_index_y: Calculated similarly for the vertical position, adjusted for the screen height and inverted (since screen coordinates start from the top).

Pixel Classification:
Checks if the pixel is within the display area (DrawX < 320 and block_index_y < 30).
Determines if the pixel is not on a block border by ensuring it does not lie on the outermost pixels of a block.
If the conditions are met, the bit from moving_block_ground corresponding to the block index is multiplied by is_ball_color to determine the is_ball value, indicating whether the pixel is part of the ball.
Pixels on borders or outside the display area are set to zero, indicating they are not part of the ball.

Module Name:
memory_block_on_moving_block_ground
Inputs:
Clk: Clock signal.
write_enable: Enable signal for writing to the memory.
block_x [24:0]: Array of X coordinates for the blocks (25 blocks total).
block_y [24:0]: Array of Y coordinates for the blocks (25 blocks total).
Outputs:
moving_block_ground [29:0]: Memory array of moving blocks, each row 20 bits wide, serving as an inout to handle internal writes and external reads.
Description:
The "memory_block_on_moving_block_ground" module manages a memory array that tracks the positions of moving blocks on a virtual ground. This module uses input coordinates of blocks to update the memory array, marking the presence of blocks within a predefined boundary. The memory is cleared and updated every clock cycle when writing is enabled, ensuring that it accurately reflects the current positions of the blocks.
Operation:
Memory Clearing:
At each positive edge of the clock, if write_enable is active, the module clears the entire memory (moving_block_ground) to reset the state of the ground, preparing it for new data.
Memory Update:
The module iterates over the block coordinates:
It checks if the coordinates are within the valid range (block_x[i] < 20 and block_y[i] < 30) and are not set to an invalid state (6'b111111).
For valid coordinates, it sets the corresponding bit in the moving_block_ground array to 1, indicating the presence of a block at that position.

Module Name:
memory_block_on_background
Inputs:
Clk: Clock signal.
block_x [24:0]: Array of X coordinates for the blocks (25 blocks total).
block_y [24:0]: Array of Y coordinates for the blocks (25 blocks total).
background [29:0]: A 20-bit wide array representing the existing background state, each element corresponding to a row of the background.

Outputs:
moving_block_ground [29:0]: Memory array of moving blocks, each row 20 bits wide. This output serves as an inout to handle both internal writes and external reads.
Description:
The "memory_block_on_background" module is designed to update a memory array that represents the positions of moving blocks overlaid on a predefined background. The module integrates the positions of dynamic blocks with a static background, ensuring that the memory array reflects a composite view of both elements. This is particularly useful in applications such as games or simulations where background and dynamic elements interact.
Operation:
Background Initialization:
At each positive edge of the clock, the module first copies the existing background state into the moving_block_ground memory. This step ensures that the dynamic block positions are overlaid onto the current background.
Memory Update for Block Positions:
The module iterates over the block coordinates:
It checks if each block's coordinates are within the valid range (block_x[i] < 20 and block_y[i] < 30) and ensures they are not set to an invalid state (6'b111111).
For valid coordinates, the corresponding bit in the moving_block_ground memory is set to 1, indicating the presence of a block at that location.

Module Name:
choose_block
Inputs:
Clk: Clock signal.
block_type: A 6-bit input specifying the type of block to be chosen.
enable_choose: A logic signal to enable the block selection process.
Outputs:
block_x [24:0]: Array of X coordinates for the block (25 blocks total).
block_y [24:0]: Array of Y coordinates for the block (25 blocks total).
Description:
The "choose_block" module is responsible for selecting a specific type of block based on the input block_type and setting its coordinates. This module uses a case statement to assign predefined coordinates to each block type when the selection is enabled. This approach is particularly useful in applications like games or simulations where different block types need to be generated and manipulated dynamically.
Operation:
Initialization:
On every positive edge of the clock when enable_choose is high, the module initializes all coordinates in block_x and block_y to an invalid state (6'b111111), indicating no block is currently active.
Block Selection:
Depending on the block_type provided, the module assigns specific coordinates to the first five blocks (from a total of 25) for simplicity. These coordinates are hardcoded for different block

configurations such as "I," "L," "Z," "O," "S," "T," "J," and others, simulating typical Tetris-like game pieces.
Configuration Examples:
I Block: Straight line configuration.
L Block: Right-angled configuration.
Z Block: Zigzag pattern.
O Block: Square shape.
S Block: Mirror zigzag pattern.
T Block: T-shape configuration.
J Block: Mirrored L-shape.

Module Name:
draw_score
Inputs:
frame_clk: Frame clock signal.
DrawX_in, DrawY_in: Current pixel coordinates, inputs to the module.
score_ground[59:0]: Memory array representing the score display area, where each row is 40 bits wide.
is_ball_color: A 5-bit input used to determine the color intensity or identification of a pixel.
Outputs:
is_ball: A 5-bit output signal indicating whether the pixel is part of the score display (interpreted as "ball" in the context).
Description:
The "draw_score" module is designed to manage the graphical representation of a scoring area in a display system. It uses a memory array that contains the layout of a score, adjusting for pixel coordinates offset from a central point to display this information on a specific part of the screen.
Operation:
Coordinate Adjustment:
Adjusts the input DrawX_in by subtracting 320 to align the drawing operations to a specific region of the screen, likely the right half.
DrawY_in remains unchanged as the vertical coordinate does not require adjustment for this operation.
Block Index Calculation:
block_index_x: Derived by dividing the adjusted DrawX by PIXELS_PER_BLOCK, which determines the horizontal position within the score display.
block_index_y: Computed similarly for the vertical position, adjusted for the total screen height and reversed to align with graphical coordinates starting from the top.
Pixel Classification:
Determines if the pixel is at the boundary of the score display area. If the pixel falls on the left boundary (block_index_x == 0), it directly sets is_ball to 1, indicating part of the display.
Checks if the pixel falls within the graphical boundaries defined by BLOCKS_PER_ROW and BLOCKS_PER_COL. If within bounds, the pixel's score display bit (score_ground[block_index_y][block_index_x]) is multiplied by is_ball_color to determine the is_ball output.

Pixels outside the defined score display area are set to zero, indicating they are not part of the score display.

Module Name:
memory_num_on_score_ground
Inputs:
Clk: Clock signal.
block_x [33:0]: Array of X coordinates for the blocks (34 blocks total).
block_y [33:0]: Array of Y coordinates for the blocks (34 blocks total).
Outputs:
score_ground [59:0]: Memory array representing the scoring area, with each row being 40 bits wide. This output serves as an inout to handle both internal writes and external reads.
Description:
The "memory_num_on_score_ground" module is tasked with updating a memory array that depicts a scoring area on a display, based on dynamic block positions provided via the block_x and block_y arrays. This module is particularly useful in applications like games or digital signage where numbers or other symbols need to be dynamically rendered on a scoreboard or similar display.
Operation:
Memory Clearing:

At each positive edge of the clock, the module first clears the entire score_ground memory to reset the scoring area, preparing it for new data. This clearing operation sets all bits in each row to 0.
Memory Update for Block Positions:
Iterates over the array of block coordinates:
Checks each block's coordinates to ensure they are within the valid range (block_x[i] < 40 and block_y[i] < 60) and ensures they are not set to an invalid state (6'b111111).
For valid coordinates, the corresponding bit in the score_ground memory is set to 1, marking the presence of a block at that location.

Module Name:
choose_num_score
Inputs:
Clk: Clock signal.
block_type: A 6-bit input specifying the numeric type of block to be chosen.
init_x: The initial x-coordinate for the starting position of the number block.
init_y: The initial y-coordinate for the starting position of the number block.
Outputs:
block_x [33:0]: Array of X coordinates for the blocks (34 blocks total), defining the structure of the number.
block_y [33:0]: Array of Y coordinates for the blocks (34 blocks total), defining the structure of the number.
Description:
The "choose_num_score" module dynamically constructs numeric displays based on a given type

of number. It uses a case statement to decide which number (0 through 9) is to be displayed based on the block_type and assigns specific coordinates to elements of the number, such as edges and points. These coordinates outline the shape of the number using predefined patterns for each numeral.

Operation:

Initialization and Selection:

On each positive clock edge, the module first initializes logic variables that control the display of points and edges for numbers.

Depending on the block_type, various segments of the number (like left and right points, top and bottom edges) are activated.

Coordinate Assignment:

For each activated segment or point, coordinates are calculated and assigned based on the initial init_x and init_y values. The placement considers the typical structure of digital numbers seen on displays like calculators or digital clocks.

For parts of the number not in use based on the selected block_type, coordinates are set to an invalid state (6'b111111), effectively turning off these blocks.

Specific Cases:

Number 0: Forms a complete enclosure using all outer edges and points.

Number 1: Activates only the right edges and points to form the numeral.

Number 2: Uses a combination of top, middle, bottom, left bottom, and right top edges to create the shape.

More complex numbers use combinations of these principles to outline the respective numerals.

Module Name:

color_mapper

Inputs:

is_ball [4:0]: Input signal indicating whether the current pixel is part of a ball (with different potential values specifying different types of balls) or the background.

DrawX [9:0], DrawY [9:0]: Current pixel coordinates.

Outputs:

VGA_R [7:0]: Red component of the VGA output.

VGA_G [7:0]: Green component of the VGA output.

VGA_B [7:0]: Blue component of the VGA output.

Description:

The "color_mapper" module is designed to determine the color of pixels for display on a VGA monitor based on the nature of the pixel (whether it is part of a ball or background). The module uses arrays to define background colors and sets colors for different types of balls using predefined values. It reads background color values from external files, allowing for a customizable and potentially complex background design.

Initialization:

Memory Initialization: At the start, the module loads color data for the background from external hexadecimal files (ZJUIR.txt, ZJUIG.txt, ZJUIB.txt) into arrays (R_bg, G_bg, B_bg), which are used later to assign background colors based on pixel coordinates.

Operation:

Index Calculation:
Computes an index (idx) for accessing the color arrays based on the current pixel coordinates, downscaled by a factor of 4 (assumes a reduction in resolution for background detailing).
Color Assignment:
Ball Detection and Coloring:
If is_ball indicates a specific type of ball (through distinct binary values), the module assigns colors accordingly:
5'b00001: White (all color channels set to maximum).
5'b00010: Red (only the red channel is set).
5'b00011: Green (only the green channel is set).
5'b00100: Custom color (in this case, pink—red and blue channels set).
Background Coloring:
If none of the specific is_ball conditions are met, the pixel is considered part of the background. The color for these pixels is fetched from the preloaded background color arrays using the calculated idx.
VGA Output:
The determined red, green, and blue values are directly assigned to the VGA output ports VGA_R, VGA_G, and VGA_B.

Module Name:
hpi_io_intf
Inputs:
Clk
Reset from_sw_address from_sw_data_out from_sw_r from_sw_w from_sw_cs from_sw_reset
Outputs:
from_sw_data_in OTG_ADDR OTG_RD_N OTG_WR_N OTG_CS_N OTG_RST_N OTG_DATA
Description:
The hpi_io_intf module interfaces the Human Processor Interface (HPI) with the OTG (On-The-Go) device, handling signals and data transfer between a processor (like a NIOS II) and the OTG chip. It manages the read and write operations through an inout data bus (OTG_DATA), which requires careful signal control to ensure proper data handling and integrity.
Operation:
The module continuously monitors the control signals and the address signal . Based on these inputs, it manages the data flow to and from the OTG chip.
Data sent to the OTG chip is buffered in a register before being sent to the OTG_DATA bus, ensuring that the data bus is driven only during appropriate write operations.
The OTG_DATA bus is set to high impedance (tri-state) when not actively driven by the NIOS II processor to prevent bus conflicts and ensure proper read operations from the OTG chip.

Module Name: lab8
Inputs:
CLOCK_50 (50 MHz clock signal from the FPGA)

KEY[3:0] (Four-bit input for keys, where bit 0 is used as a reset)

Outputs:

HEX0 (7-segment display HEX0)

HEX1 (7-segment display HEX1)

VGA_R (8-bit output for VGA red channel)

VGA_G (8-bit output for VGA green channel)

VGA_B (8-bit output for VGA blue channel)

VGA_CLK (VGA clock signal)

VGA_SYNC_N (VGA synchronization signal)

VGA_BLANK_N (VGA blanking signal)

VGA_VS (VGA vertical synchronization signal)

VGA_HS (VGA horizontal synchronization signal)

OTG_ADDR (2-bit output for CY7C67200 USB controller address)

OTG_CS_N (Chip select for CY7C67200, active low)

OTG_RD_N (Read signal for CY7C67200, active low)

OTG_WR_N (Write signal for CY7C67200, active low)

OTG_RST_N (Reset for CY7C67200, active low)

DRAM_ADDR (13-bit output for SDRAM address)

DRAM_BA (2-bit output for SDRAM bank address)

DRAM_DQM (4-bit output for SDRAM data mask)

DRAM_RAS_N (SDRAM row address strobe, active low)

DRAM_CAS_N (SDRAM column address strobe, active low)

DRAM_CKE (SDRAM clock enable)

DRAM_WE_N (SDRAM write enable, active low)

DRAM_CS_N (SDRAM chip select, active low)

DRAM_CLK (SDRAM clock)

Inout:

OTG_DATA (16-bit bidirectional data bus for CY7C67200 USB controller)

DRAM_DQ (32-bit bidirectional data bus for SDRAM)

Description:

The lab8 module serves as the main integration point for the Lab 8 project in the ECE 385 course. It connects the FPGA to various peripherals including a VGA display and a USB controller, and interfaces with SDRAM for data storage.

Operation:

The module initializes and manages the signals for interfacing with a VGA display and the CY7C67200 USB controller. It also handles communication with SDRAM utilized by the Nios II processor for data storage and retrieval.

The VGA_controller, ball, color_mapper, and HexDriver modules are instantiated to manage the display and interaction elements of the project. Key signals and data are routed between the FPGA and the peripherals, ensuring synchronized operations for display and USB communication.

Module Name: VGA_controller

Inputs:

Clk (50 MHz clock)

Reset (Active-high reset signal)

VGA_CLK (25 MHz VGA clock input, used for timing critical operations)
Outputs:
VGA_HS (Horizontal sync pulse, active low)
VGA_VS (Vertical sync pulse, active low)
VGA_BLANK_N (Blanking interval indicator, active low)
VGA_SYNC_N (Composite Sync signal, active low; not used in this lab but required for hardware)
DrawX (Horizontal coordinate output, 10 bits)
DrawY (Vertical coordinate output, 10 bits)
Description:
The VGA_controller module manages the generation of timing and synchronization signals for a VGA display operating at a resolution of 640x480 pixels. This includes generating horizontal and vertical sync pulses, managing the drawing coordinates for pixels on the display, and controlling the timing for when the display is active or in a blanking interval.
Operation:
The module operates by counting pixels (h_counter) and lines (v_counter) to manage the synchronization pulses and the active display time for a standard VGA display. These counters are incremented on every positive edge of the VGA clock (VGA_CLK), which should be 25 MHz.
Horizontal Timing: The horizontal sync pulse (VGA_HS) is generated by checking the h_counter value and is active low for 96 clock cycles between 656 and 752.
Vertical Timing: The vertical sync pulse (VGA_VS) is active low for 2 lines, specifically when v_counter is between 490 and 492.
Blanking Interval: The VGA_BLANK_N signal is used to indicate when the VGA output should be blanked (not displaying pixels). It is active when both h_counter and v_counter are within the displayable area (0-639 for horizontal and 0-479 for vertical).

USB protocol
USBRead: USBRead is used to retrieve data from the registers in the CY7C67200 USB Controller. It employs IO_Write to set the desired address and then utilizes IO_Read to fetch and return the data stored at that address.
USBWrite: USBWrite facilitates writing data to the internal registers of the USB controller. It achieves this by calling IO_Write twice: once to specify the register address and a second time to write the data to that address.
IO_Read: IO_Read returns 16-bit data from the otg_hpi_data register after updating the otg_hpi_address to the specified 8-bit input address. This function is critical for reading operations where precise control over the USB controller's address space is necessary.
IO_Write: IO_Write sends data to the USB Controller registers. It takes two inputs, address and data, and writes them respectively to the address and data pointers in the USB controller.