CS256 Class Project Report: FPGA Tetris

Mohammad Alkhalifah
mohammad.alkhalifah@kaust.edu.sa

Mustafa Albahrani*
mustafa.albahrani@kaust.edu.sa

December 11, 2025

*Joint project; however, this report is an individual submission as per the guidelines

Contents

1 Introduction **4**
- 1.1 What is Tetris? . . . 4
- 1.2 Why Implement Tetris on an FPGA? . . . 4
- 1.3 Project Overview . . . 4

2 VGA Signaling and Timing Generation **4**
- 2.1 Protocol Overview . . . 5
- 2.2 Circuit Implementation . . . 5
- 2.3 Determining Pixel Coordinates . . . 5

3 Module Architecture **6**
- 3.1 Resource Utilization . . . 7

4 Design Description and Drawing Logic **8**
- 4.1 Drawing Strategy: Pipelined Renderer . . . 8
- 4.2 4-Stage Pipeline Details . . . 8
- 4.3 Clock Architecture . . . 8
- 4.4 Input Processing . . . 9
  - 4.4.1 PS/2 Keyboard Protocol . . . 11
- 4.5 Game Logic . . . 11
  - 4.5.1 Randomizer (7-Bag System) . . . 11
  - 4.5.2 Rotation System (SRS) . . . 11
  - 4.5.3 Scoring and Level Progression . . . 12
  - 4.5.4 T-Spin Detection . . . 12
  - 4.5.5 Finite State Machine (FSM) . . . 12
  - 4.5.6 Ghost Piece . . . 13
- 4.6 Display Logic . . . 14
  - 4.6.1 Sprite System Details . . . 14
- 4.7 Game Logic Integration . . . 15

5 Testing and Verification **15**
- 5.1 Testbench Summary . . . 16
- 5.2 Game Control Simulation (`tb_game_control`) . . . 16
- 5.3 Input Manager Test (`tb_input_manager`) . . . 16
- 5.4 Rotation Test (`tb_rotate_tetromino`) . . . 17
- 5.5 Hold Feature Test (`tb_hold_feature`) . . . 17
- 5.6 Randomizer Test (`tb_generate_tetromino`) . . . 17
- 5.7 Hardware Validation . . . 17

6 References **18**

7 Reflection **18**

A Appendix: Verilog Code **20**
- A.1 Top Level Module . . . 20
- A.2 Global Definitions . . . 29
- A.3 Display Modules . . . 31

A.4 Game Logic Modules 56
A.5 Input Processing Modules 82
A.6 Testbenches and Simulation Logs 89

1 Introduction

1.1 What is Tetris?

Tetris is a tile-matching puzzle game originally designed by Alexey Pajitnov in 1985. The player manipulates falling geometric shapes called *tetrominoes* rotating and positioning them to complete horizontal lines on a 10×20 grid. Completed lines are cleared, and the game ends when pieces stack to the top of the playfield.

1.2 Why Implement Tetris on an FPGA?

Implementing Tetris on an FPGA presents unique challenges that make it an ideal educational project for digital design:

- **Real-Time Constraints**: The game requires generating VGA video signals at precise timing intervals (83.46 MHz pixel clock) while simultaneously processing user input and updating game state.
- **Parallel Processing**: Unlike software implementations that execute sequentially, hardware must handle input sampling, game logic, and pixel rendering concurrently across different clock domains.
- **State Machine Complexity**: The game logic involves a multi-state FSM handling piece spawning, movement, rotation with wall kicks, line clearing, and scoring.
- **Protocol Implementation**: Direct interfacing with VGA displays and PS/2 keyboards requires understanding and implementing timing-critical communication protocols.
- **Advanced Game Mechanics**: Modern Tetris features like the Super Rotation System (SRS) with wall kicks, Delayed Auto Shift (DAS) for smooth movement, ghost piece projection, and the 7-bag randomizer all require dedicated hardware logic.

1.3 Project Overview

The objective of this class project was to implement a fully functional Tetris game on an FPGA verification board. The project utilizes SystemVerilog to describe the hardware logic required for generating VGA video signals, processing user input via a PS/2 keyboard, and maintaining the complex game state.

The system is designed to interface with standard peripherals: a VGA monitor for display and a keyboard for control. The implementation demonstrates key digital design concepts such as finite state machines (FSMs), clock domain crossing (CDC), protocol handling (PS/2), video timing generation, and pipelined rendering.

2 VGA Signaling and Timing Generation

VGA (Video Graphics Array) is an analog video transmission standard that requires precise timing signals to control the electron beam in a CRT monitor (or the pixel scanning in modern LCDs).

2.1 Protocol Overview

The video signal consists of three color channels (Red, Green, Blue) and two synchronization pulses:

1. **Horizontal Sync (HS)**: Signals the end of a line and controls the horizontal retrace.

2. **Vertical Sync (VS)**: Signals the end of a frame and controls the vertical retrace.

To achieve the target resolution of 1280×800 at 60Hz, our system operates with a pixel clock of approximately 83.46 MHz [1]. The visible pixels are output only during the *active area*, with blanking intervals for sync pulses and porch regions.

In traditional CRT displays, a single cathode ray was deflected across the screen to produce individual lit points on the phosphor, scanning from top-left across to complete the first row, followed by starting from the left again to produce the second row, and so on. Since it takes time for the beam to move from right to left (horizontal retrace) and from bottom to top (vertical retrace), there are extra cycles where no visible pixels are produced, these are the blanking intervals.

2.2 Circuit Implementation

We implemented the timing logic in the `vga_out` module. Two counters, `hcount` and `vcount`, track the scanning beam's position. The horizontal counter `hcount` increments on every pixel clock, resetting after reaching `H_MAX` (1679). The vertical counter `vcount` increments whenever `hcount` completes a line.

The synchronization pulses are generated by comparing these counters against specific thresholds:

```
// README SPEC: hsync 0 when hcount 0-135 (Active Low)
assign hsync = ~(hcount <= H_SYNC_END); // H_SYNC_END = 135

// README SPEC: vsync 1 when vcount 0-2 (Active High)
assign vsync = (vcount <= V_SYNC_END);  // V_SYNC_END = 2
```

Listing 1: VGA Sync Logic in vga_out.sv

2.3 Determining Pixel Coordinates

To draw objects, we must know the coordinate of the pixel currently being rendered. The raw counters include "blanking" intervals (front/back porch) where no video is sent. We calculate `curr_x` and `curr_y` relative to the *visible* area only.

The `active_area` signal is high only when the counters are within the visible window.

```
// Active Area Definition
assign active_area = (hcount >= H_VIS_START && hcount <= H_VIS_END) &&
                     (vcount >= V_VIS_START && vcount <= V_VIS_END);

// Coordinate Calculation (Relative to Top-Left of Screen)
// Output coordinates relative to active area
always_comb begin
    if (active_area) begin
        curr_x = hcount - H_VIS_START; // Offset by 336
        curr_y = vcount - V_VIS_START; // Offset by 27
```

```
    end else begin
        curr_x = 0; curr_y = 0;
    end
end
```

Listing 2: Active Area and Coordinate Calculation

As visualized in Figure 1, the synthesized schematic confirms this logic. The large counter blocks (seen as accumulators in the center) correspond to `hcount` and `vcount`, while the comparators on the right generate the sync pulses based on the parameters defined in `GLOBAL.sv`. This hardware implementation ensures cycle-accurate timing compliant with the VESA standard.

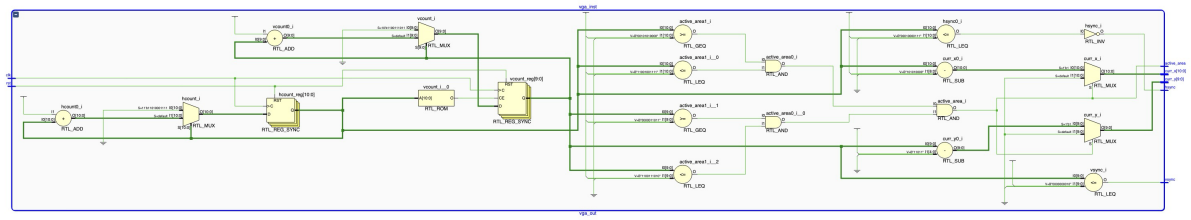


Figure 1: Synthesized Schematic of the ‘vga_out‘ module showing counters and sync comparators.

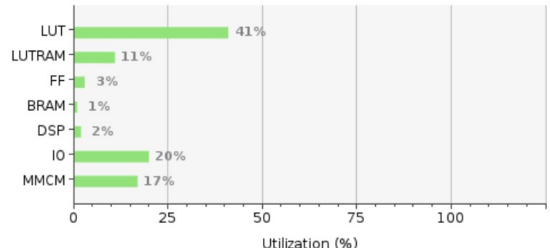
3 Module Architecture

The system serves as a complex hierarchy of SystemVerilog modules. Below is a definition of the key components and their roles in the design:

Module	Description
`game_top`	The top-level wrapper. It generates all system clocks (using MMCM or dividers), handles the global reset, and instantiates the IO drivers (VGA, PS/2).
`vga_out`	The video timing generator. It produces the `hsync`, `vsync`, and `active_area` signals required for the 1280×800 display standard.
`game_control`	The core logic hub. It contains the main FSM that governs game flow (spawning, gravity, line clearing) and maintains the board state array. The board is implemented as a 10×22 grid, with 2 invisible buffer rows at the top for piece spawning, following standard Tetris convention. Only rows 2-21 are visible to the player.
`draw_tetris`	The rendering engine. It takes the game state and current pixel coordinate to output an RGB color, leveraging sprites and using a multi-stage pipeline.
`input_manager`	The input processor. It implements Delayed Auto Shift (DAS) to make controls feel responsive, converting raw key presses into game commands.
`check_valid`	A combinational logic block that checks if a proposed piece position collides with walls or existing blocks.
`ghost_calc`	A helper module that calculates the "shadow" position of the piece by projecting it downwards until a collision occurs.
`GLOBAL.sv`	A shared header file containing game-wide constants (grid dimensions, timing parameters, tetromino definitions) used by all modules to ensure consistency.

3.1 Resource Utilization

The design was synthesized and implemented for the Artix-7 FPGA Figure 2 shows the resource utilization breakdown.



(a) Utilization Bar Chart

Resource	Utilization	Avail.	Util. %
LUT	25,787	63,400	40.67%
LUTRAM	2,004	19,000	10.55%
FF	3,941	126,800	3.11%
BRAM	0.5	135	0.37%
DSP	5	240	2.08%
IO	41	210	19.52%
MMCM	1	6	16.67%

(b) Utilization Table

Figure 2: FPGA Resource Utilization Summary.

The design uses approximately 40% of available LUTs, primarily for the game logic FSM and collision detection. The low BRAM usage (0.5 blocks) reflects the efficient use of distributed RAM for the game board. A single MMCM generates the multiple clock domains required.

4 Design Description and Drawing Logic

4.1 Drawing Strategy: Pipelined Renderer

Drawing Objects on the screen is performed by the `draw_tetris` module. Unlike software rendering which clears a buffer and redraws, hardware rendering must decide the color of *the specific pixel being clocked out right now.*

To handle the complex decision making within a single pixel clock cycle (<12ns), the design uses a pipelined approach shown in Figure 9.

1. **Stage 1: Region Detection (Fig 9a).** Logic comparators classify the current `curr_x/y` pixel into regions: Board, Next Piece, Score, or Border. This reduces the problem space for the next stages.

2. **Stage 2: Data Access.** Based on the region, the system fetches data from the game board RAM or character ROM.

3. **Stage 3: Sprite Output (Fig 9b).** The retrieved block ID is used to address the Sprite ROM. The schematic shows the multiplexing logic that selects the final pixel color based on the sprite data and piece type (Cyan, Purple, etc.).

4.2 4-Stage Pipeline Details

The logic checks the grid coordinates derived from `curr_x/y`. If the pixel corresponds to a non-empty cell in the `display.data` array, we assign a color index.

```
// Calculate Grid Indices
s1_grid_col <= (curr_x - GRID_X_START) / BLOCK_SIZE;
s1_grid_row <= (curr_y - GRID_Y_START) / BLOCK_SIZE;

// Check if block exists
// The +2 offset maps visible screen coordinates to the internal
// board array, which includes 2 hidden spawn rows at the top.
if (display.data[s1_grid_row + 2][s1_grid_col].data != `TETROMINO_EMPTY
    ) begin
     s2_cell_color_idx <= display.data[s1_grid_row + 2][s1_grid_col].
    data + 1;
end
```

Listing 3: Grid Drawing Logic (draw_tetris.sv)

This logic ensures that as the beam scans across the screen, it "picks up" the color of the Tetris block at that location.

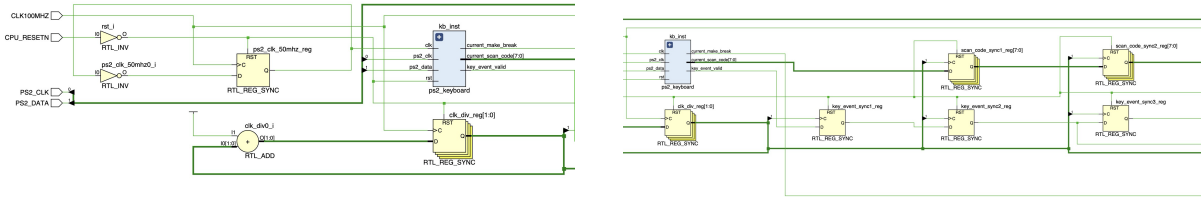
4.3 Clock Architecture

The system utilizes a complex clocking scheme to balance performance and protocol requirements. Five distinct clock domains are generated:

- **100 MHz**: Main system clock and 7-segment display driver.
- **83.46 MHz**: Pixel clock for 1280x800 VGA timing. This frequency is derived from the VESA Display Monitor Timing Standard[1] which specifies 83.46 MHz as the required pixel clock for 1280×800 @ 60Hz resolution.

- **50 MHz**: PS/2 clock for reliable keyboard sampling (2x oversampling of max PS/2 frequency). The PS/2 protocol typically operates at 10-16.7 kHz, so 50 MHz provides sufficient oversampling margin for robust signal capture.
- **25 MHz**: Game Logic clock. The FSM runs at this lower frequency to allow simple single-cycle logic without timing violations relative to the fast pixel clock, while still being fast enough for responsive gameplay.
- **60 Hz**: Game Tick. Generated from the 25 MHz clock to drive gravity and animation updates, matching the display's refresh rate for smooth visual feedback.

Figure 3 shows the synthesized clock generation circuit and the CDC synchronizers used to safely transfer signals between domains.



(a) Clock Generation Circuit (b) CDC Synchronizers

Figure 3: Clock Architecture and Domain Crossing Logic.

4.4 Input Processing

The system supports two input methods: a PS/2 keyboard and on-board pushbuttons. As shown in Figure 4, OR gates combine these sources before passing them to the input manager. This allows seamless switching between input methods without software configuration.

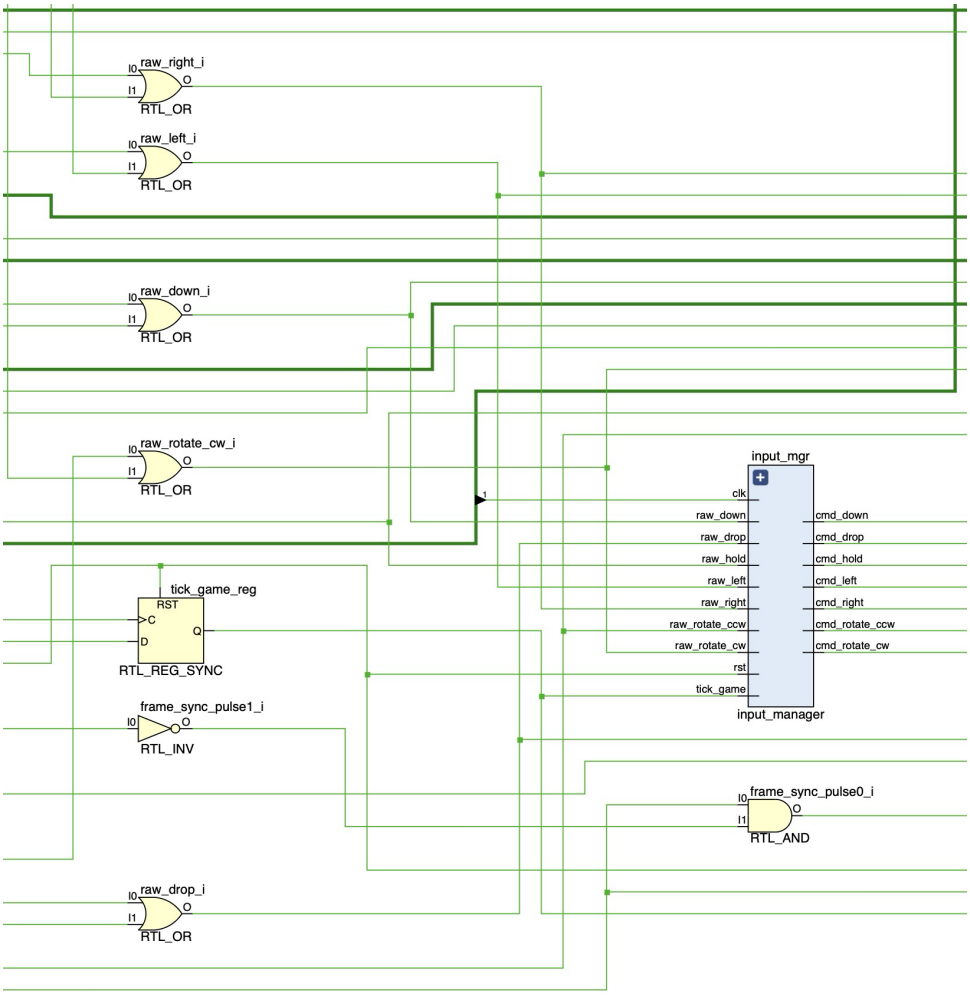
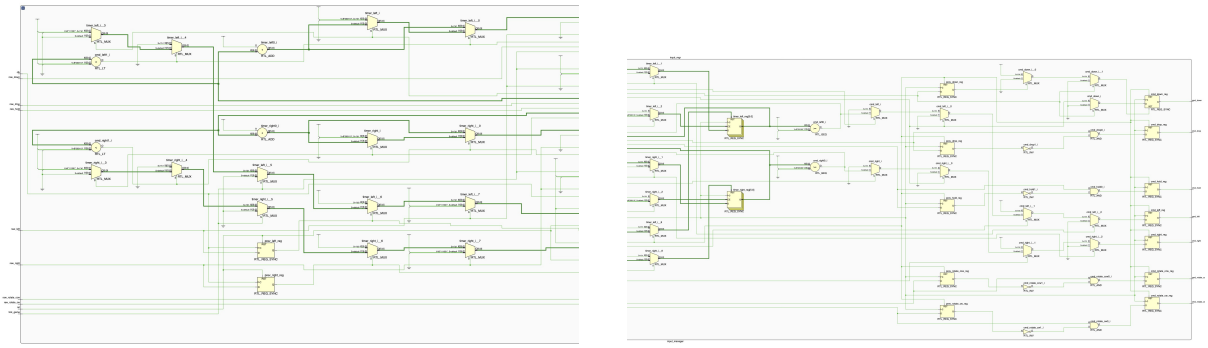


Figure 4: Input merging logic showing OR gates combining keyboard and physical button inputs.

The input manager then ensures that player controls are fluid. As shown in Figure 5, the implementation relies on dedicated counters for the "Delayed Auto Shift" (DAS) mechanism. The counters (‘timer_left‘ and ‘timer_right‘ visible in the schematic) increment whilst a key is held. Once they cross a threshold (DAS_DELAY), the system generates rapid movement pulses. This hardware-based timing ensures that the repeat rate is independent of the frame rate or game logic load.



(a) Left DAS Counter

(b) Right DAS Counter

Figure 5: Input Manager DAS Logic for Left/Right Movement.

4.4.1 PS/2 Keyboard Protocol

The project supports full keyboard input via the PS/2 protocol. The `PS2Receiver` module implements the low-level deserialization, sampling the PS/2 clock and data lines to extract 11-bit frames (1 start, 8 data, 1 parity, 1 stop). These raw scan codes are then decoded by `ps2_keyboard`, which handles:

- **Make/Break Detection**: A `0xF0` prefix indicates a key release.
- **Extended Codes**: A `0xE0` prefix is used for arrow keys and special keys.

The decoded key events are synchronized to the game clock domain using a 3-stage synchronizer chain to avoid metastability.

4.5 Game Logic

The core logic resides in ‘game_control‘. It works in tandem with specialized sub-modules:

4.5.1 Randomizer (7-Bag System)

To ensure fair gameplay, we implemented a **7-Bag Randomizer**[4] in ‘generate_tetromino.sv‘. Instead of pure random selection, the system generates a ”bag” of all 7 pieces and draws from it until empty, then regenerates. This guarantees that a player will never go more than 13 moves without seeing a specific piece (e.g., the ’I’ bar).

4.5.2 Rotation System (SRS)

The project implements the **Super Rotation System (SRS)**[2] standards as defined in the Tetris Guideline[3]. The ‘rotate_tetromino‘ module handles:

- Basic 90-degree matrix rotation.
- **Wall Kicks**: If a rotation would cause a collision, the system tries alternative offsets (up, left, right) derived from standard look-up tables. This allows players to rotate pieces even in tight spaces.

4.5.3 Scoring and Level Progression

The score is calculated using classic NES Tetris rules, scaled by the current level:

- Single: 40 × (Level + 1)
- Double: 100 × (Level + 1)
- Triple: 300 × (Level + 1)
- Tetris (4 lines): 1200 × (Level + 1)

The level increases every 10 lines cleared, which in turn speeds up the gravity (drop timer decreases). This is implemented in `game_control.sv` using a lookup table for `drop_speed_frames`.

4.5.4 T-Spin Detection

Modern Tetris rewards advanced techniques like the T-Spin, where the T-piece rotates into a tight spot. The `spin_detector` module checks if the piece is "immobile" (3 of 4 corners occupied) immediately after a rotation. If true and lines are cleared, bonus points are awarded. Figure 6 illustrates a typical T-Spin scenario.

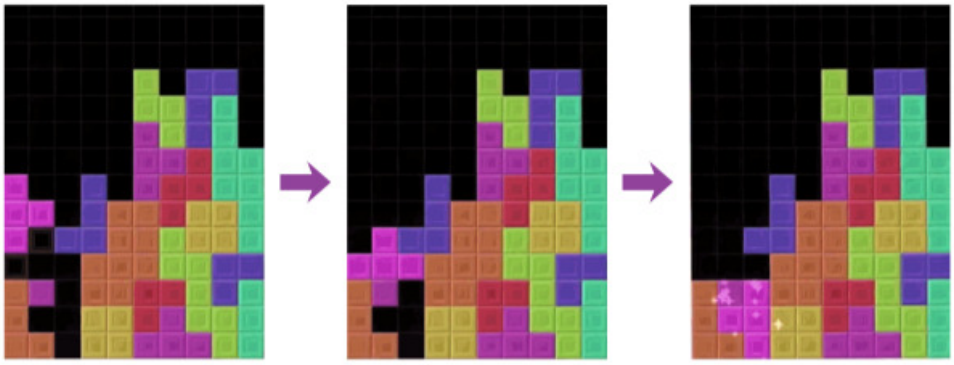


Figure 6: Example of a T-Spin: The T-piece rotates into a slot where it is immobile.

4.5.5 Finite State Machine (FSM)

The complex behavior of the game is governed by a central FSM in `game_control.sv`. Figure 7 shows the state transitions between the 15 states, including the main loops for input handling ('IDLE'), piece manipulation ('MOVE'/'ROTATE'), and gravity ('DOWN'). The key states are summarized in Table 1.

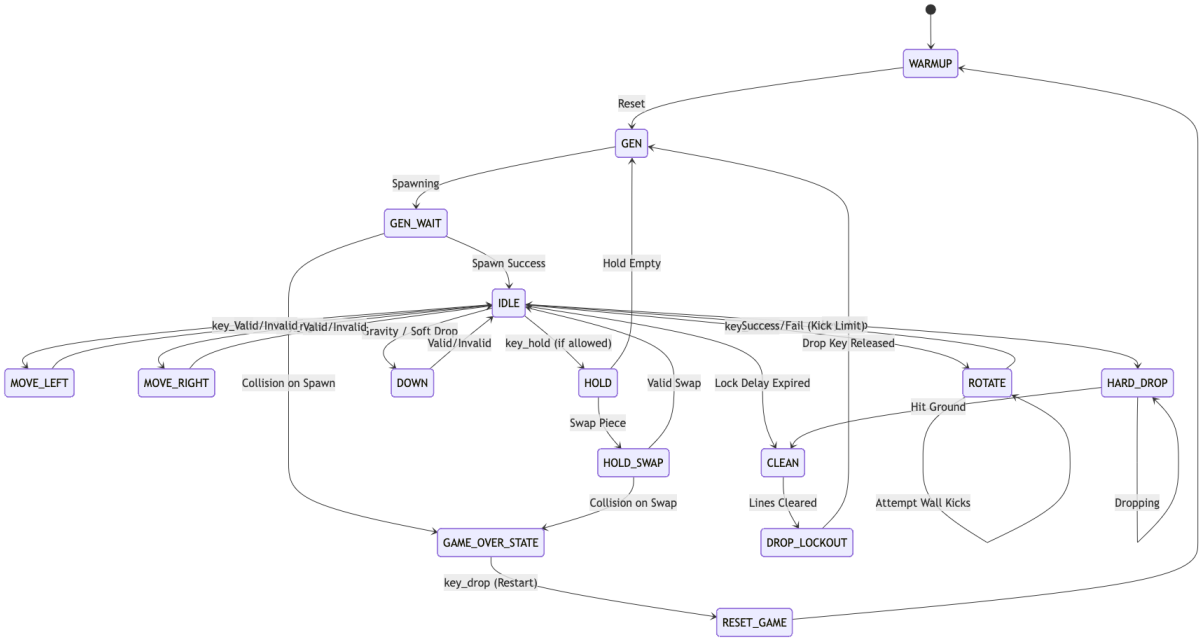


Figure 7: Game Control FSM Diagram showing major state transitions.

State	Description
`GEN`	Spawns a new tetromino from the randomizer.
`IDLE`	Waits for user input (Left, Right, Rotate) or gravity timer.
`MOVE_*`	Updates x-coordinate. Checks collision; reverts if invalid.
`ROTATE`	Tries standard rotation. If invalid, attempts Wall Kicks (SRS).
`DOWN`	Moves piece down. If blocked, locks piece and goes to CLEAN.
`CLEAN`	Scans for full rows, removes them, and shifts grid down.
`HOLD`	Swaps current piece with held piece (if allowed).
`GAME_OVER`	Halts game until reset command is received.

Table 1: Description of Main FSM States in `game_control`.

4.5.6 Ghost Piece

The ‘ghost_calc‘ module continuously calculates where the current piece would land if dropped instantly. This ”Ghost Piece” is rendered semi-transparently to aid player accuracy. Figure 8 shows the hardware logic for the drop and rotate operations.

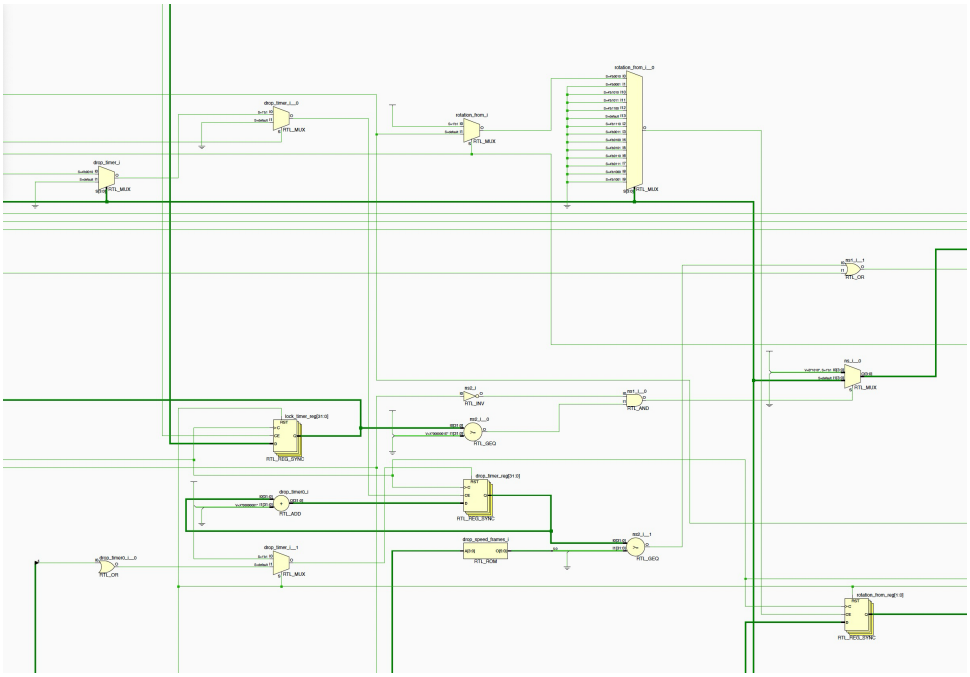


Figure 8: Logic for piece manipulation (Drop and Rotate).

4.6 Display Logic

The display logic is decoupled from the game state update rate.

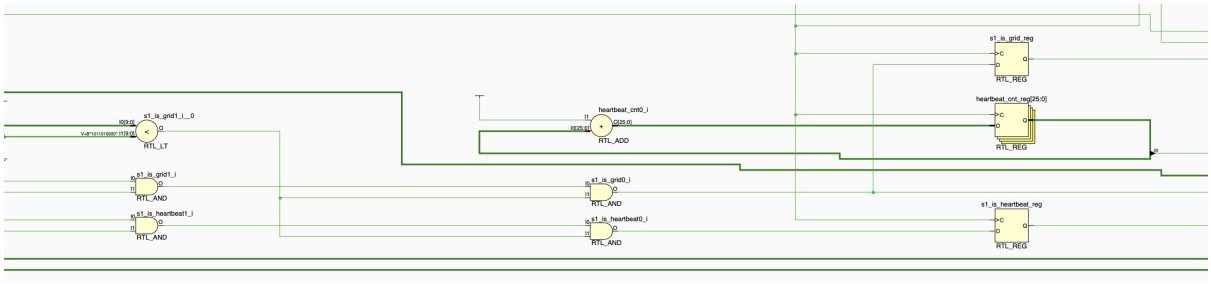
- **CDC for Display**: The game state (field, current piece position) is transferred from the game domain to the pixel domain at the start of each frame (VSync) to ensure a tear-free image.
- **Rendering**: ‘draw_tetris‘ calculates the color of the current pixel.
- **Sprites**: A ROM-based sprite system (‘block_sprite‘) adds a beveled 3D look to the blocks. The sprites are stored as grayscale and tinted dynamically based on the piece type (e.g., Cyan for ’I’, Purple for ’T’).

4.6.1 Sprite System Details

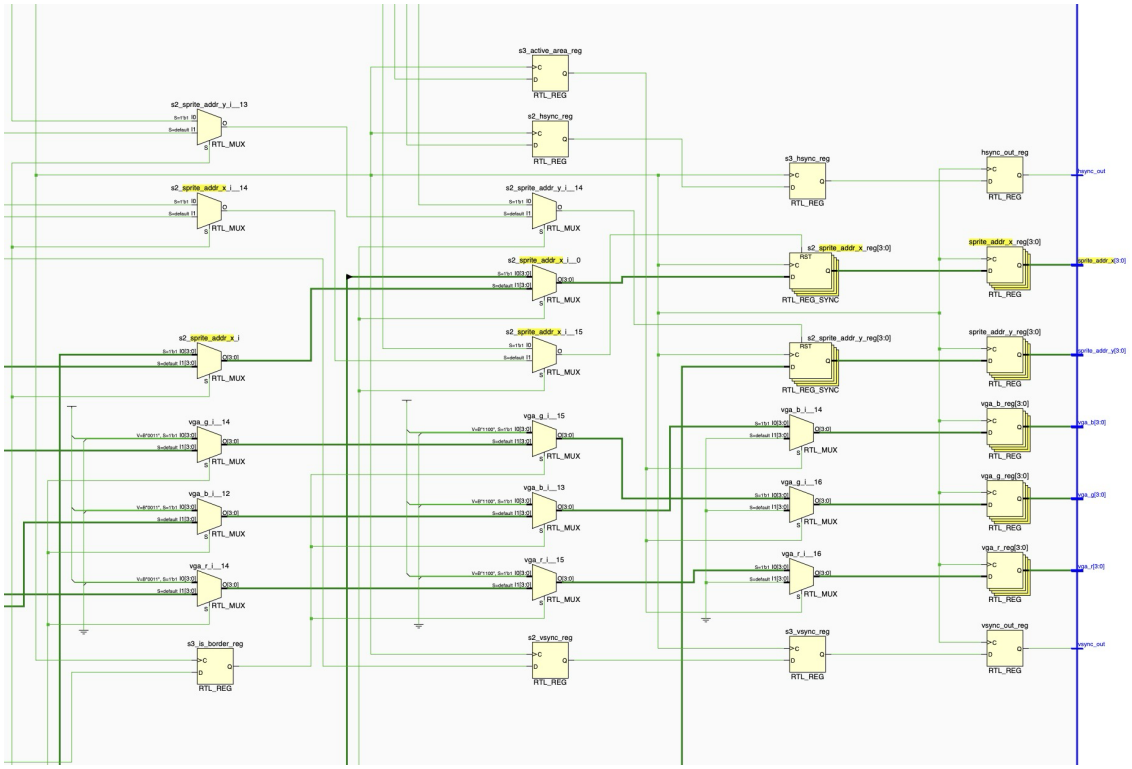
The `block_sprite` module stores a 16x16 pixel grayscale texture in ROM. Each pixel is a 12-bit value representing a brightness level. During rendering:

1. The current pixel’s position within a 32x32 block is calculated (using `curr_x % 32`).
2. This address is sent to the sprite ROM, which returns a brightness value.
3. The brightness is multiplied by a per-piece color (e.g., Cyan for I, Purple for T) to produce the final RGB output.

This allows a single sprite texture to be reused for all 7 piece types, saving ROM space.



(a) Region Detection Logic (Stage 1)



(b) Sprite Output Logic (Stage 3)

Figure 9: Rendering Pipeline Implementation: Region detection determines the active object, while the sprite pipeline fetches pixel data.

4.7 Game Logic Integration

The game state is managed by the `game_control` module. It handles:

- **Gravity**: A counter generates a "tick" (approx 60Hz) to move pieces down.
- **Collisions**: The `check_valid` module predicts next positions to prevent overlapping.
- **Ghost Piece**: We calculate where the piece would land if dropped instantly and render it semi-transparently.

5 Testing and Verification

We developed a comprehensive suite of testbenches to verify individual modules before integration. All simulations were run in Vivado XSim.

5.1 Testbench Summary

Table 2 lists the key testbenches and their verification targets.

Testbench	Module Under Test	Result
`tb_game_control`	Core FSM (IDLE, MOVE, CLEAN)	9/9 PASS
`tb_input_manager`	DAS, one-shot logic	17/17 PASS
`tb_rotate_tetromino`	CW/CCW rotation matrix	4/4 PASS
`tb_hold_feature`	Hold swap, lockout	9/9 PASS
`tb_generate_tetromino`	7-Bag randomizer	13/13 PASS

Table 2: Summary of Testbenches and Results.

5.2 Game Control Simulation (`tb_game_control`)

The core FSM was verified by simulating key game scenarios: piece spawning, movement, rotation, hold functionality, and game over detection. Figure 10 shows a representative simulation waveform from Vivado XSim.

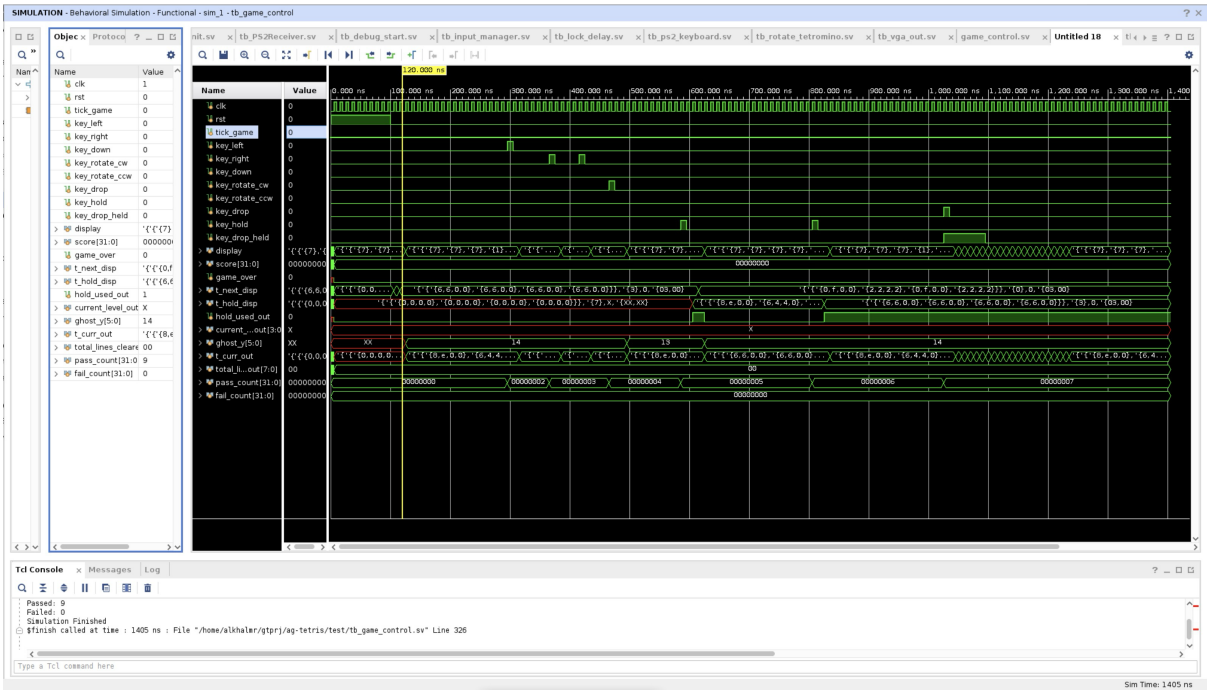


Figure 10: Simulation waveform from `tb_game_control` showing FSM state transitions, piece position updates, and control signals.

5.3 Input Manager Test (`tb_input_manager`)

This testbench verifies **DAS** (Delayed Auto Shift) and **one-shot** behavior. The log confirms that the rotation commands only fire once per key press, and the DAS repeat only triggers after a delay.

```
Test 1: Rotate CW One-Shot
  PASS: Rotate CW Triggered on press
```

```
  PASS: Rotate CW Pulse Ended after one cycle
  PASS: Rotate CW did not re-trigger while holding

Test 4: Left DAS (Delayed Auto Shift)
  PASS: Left Initial Move on press
  PASS: No trigger during DAS delay (15 frames)
  PASS: Left DAS Auto-Repeat Triggered
```

5.4 Rotation Test (tb_rotate_tetromino)

This testbench checks the basic matrix rotation (CW and CCW), ensuring the rotation index wraps correctly (e.g., $3 \rightarrow 0$ for CW).

```
Test 1: CW 0 -> 1
PASS: CW rotation 0 -> 1
Test 2: CW wrap 3 -> 0
PASS: CW rotation 3 -> 0
Test 3: CCW 0 -> 3
PASS: CCW rotation 0 -> 3
```

5.5 Hold Feature Test (tb_hold_feature)

This tests the swap-and-lockout mechanic. It verifies that after one hold, the player cannot hold again until a new piece has been placed.

```
Test 2: First Hold (Empty -> Store)
  Current piece before hold: J (1)
  After hold: curr=0, hold=J, hold_used=1
  PASS: First piece stored in hold

Test 3: Hold Lockout (Cannot Hold Twice Per Piece)
  Attempting second hold while hold_used=1...
  PASS: Lockout prevented second hold
```

5.6 Randomizer Test (tb_generate_tetromino)

Verified that the 7-Bag randomizer produces valid piece indices on sequential requests.

```
=== Test 2: Sequential Generation ===
PASS: Piece 0 - Valid index: 2
PASS: Piece 1 - Valid index: 2
...
PASS: Piece 9 - Valid index: 3
```

5.7 Hardware Validation

On the physical FPGA board (Nexys A7), we verified:

- **Visual Output**: The Tetris grid is centered, block colors match type (Cyan I, Purple T), and the ghost piece renders correctly.

- **Input Response**: The DAS feels responsive, similar to official Tetris games.
- **Gameplay**: Full games were played through including line clears, reaching max level, game over, and reset functionality.

6 References

References

[1] "VESA Signal 1280 x 800 @ 60 Hz timing," *tinyvga.com*. [Online]. Available: http://www.tinyvga.com/vga-timing/1280x800@60Hz

[2] "Super Rotation System," *TetrisWiki*. [Online]. Available: https://tetris.wiki/Super_Rotation_System

[3] "Tetris Guideline," *Hard Drop Tetris Wiki*. [Online]. Available: https://harddrop.com/wiki/Tetris_Guideline

[4] "Random Generator," *TetrisWiki*. [Online]. Available: https://tetris.wiki/Random_Generator

7 Reflection

This project provided invaluable hands-on experience in designing hardware systems from first principles to meet complex application requirements. The most significant technical skill gained was learning to architect hardware pipelines and finite state machines that operate across multiple clock domains, a challenge rarely encountered in pure software development but fundamental to digital design.

The video timing generation was particularly rewarding, as it offered concrete insight into how digital signals translate directly to visual information on a display. Implementing the 4-stage rendering pipeline and ensuring pixel-perfect synchronization with the VGA protocol deepened my understanding of hardware parallelism and real-time constraints.

Beyond clock domain crossing, the project presented substantial challenges in implementing smooth, responsive controls and the complex Super Rotation System with wall kicks and T-spin detection. Achieving fluid input handling required careful design of the Delayed Auto Shift (DAS) mechanism with dedicated hardware counters, while the SRS rotation system demanded implementing lookup tables and collision checking logic that could execute within tight timing constraints. Balancing game logic complexity with the need for deterministic, cycle-accurate behavior across multiple concurrent processes proved to be one of the most demanding aspects of the design.

If I were to approach this project again, I would invest more time upfront in formal verification of the FSM state transitions and create more comprehensive waveform-based testbenches for the rendering pipeline. While simulation testing caught most bugs, some visual artifacts only appeared during hardware deployment—earlier detection through more rigorous simulation could have saved debugging time on the physical FPGA.

Future enhancements to this implementation could include audio output using PWM for sound effects and background music, animated score displays with particle effects

during line clears, and support for two-player competitive modes. Additionally, implementing variable DAS settings and customizable key mappings would improve the user experience for players with different skill levels and preferences.

A Appendix: Verilog Code

This appendix contains the complete SystemVerilog source code for the AG Tetris project. Each file is prefaced with a brief description of its purpose.

A.1 Top Level Module

game_top.sv

The main system integration module. It generates all required clock domains (83.46 MHz pixel, 50 MHz PS/2, 25 MHz game), instantiates all subsystems (VGA, input, game logic), and handles clock domain crossing between them.

```
`include "src/GLOBAL.sv"

module game_top(
    input  wire logic CLK100MHZ,
    input  wire logic CPU_RESETN, // Active Low
    input  wire logic PS2_CLK,
    input  wire logic PS2_DATA,
    input  wire logic btn_l,
    input  wire logic btn_r,
    input  wire logic btn_u,
    input  wire logic btn_d,
    input  wire logic btn_c,
    output logic [3:0] VGA_R,
    output logic [3:0] VGA_G,
    output logic [3:0] VGA_B,
    output logic VGA_HS,
    output logic VGA_VS,
    output logic [1:0] LED,      // Debug LEDs
    // 7-Segment Display
    output logic [6:0] SEG,      // Segment pattern (active low)
    output logic [7:0] AN,       // Anode select (active low)
    output logic       DP        // Decimal point (active low)
    );

    logic rst;
    assign rst = ~CPU_RESETN;

    // Debug LEDs - PS/2 Hardware Signals
    // NOTE: PS/2 uses pull-ups, so lines are HIGH when idle/
   disconnected
    // With keyboard active (typing), you'll see flickering/blinking
    // LED[0] = PS2_CLK (will flicker when keyboard sends data)
    // LED[1] = PS2_DATA (will change based on transmitted bits)
    assign LED[0] = PS2_CLK;
    assign LED[1] = PS2_DATA;

    // Clock Generation
    logic pix_clk; // 83.46 MHz (approx)
    logic locked;

    // Instantiate Clock Wizard
    clk_wiz_0 clk_gen (
        .clk_in1(CLK100MHZ),
        .clk_out1(pix_clk),
```

```
        .reset(rst),
        .locked(locked)
    );

    // PS2 Clock Generation (50 MHz) - Better for PS2 protocol timing
    // Divide 100MHz by 2
    logic ps2_clk_50mhz;
    always_ff @(posedge CLK100MHZ) begin
        if (rst) ps2_clk_50mhz <= 0;
        else ps2_clk_50mhz <= ~ps2_clk_50mhz;
    end

    // Game Clock Generation (25 MHz)
    // Divide 100MHz by 4
    logic [1:0] clk_div;
    logic game_clk;

    always_ff @(posedge CLK100MHZ) begin
        if (rst) clk_div <= 0;
        else clk_div <= clk_div + 1;
    end
    assign game_clk = clk_div[1]; // 25 MHz

    // Game Tick Generation (60Hz)
    // 25 MHz / 60 Hz ~= 416,666
    logic [18:0] tick_counter;
    logic tick_game;

    always_ff @(posedge game_clk) begin
        if (rst) begin
            tick_counter <= 0;
            tick_game <= 0;
        end else begin
            if (tick_counter == 416666) begin
                tick_counter <= 0;
                tick_game <= 1;
            end else begin
                tick_counter <= tick_counter + 1;
                tick_game <= 0;
            end
        end
    end

    //
    ============================================================================

    // PS2 Keyboard Input (50MHz domain)
    //
    ============================================================================

    logic [7:0] scan_code_50;
    logic make_break_50;
    logic key_event_valid_50;

    ps2_keyboard kb_inst (
        .clk(ps2_clk_50mhz),
        .rst(rst),
        .ps2_clk(PS2_CLK),
```

```
        .ps2_data(PS2_DATA),
        .current_scan_code(scan_code_50),
        .current_make_break(make_break_50),
        .key_event_valid(key_event_valid_50)
    );

    //
    ============================================================================

    // CDC: Clock Domain Crossing from 50MHz to 25MHz (game_clk)
    //
    ============================================================================

    // Synchronize key_event_valid pulse to game_clk domain
    // Note: key_event_valid_50 is now extended to 4 cycles (80ns) to
    ensure capture
    logic key_event_sync1, key_event_sync2, key_event_sync3;
    logic key_event_pulse;

    always_ff @(posedge game_clk) begin
        if (rst) begin
            key_event_sync1 <= 0;
            key_event_sync2 <= 0;
            key_event_sync3 <= 0;
        end else begin
            key_event_sync1 <= key_event_valid_50;
            key_event_sync2 <= key_event_sync1;
            key_event_sync3 <= key_event_sync2;
        end
    end

    // Detect rising edge in game_clk domain
    assign key_event_pulse = key_event_sync2 & ~key_event_sync3;

    // Synchronize scan_code and make_break (they're stable during
    extended pulse)
    logic [7:0] scan_code_sync1, scan_code_sync2;
    logic make_break_sync1, make_break_sync2;

    always_ff @(posedge game_clk) begin
        if (rst) begin
            scan_code_sync1 <= 8'h00;
            scan_code_sync2 <= 8'h00;
            make_break_sync1 <= 1'b0;
            make_break_sync2 <= 1'b0;
        end else begin
            scan_code_sync1 <= scan_code_50;
            scan_code_sync2 <= scan_code_sync1;
            make_break_sync1 <= make_break_50;
            make_break_sync2 <= make_break_sync1;
        end
    end

    // For 7-segment display use
    logic [7:0] scan_code;
    logic make_break;
    assign scan_code = scan_code_sync2;
    assign make_break = make_break_sync2;
```

```
150
151     //
    ============================================================================

152     // Decode Raw Levels (Held State) - Now in game_clk domain
153     // WITH WATCHDOG INTEGRATED (single driver!)
154     //
    ============================================================================

155     logic raw_left_kb, raw_right_kb, raw_down_kb, raw_rotate_cw_kb,
    raw_rotate_ccw_kb, raw_drop_kb, raw_hold_kb;
156
157     // Watchdog counter (declared here, used in same block)
158     logic [19:0] watchdog_counter;
159     logic watchdog_timeout;
160
161     always_ff @(posedge game_clk) begin
162         if (rst) begin
163             raw_left_kb <= 0; raw_right_kb <= 0; raw_down_kb <= 0;
164             raw_rotate_cw_kb <= 0; raw_rotate_ccw_kb <= 0; raw_drop_kb
    <= 0; raw_hold_kb <= 0;
165             watchdog_counter <= 0;
166             watchdog_timeout <= 0;
167         end else begin
168             // === WATCHDOG LOGIC ===
169             if (raw_left_kb || raw_right_kb || raw_down_kb ||
170                 raw_rotate_cw_kb || raw_rotate_ccw_kb || raw_drop_kb ||
     raw_hold_kb) begin
171                 if (watchdog_counter < 20'd750000) begin
172                     watchdog_counter <= watchdog_counter + 1;
173                     watchdog_timeout <= 0;
174                 end else begin
175                     watchdog_timeout <= 1;
176                 end
177             end else begin
178                 watchdog_counter <= 0;
179                 watchdog_timeout <= 0;
180             end
181
182             // === KEYBOARD DECODE (only driver of raw_*_kb signals)
    ===
183             if (watchdog_timeout) begin
184                 // Watchdog fired - clear all keys
185                 raw_left_kb <= 0;
186                 raw_right_kb <= 0;
187                 raw_down_kb <= 0;
188                 raw_rotate_cw_kb <= 0;
189                 raw_rotate_ccw_kb <= 0;
190                 raw_drop_kb <= 0;
191                 raw_hold_kb <= 0;
192             end else if (key_event_pulse) begin
193                 // Normal keyboard events
194                 case (scan_code_sync2)
195                     `LEFT_ARROW_C:  raw_left_kb   <= make_break_sync2;
196                     `RIGHT_ARROW_C: raw_right_kb  <= make_break_sync2;
197                     `DOWN_ARROW_C:  raw_down_kb   <= make_break_sync2;
198                     `UP_ARROW_C:    raw_rotate_cw_kb <=
    make_break_sync2;
```

```
                    `X_KEY_C:       raw_rotate_cw_kb <=
make_break_sync2;
                    `Z_KEY_C:       raw_rotate_ccw_kb <=
make_break_sync2;
                    `SPACE_C:       raw_drop_kb   <= make_break_sync2;
                    `LSHIFT_C:      raw_hold_kb   <= make_break_sync2;
                    default: ; // No change for other keys
                endcase
            end
        end
    end

    // Debounce Buttons (use separate debouncer for buttons, different
timing)
    logic btn_l_db, btn_r_db, btn_u_db, btn_d_db, btn_c_db;
    logic unused_db;

    // Button debouncer with longer timing for mechanical buttons
    debouncer_btn db_lr (
        .clk(game_clk),
        .I0(btn_l), .I1(btn_r),
        .O0(btn_l_db), .O1(btn_r_db)
    );

    debouncer_btn db_ud (
        .clk(game_clk),
        .I0(btn_u), .I1(btn_d),
        .O0(btn_u_db), .O1(btn_d_db)
    );

    debouncer_btn db_c (
        .clk(game_clk),
        .I0(btn_c), .I1(1'b0),
        .O0(btn_c_db), .O1(unused_db)
    );

    // Combine with Buttons (Active High)
    logic raw_left, raw_right, raw_down, raw_rotate_cw, raw_rotate_ccw,
    raw_drop, raw_hold;
    assign raw_left   = raw_left_kb   | btn_l_db;
    assign raw_right  = raw_right_kb  | btn_r_db;
    assign raw_down   = raw_down_kb   | btn_d_db;
    assign raw_rotate_cw = raw_rotate_cw_kb | btn_u_db;
    assign raw_rotate_ccw = raw_rotate_ccw_kb;
    assign raw_drop   = raw_drop_kb   | btn_c_db;
    assign raw_hold   = raw_hold_kb;  // Hold only via keyboard (no
button)

    // Input Manager (DAS & One-Shot)
    logic key_left, key_right, key_down, key_rotate_cw, key_rotate_ccw,
    key_drop, key_hold;

    input_manager input_mgr (
        .clk(game_clk),
        .rst(rst),
        .tick_game(tick_game),
        .raw_left(raw_left),
        .raw_right(raw_right),
```

```
        .raw_down(raw_down),
        .raw_rotate_cw(raw_rotate_cw),
        .raw_rotate_ccw(raw_rotate_ccw),
        .raw_drop(raw_drop),
        .raw_hold(raw_hold),
        .cmd_left(key_left),
        .cmd_right(key_right),
        .cmd_down(key_down),
        .cmd_rotate_cw(key_rotate_cw),
        .cmd_rotate_ccw(key_rotate_ccw),
        .cmd_drop(key_drop),
        .cmd_hold(key_hold)
    );

    // Game Logic
    field_t display_field;
    logic [31:0] score;
    logic game_over;
    tetromino_ctrl t_next; // Next piece signal
    tetromino_ctrl t_hold; // Hold piece signal
    logic hold_used;       // Hold was used this piece
    logic [3:0] current_level;
    logic signed [`FIELD_VERTICAL_WIDTH : 0] ghost_y;
    tetromino_ctrl t_curr;
    logic [7:0] total_lines_cleared; // NEW: For level bar

    game_control game_inst (
        .clk(game_clk),
        .rst(rst),
        .tick_game(tick_game),
        .key_left(key_left),
        .key_right(key_right),
        .key_down(key_down),
        .key_rotate_cw(key_rotate_cw),
        .key_rotate_ccw(key_rotate_ccw),
        .key_drop(key_drop),
        .key_hold(key_hold),
        .key_drop_held(raw_drop), // Connect raw state for lockout
        .display(display_field),
        .score(score),
        .game_over(game_over),
        .t_next_disp(t_next),
        .t_hold_disp(t_hold),
        .hold_used_out(hold_used),
        .current_level_out(current_level),
        .ghost_y(ghost_y),
        .t_curr_out(t_curr),
        .total_lines_cleared_out(total_lines_cleared)
    );


    //
    ================================================================================

    // CDC: Game State Synchronization (game_clk     pix_clk)
    // Frame-sync to reduce CDC traffic from 83MHz to 60Hz
    //
    ================================================================================
```

```
306     field_t display_field_sync;
307     logic [31:0] score_sync;
308     logic game_over_sync;
309     tetromino_ctrl t_next_sync, t_hold_sync, t_curr_sync;
310     logic hold_used_sync;
311     logic [3:0] current_level_sync;
312     logic [7:0] total_lines_cleared_sync;
313     logic signed [`FIELD_VERTICAL_WIDTH : 0] ghost_y_sync;
314 
315     // Vsync edge detection for frame sync
316     logic vsync_prev;
317     logic frame_sync_pulse;
318 
319     always_ff @(posedge pix_clk) begin
320         if (rst) begin
321             vsync_prev <= 0;
322             frame_sync_pulse <= 0;
323         end else begin
324             vsync_prev <= vsync_raw;
325             frame_sync_pulse <= vsync_raw && !vsync_prev; // Rising
    edge of vsync
326         end
327     end
328 
329     // Only sync game state once per frame (60 Hz instead of 83.46 MHz)
330     always_ff @(posedge pix_clk) begin
331         if (rst) begin
332             display_field_sync <= '0;
333             score_sync <= 0;
334             game_over_sync <= 0;
335             t_next_sync <= '0;
336             t_hold_sync <= '0;
337             t_curr_sync <= '0;
338             hold_used_sync <= 0;
339             current_level_sync <= 0;
340             total_lines_cleared_sync <= 0;
341             ghost_y_sync <= 0;
342         end else if (frame_sync_pulse) begin  //     KEY CHANGE: Only
    update at frame boundary
343             display_field_sync <= display_field;
344             score_sync <= score;
345             game_over_sync <= game_over;
346             t_next_sync <= t_next;
347             t_hold_sync <= t_hold;
348             t_curr_sync <= t_curr;
349             hold_used_sync <= hold_used;
350             current_level_sync <= current_level;
351             total_lines_cleared_sync <= total_lines_cleared;
352             ghost_y_sync <= ghost_y;
353         end
354         // else: Hold previous values (no update)
355     end
356 
357     // VGA Output (Raw)
358     logic [10:0] curr_x_raw;
359     logic [9:0]  curr_y_raw;
360     logic active_area_raw;
```

```
    logic hsync_raw, vsync_raw;

    vga_out vga_inst (
        .clk(pix_clk),
        .rst(rst),
        .curr_x(curr_x_raw),
        .curr_y(curr_y_raw),
        .hsync(hsync_raw),
        .vsync(vsync_raw),
        .active_area(active_area_raw)
    );

    // Sprite ROM
    logic [3:0] sprite_addr_x;
    logic [3:0] sprite_addr_y;
    logic [11:0] sprite_pixel;

    block_sprite sprite_inst (
        .clk(pix_clk),
        .addr_x(sprite_addr_x),
        .addr_y(sprite_addr_y),
        .pixel_out(sprite_pixel)
    );

    // Drawing Logic (Raw)
    logic [3:0] vga_r_raw, vga_g_raw, vga_b_raw;
    logic hsync_pipelined, vsync_pipelined;

    draw_tetris draw_inst (
    .clk(pix_clk),
    .rst(rst),
    .curr_x(curr_x_raw),
    .curr_y(curr_y_raw),
    .active_area(active_area_raw),
    .hsync_in(hsync_raw),
    .vsync_in(vsync_raw),

    .display(display_field_sync),        //     Use synchronized
    .score(score_sync),                  //     Use synchronized
    .game_over(game_over_sync),          //     Use synchronized
    .t_next(t_next_sync),                //     Use synchronized
    .t_hold(t_hold_sync),                //     Use synchronized
    .hold_used(hold_used_sync),          //     Use synchronized
    .current_level(current_level_sync),  //     Use synchronized
    .total_lines_cleared(total_lines_cleared_sync), //     Use
   synchronized
    .ghost_y(ghost_y_sync),              //     Use synchronized
    .t_curr(t_curr_sync),                //     Use synchronized

    .sprite_addr_x(sprite_addr_x),
    .sprite_addr_y(sprite_addr_y),
    .sprite_pixel(sprite_pixel),

    .vga_r(vga_r_raw),
    .vga_g(vga_g_raw),
    .vga_b(vga_b_raw),

    .hsync_out(hsync_pipelined),
```

```
    .vsync_out(vsync_pipelined)
);

    // Output Pipeline (Final Stage)
    // We keep this stage for clean output timing, effectively making
   it a 4-stage pipeline.
    always_ff @(posedge pix_clk) begin
        VGA_R <= vga_r_raw;
        VGA_G <= vga_g_raw;
        VGA_B <= vga_b_raw;
        VGA_HS <= hsync_pipelined;
        VGA_VS <= vsync_pipelined;
    end

    //
   ============================================================================

    // 7-Segment Display for Keyboard Input Visualization
    //
   ============================================================================

    seg7_key_display seg7_inst (
        .clk(CLK100MHZ),
        .rst(rst),
        .scan_code(scan_code),
        .key_valid(key_event_pulse),
        .key_left(raw_left),
        .key_right(raw_right),
        .key_down(raw_down),
        .key_rotate_cw(raw_rotate_cw),
        .key_rotate_ccw(raw_rotate_ccw),
        .key_drop(raw_drop),
        .key_hold(raw_hold),
        .SEG(SEG),
        .AN(AN),
        .DP(DP)
    );

endmodule

//
   ============================================================================

// Button Debouncer (longer timing for mechanical buttons)
//
   ============================================================================

module debouncer_btn(
    input clk,
    input I0,
    input I1,
    output reg O0,
    output reg O1
    );

    // Use larger counter for button debouncing (~10ms at 25MHz)
    reg [17:0] cnt0, cnt1;
    reg Iv0 = 0, Iv1 = 0;
```

```
467
468     localparam CNT_MAX = 250000; // ~10ms at 25MHz
469
470 always @(posedge clk) begin
471     // Debounce I0
472     if (I0 == Iv0) begin
473         if (cnt0 == CNT_MAX)
474             O0 <= I0;
475         else
476             cnt0 <= cnt0 + 1;
477     end else begin
478         cnt0 <= 18'b0;
479         Iv0 <= I0;
480     end
481
482     // Debounce I1
483     if (I1 == Iv1) begin
484         if (cnt1 == CNT_MAX)
485             O1 <= I1;
486         else
487             cnt1 <= cnt1 + 1;
488     end else begin
489         cnt1 <= 18'b0;
490         Iv1 <= I1;
491     end
492 end
493
494 endmodule
```

A.2 Global Definitions

GLOBAL.sv

Shared header file containing game-wide constants such as grid dimensions (10×20), tetromino type definitions, color indices, timing parameters, and custom data types used across all modules.

```
1  `ifndef GLOBAL_VH_
2  `define GLOBAL_VH_
3
4  // PS2 KEYBOARD SCAN CODE
5  `define SPACE_C          8'h29
6  `define ESC_C            8'h76
7
8  `define X_KEY_C          8'h22
9  `define Z_KEY_C          8'h1A
10 `define N_KEY_C          8'h31
11 `define LSHIFT_C         8'h12   // Left Shift key for HOLD
12
13 `define LEFT_ARROW_C     8'h6B
14 `define DOWN_ARROW_C     8'h72
15 `define RIGHT_ARROW_C    8'h74
16 `define UP_ARROW_C       8'h75
17
18 // FIELD
19 `define FIELD_HORIZONTAL         10
20 `define FIELD_VERTICAL_DISPLAY   20
```

```
`define FIELD_VERTICAL          22  // additional 2 blocks for spawning

`define FIELD_HORIZONTAL_WIDTH  $clog2(`FIELD_HORIZONTAL - 1)
`define FIELD_VERTICAL_WIDTH    $clog2(`FIELD_VERTICAL - 1)


// TETROMINO TYPE
`define NUMBER_OF_TETROMINO 7

`define TETROMINO_I_IDX     3'b000
`define TETROMINO_J_IDX     3'b001
`define TETROMINO_L_IDX     3'b010
`define TETROMINO_O_IDX     3'b011
`define TETROMINO_S_IDX     3'b100
`define TETROMINO_T_IDX     3'b101
`define TETROMINO_Z_IDX     3'b110
`define TETROMINO_EMPTY     3'b111

// TETROMINO AND ALL OF ITS ROTATION
// [rotation][row][column]
typedef struct packed {
          logic [0:3][0:3][0:3] data;
        } tetromino_t;

// KICK OFF
// Add this typedef to GLOBAL.sv
typedef struct packed {
    logic signed [2:0] x;
    logic signed [2:0] y;
} kick_offset_t;

// COLOR TYPE (color_mapper for more detail)
`define C_BLACK     4'b0000
`define C_FREE1     4'b0001
`define C_FREE2     4'b0010
`define C_FREE3     4'b0011
`define C_FREE4     4'b0100
`define C_FREE5     4'b0101
`define C_FREE6     4'b0110
`define C_FREE7     4'b0111
`define COLOR_I     4'b1000
`define COLOR_J     4'b1001
`define COLOR_L     4'b1010
`define COLOR_O     4'b1011
`define COLOR_S     4'b1100
`define COLOR_T     4'b1101
`define COLOR_Z     4'b1110
`define C_WHITE     4'b1111

`define C_BORDER    `C_BLACK
`define C_EMPTY     `C_WHITE

`define C_BORDER_FULL     24'h00_00_00  // black
`define C_EMPTY_FULL      24'hb2_b2_b2  // darker grey
`define C_BLOCK_BORDER    24'h66_66_66  // lighter grey

typedef struct packed {
          logic [3:0] data;
```

```
        } color_t;

// TETROMINO COLOR ALWAYS HAS START BIT 1,
// CAN BE USED TO IDENTIFY TETROMINO
typedef struct packed {
          logic [2:0] data;
        } tetromino_idx_t;

// COORDINATE OF TETROMINO
typedef struct packed {
          logic signed [`FIELD_HORIZONTAL_WIDTH : 0]    x;
          logic signed [`FIELD_VERTICAL_WIDTH : 0]      y;
        } coor_t;

// FIELD DISPLAY
typedef struct packed {
          tetromino_idx_t [0 : `FIELD_VERTICAL - 1][0 :
   `FIELD_HORIZONTAL - 1] data;
        } field_t;

// CONTROLING TETROMINO
typedef struct packed {
          tetromino_t         tetromino;
          tetromino_idx_t     idx;
          logic [1:0]         rotation; // spawn rotation
          coor_t              coordinate;
} tetromino_ctrl;

// DRAWING START AND END COORDINATE, INCLUSIVE START, EXCLUSIVE END
`define BLOCK_PIXEL 32
`define FIELD_START_X 200
`define FIELD_END_X (`FIELD_START_X + `FIELD_HORIZONTAL*`BLOCK_PIXEL)

`define FIELD_START_Y 40
`define FIELD_END_Y (`FIELD_START_Y + `FIELD_VERTICAL_DISPLAY*
   `BLOCK_PIXEL)

`define FIELD_BORDER_THICKNESS 8
`define FIELD_BORDER_START_X (`FIELD_START_X - `FIELD_BORDER_THICKNESS)
`define FIELD_BORDER_END_X (`FIELD_END_X + `FIELD_BORDER_THICKNESS)

`define FIELD_BORDER_START_Y (`FIELD_START_Y - `FIELD_BORDER_THICKNESS)
`define FIELD_BORDER_END_Y (`FIELD_END_Y + `FIELD_BORDER_THICKNESS)

`endif
```

A.3 Display Modules

vga_out.sv

*VGA timing generator. Produces horizontal/vertical sync pulses and active area signals for 1280× 800 resolution at 60 Hz. Outputs the current pixel coordinates (**curr_x**, **curr_y**) for the rendering pipeline.*

```
`timescale 1ns / 1ps
// vga_out: generates timing for 1280x800-style raster per README
   counts,
```

```
// producing hsync/vsync, active-area flag, and visible coordinates.
module vga_out(
    input  wire logic clk,
    input  wire logic rst,
    output logic [10:0] curr_x, // 0-1279
    output logic [9:0]  curr_y, // 0-799
    output logic hsync,
    output logic vsync,
    output logic active_area    // High when in visible area
    );

    // Counters
    logic [10:0] hcount;
    logic [9:0]  vcount;

    // Parameters from README
    localparam H_MAX = 1679;
    localparam V_MAX = 827;

    localparam H_SYNC_END = 135;
    localparam V_SYNC_END = 2;

    localparam H_VIS_START = 336;
    localparam H_VIS_END   = 1615;
    localparam V_VIS_START = 27;
    localparam V_VIS_END   = 826;

    always_ff @(posedge clk) begin
        if (rst) begin
            hcount <= 0;
            vcount <= 0;
        end else begin
            if (hcount == H_MAX) begin
                hcount <= 0;
                if (vcount == V_MAX) begin
                    vcount <= 0;
                end else begin
                    vcount <= vcount + 1;
                end
            end else begin
                hcount <= hcount + 1;
            end
        end
    end

    // Sync signals
    // README: hsync 0 when hcount 0-135, else 1. (Active Low)
    assign hsync = ~(hcount <= H_SYNC_END);

    // README: vsync 1 when vcount 0-2, else 0. (Active High)
    assign vsync = (vcount <= V_SYNC_END);

    // Active Area
    assign active_area = (hcount >= H_VIS_START && hcount <= H_VIS_END)
    &&
                         (vcount >= V_VIS_START && vcount <= V_VIS_END)
   ;

```

```
    // Current X and Y (relative to visible area)
    always_comb begin
        if (active_area) begin
            curr_x = hcount - H_VIS_START;
            curr_y = vcount - V_VIS_START;
        end else begin
            curr_x = 0;
            curr_y = 0;
        end
    end

endmodule
```

draw_tetris.sv

Main rendering engine. Implements a 4-stage pipeline to determine the color of each pixel by checking if it falls within the game grid, next piece preview, hold area, score display, or UI borders.

```
// draw_tetris: VGA renderer for the Tetris UI. Pipelines coordinates through
// region detection, text prerendering, block/ghost drawing, and final color
// mapping for the grid, next/hold previews, score, level, and heartbeat.
`include "../GLOBAL.sv"

module draw_tetris(
    input  wire logic clk,
    input  wire logic rst,

    input  wire logic [10:0] curr_x,
    input  wire logic [9:0]  curr_y,
    input  wire logic active_area,
    input  wire logic hsync_in,
    input  wire logic vsync_in,

    input   field_t     display,
    input  logic [31:0]    score,
    input  logic           game_over,
    input  tetromino_ctrl  t_next, // Next piece
    input  tetromino_ctrl  t_hold, // Hold piece
    input  logic           hold_used, // Whether hold was used this piece
    input  logic [3:0]     current_level, // Game level
    input  logic [7:0]     total_lines_cleared, // NEW: For level bar
    input  logic signed [`FIELD_VERTICAL_WIDTH : 0] ghost_y,
    input  tetromino_ctrl  t_curr, // Current piece for ghost rendering

    output logic [3:0]     sprite_addr_x,
    output logic [3:0]     sprite_addr_y,
    input  logic [11:0]    sprite_pixel,

    // VGA Output
    output logic [3:0]     vga_r,
    output logic [3:0]     vga_g,
    output logic [3:0]     vga_b,
```

```
    output logic            hsync_out,
    output logic            vsync_out
    );

    // Constants
    localparam BLOCK_SIZE = 32;
    localparam GRID_W = `FIELD_HORIZONTAL * BLOCK_SIZE; // 320
    localparam GRID_H = `FIELD_VERTICAL_DISPLAY * BLOCK_SIZE; // 640

    // Centered Grid
    localparam GRID_X_START = (1280 - GRID_W) / 2; // 480
    localparam GRID_Y_START = (800 - GRID_H) / 2;  // 80

    // Right Sidebar (Next Piece & Score)
    localparam SIDE_X_START = GRID_X_START + GRID_W + 50;
    localparam NEXT_Y_START = GRID_Y_START;
    localparam SCORE_Y_START = NEXT_Y_START + 200;
    localparam MESSAGE_Y_START = SCORE_Y_START + 100; // Below score
    localparam LEVEL_Y_START = MESSAGE_Y_START + 50;

    // Left Sidebar (Hold Piece)
    localparam HOLD_X_START = GRID_X_START - 200;
    localparam HOLD_Y_START = GRID_Y_START;

    // Colors - Vibrant scheme for black background
    logic [11:0] color_map [0:7];
    initial begin
        color_map[0] = 12'h000; // Empty (Black)
        color_map[1] = 12'h0FF; // I - Cyan
        color_map[2] = 12'h00F; // J - Blue
        color_map[3] = 12'hF80; // L - Orange
        color_map[4] = 12'hFF0; // O - Yellow
        color_map[5] = 12'h0F0; // S - Green
        color_map[6] = 12'hF0F; // T - Magenta
        color_map[7] = 12'hF00; // Z - Red
    end

    //
    ============================================================================

    // PIPELINE STAGE 1: Coordinate Calculation & Region Detection
    //
    ============================================================================


    // Stage 1 Registers
    logic s1_active_area;
    logic s1_hsync, s1_vsync;
    logic [10:0] s1_curr_x;
    logic [9:0]  s1_curr_y;

    // Regions
    logic s1_is_grid;
    logic s1_is_border;
    logic s1_is_next;
    logic s1_is_hold;
    logic s1_is_score;
    logic s1_is_level;
```

```
    logic s1_is_heartbeat;

    // Grid Coordinates
    logic signed [11:0] s1_rel_x, s1_rel_y;
    logic signed [11:0] s1_grid_col, s1_grid_row;
    logic [4:0] s1_block_pixel_x, s1_block_pixel_y;

    // Sidebar relative coords
    logic [10:0] s1_next_rel_x, s1_hold_rel_x;
    logic [9:0]  s1_next_rel_y, s1_hold_rel_y;

    always_ff @(posedge clk) begin
        // Pass-through control signals
        s1_active_area <= active_area;
        s1_hsync <= hsync_in;
        s1_vsync <= vsync_in;
        s1_curr_x <= curr_x;
        s1_curr_y <= curr_y;

        // Calculate Relative Coordinates
        s1_rel_x <= curr_x - GRID_X_START;
        s1_rel_y <= curr_y - GRID_Y_START;

        // Calculate Grid Indices
        s1_grid_col <= (curr_x - GRID_X_START) / BLOCK_SIZE;
        s1_grid_row <= (curr_y - GRID_Y_START) / BLOCK_SIZE;
        s1_block_pixel_x <= (curr_x - GRID_X_START) % BLOCK_SIZE;
        s1_block_pixel_y <= (curr_y - GRID_Y_START) % BLOCK_SIZE;

        // Region Detection
        s1_is_grid <= (curr_x >= GRID_X_START && curr_x < GRID_X_START
    + GRID_W &&
                       curr_y >= GRID_Y_START && curr_y < GRID_Y_START
    + GRID_H);

        s1_is_border <= (curr_x >= GRID_X_START - 4 && curr_x <
    GRID_X_START + GRID_W + 4 &&
                         curr_y >= GRID_Y_START - 4 && curr_y <
    GRID_Y_START + GRID_H + 4) &&
                        !(curr_x >= GRID_X_START && curr_x <
    GRID_X_START + GRID_W &&
                          curr_y >= GRID_Y_START && curr_y <
    GRID_Y_START + GRID_H);

        s1_is_next <= (curr_x >= SIDE_X_START && curr_x < SIDE_X_START
    + 150 &&
                       curr_y >= NEXT_Y_START && curr_y < NEXT_Y_START
    + 150);

        s1_is_hold <= (curr_x >= HOLD_X_START && curr_x < HOLD_X_START
    + 150 &&
                       curr_y >= HOLD_Y_START && curr_y < HOLD_Y_START
    + 150);

        s1_is_score <= (curr_x >= SIDE_X_START && curr_x < SIDE_X_START
     + 200 &&
                        curr_y >= SCORE_Y_START && curr_y <
    SCORE_Y_START + 100);
```

```
135
136         s1_is_level <= (curr_x >= SIDE_X_START && curr_x < SIDE_X_START
    + 200 &&
137                         curr_y >= LEVEL_Y_START && curr_y <
    LEVEL_Y_START + 100);
138
139         s1_is_heartbeat <= (curr_x >= GRID_X_START + GRID_W - 10 &&
    curr_x < GRID_X_START + GRID_W &&
140                             curr_y >= GRID_Y_START + GRID_H - 10 &&
    curr_y < GRID_Y_START + GRID_H);
141
142         // Relative coords for sidebars
143         s1_next_rel_x <= curr_x - SIDE_X_START;
144         s1_next_rel_y <= curr_y - NEXT_Y_START;
145         s1_hold_rel_x <= curr_x - HOLD_X_START;
146         s1_hold_rel_y <= curr_y - HOLD_Y_START;
147     end
148
149     //
    ================================================================================
150     // PRE-RENDERED TEXT BITMAPS (Option 1 - Frame-Based Rendering)
151     //
    ================================================================================
152
153     // Bitmap storage for pre-rendered text (updated once per frame at
    60Hz)
154     // Score region: 200 pixels wide    60 pixels tall
155     // Level region: 200 pixels wide    20 pixels tall
156     logic [199:0] score_text_bitmap [0:59];
157     logic [199:0] level_text_bitmap [0:19];
158
159     // Frame boundary detection
160     logic vsync_prev;
161     logic frame_start;
162
163     always_ff @(posedge clk) begin
164         vsync_prev <= vsync_in;
165         frame_start <= vsync_in && !vsync_prev; // Rising edge of vsync
166     end
167
168     // Text rendering state machine (runs during vblank)
169     typedef enum logic [2:0] {
170         TR_IDLE,
171         TR_SCORE,
172         TR_LEVEL,
173         TR_DONE
174     } text_render_state_t;
175
176     text_render_state_t tr_state;
177     logic [7:0] tr_y_counter;
178     logic [7:0] tr_x_counter;
179
180     // Intermediate signals for text rendering
181     logic [10:0] render_curr_x;
182     logic [9:0] render_curr_y;
183     logic render_score_pixel;
```

```
    logic render_level_pixel;

    // Text module instantiations (for rendering - NOT for display)
    draw_number score_renderer (
        .curr_x(render_curr_x),
        .curr_y(render_curr_y),
        .pos_x(11'd0),  // Relative to bitmap origin
        .pos_y(10'd0),
        .number(score),
        .pixel_on(render_score_pixel)
    );

    logic [7:0] render_level_chars [0:15];
    logic [3:0] render_level_len;

    always_comb begin
        render_level_chars[0] = 8'h4C; // L
        render_level_chars[1] = 8'h45; // E
        render_level_chars[2] = 8'h56; // V
        render_level_chars[3] = 8'h45; // E
        render_level_chars[4] = 8'h4C; // L
        render_level_chars[5] = 8'h3A; // :
        render_level_chars[6] = 8'h20; // Space
        render_level_chars[7] = 8'h20; // Space (default)
        render_level_chars[8] = 8'h20; // Space (default)
        render_level_chars[9] = 8'h20;
        render_level_chars[10] = 8'h20;
        render_level_chars[11] = 8'h20;
        render_level_chars[12] = 8'h20;
        render_level_chars[13] = 8'h20;
        render_level_chars[14] = 8'h20;
        render_level_chars[15] = 8'h20;

        if (current_level < 10) begin
            render_level_chars[7] = 8'h30 + current_level;
            render_level_len = 8;
        end else begin
            render_level_chars[7] = 8'h31;
            render_level_chars[8] = 8'h30 + (current_level - 10);
            render_level_len = 9;
        end

    end

    draw_string_line level_renderer (
        .curr_x(render_curr_x),
        .curr_y(render_curr_y),
        .pos_x(11'd0),
        .pos_y(10'd0),
        .str_chars(render_level_chars),
        .str_len(render_level_len),
        .scale(2'd2),
        .pixel_on(render_level_pixel)
    );

    // State machine to render text to bitmaps once per frame (
   PIPELINED)
always_ff @(posedge clk) begin
```

```
241     if (rst) begin
242         tr_state <= TR_IDLE;
243         tr_y_counter <= 0;
244         tr_x_counter <= 0;
245         render_curr_x <= 0;
246         render_curr_y <= 0;
247     end else begin
248         case (tr_state)
249             TR_IDLE: begin
250                 if (frame_start) begin
251                     tr_state <= TR_SCORE;
252                     tr_y_counter <= 0;
253                     tr_x_counter <= 0;
254                     // Prime the pipeline
255                     render_curr_x <= 0;
256                     render_curr_y <= 0;
257                 end
258             end
259 
260             TR_SCORE: begin
261                 // PIPELINE: Set coordinates this cycle, store result
    NEXT cycle
262                 // (Text modules process combinationally, result
    available next clock)
263 
264                 // Store the result from PREVIOUS cycle
265                 if (tr_x_counter > 0 || tr_y_counter > 0) begin
266                     score_text_bitmap[tr_y_counter][tr_x_counter] <=
    render_score_pixel;
267                 end
268 
269                 // Advance to next pixel and set coordinates for NEXT
    cycle
270                 if (tr_x_counter < 199) begin
271                     tr_x_counter <= tr_x_counter + 1;
272                     render_curr_x <= tr_x_counter + 1;
273                     render_curr_y <= tr_y_counter;
274                 end else begin
275                     tr_x_counter <= 0;
276                     render_curr_x <= 0;
277                     if (tr_y_counter < 59) begin
278                         tr_y_counter <= tr_y_counter + 1;
279                         render_curr_y <= tr_y_counter + 1;
280                     end else begin
281                         // Done with score, move to level
282                         tr_state <= TR_LEVEL;
283                         tr_y_counter <= 0;
284                         tr_x_counter <= 0;
285                         render_curr_x <= 0;
286                         render_curr_y <= 0;
287                     end
288                 end
289             end
290 
291             TR_LEVEL: begin
292                 // Store result from previous cycle
293                 if (tr_x_counter > 0 || tr_y_counter > 0) begin
```

```
                    level_text_bitmap[tr_y_counter][tr_x_counter] <=
    render_level_pixel;
                end

                // Advance and set coordinates for next cycle
                if (tr_x_counter < 199) begin
                    tr_x_counter <= tr_x_counter + 1;
                    render_curr_x <= tr_x_counter + 1;
                    render_curr_y <= tr_y_counter;
                end else begin
                    tr_x_counter <= 0;
                    render_curr_x <= 0;
                    if (tr_y_counter < 19) begin
                        tr_y_counter <= tr_y_counter + 1;
                        render_curr_y <= tr_y_counter + 1;
                    end else begin
                        tr_state <= TR_DONE;
                    end
                end
            end

            TR_DONE: begin
                tr_state <= TR_IDLE; // Wait for next frame
            end
        endcase
    end
end


    //
    ================================================================================

    // FAST TEXT LOOKUP (replaces Stage 1.5, 1.75 entirely)
    //
    ================================================================================


    // Fast bitmap lookup (just 1 cycle - no 32-level logic!)
    logic s1b_score_pixel_on;
    logic s1b_level_text_pixel_on;

    always_ff @(posedge clk) begin
        s1b_score_pixel_on <= 0;
        s1b_level_text_pixel_on <= 0;

        // Score lookup
        if (s1_is_score && s1_curr_y >= SCORE_Y_START + 40 && s1_curr_y
    < SCORE_Y_START + 100) begin
            automatic int bmp_y = s1_curr_y - (SCORE_Y_START + 40);
            automatic int bmp_x = s1_curr_x - SIDE_X_START;
            if (bmp_x >= 0 && bmp_x < 200 && bmp_y >= 0 && bmp_y < 60)
    begin
                s1b_score_pixel_on <= score_text_bitmap[bmp_y][bmp_x];
            end
        end

        // Level lookup
```

```
344         if (s1_is_level && s1_curr_y >= LEVEL_Y_START && s1_curr_y <
    LEVEL_Y_START + 20) begin
345             automatic int bmp_y = s1_curr_y - LEVEL_Y_START;
346             automatic int bmp_x = s1_curr_x - SIDE_X_START;
347             if (bmp_x >= 0 && bmp_x < 200 && bmp_y >= 0 && bmp_y < 20)
    begin
348                 s1b_level_text_pixel_on <= level_text_bitmap[bmp_y][
    bmp_x];
349             end
350         end
351     end
352
353     //
    ============================================================================
354     // PIPELINE STAGE 2: Data Access & Logic
355     //
    ============================================================================
356
357     // Stage 2 Registers
358     logic s2_active_area;
359     logic s2_hsync, s2_vsync;
360
361     // Region Flags
362     logic s2_is_border;
363     logic s2_is_grid;
364     logic s2_is_next;
365     logic s2_is_hold;
366     logic s2_is_score;
367     logic s2_is_level;
368     logic s2_is_heartbeat;
369     logic s2_is_grid_line;
370     logic s2_is_ghost;
371
372     // Visual Data
373     logic [3:0] s2_cell_color_idx;
374     logic [3:0] s2_sprite_addr_x;
375     logic [3:0] s2_sprite_addr_y;
376
377     // Text/UI Pixels
378     logic s2_score_pixel;
379     logic s2_level_text_pixel;
380     logic s2_level_bar_pixel;
381     logic [3:0] s2_level_bar_color_idx;
382     logic s2_heartbeat_pixel;
383     logic s2_level_bar_border;
384
385     // Headers
386     logic s2_next_header;
387     logic s2_hold_header;
388     logic s2_score_header;
389     logic s2_level_header;
390     logic s2_hold_used_header;
391
392     // Heartbeat Counter
393     logic [25:0] heartbeat_cnt;
394     always_ff @(posedge clk) begin
```

```
395         heartbeat_cnt <= heartbeat_cnt + 1;
396     end
397 
398     // Stage 2 Logic Block
399     always_ff @(posedge clk) begin
400         // Pass-through control signals
401         s2_active_area <= s1_active_area;
402         s2_hsync <= s1_hsync;
403         s2_vsync <= s1_vsync;
404 
405         // Pass through all region flags
406         s2_is_border <= s1_is_border;
407         s2_is_grid <= s1_is_grid;
408         s2_is_next <= s1_is_next;
409         s2_is_hold <= s1_is_hold;
410         s2_is_score <= s1_is_score;
411         s2_is_level <= s1_is_level;
412         s2_is_heartbeat <= s1_is_heartbeat;
413 
414         // Defaults
415         s2_cell_color_idx <= 0;
416         s2_sprite_addr_x <= 0;
417         s2_sprite_addr_y <= 0;
418         s2_is_ghost <= 0;
419         s2_is_grid_line <= 0;
420         s2_level_bar_border <= 0;
421 
422         s2_next_header <= 0;
423         s2_hold_header <= 0;
424         s2_score_header <= 0;
425         s2_level_header <= 0;
426         s2_hold_used_header <= 0;
427         s2_score_pixel <= 0;
428         s2_level_text_pixel <= 0;
429         s2_level_bar_pixel <= 0;
430         s2_heartbeat_pixel <= 0;
431 
432         if (s1_curr_x >= SIDE_X_START - 2 && s1_curr_x < SIDE_X_START +
    202 &&
433             s1_curr_y >= LEVEL_Y_START + 38 && s1_curr_y <
    LEVEL_Y_START + 62) begin
434 
435             if (s1_curr_x < SIDE_X_START || s1_curr_x >= SIDE_X_START +
    200 ||
436                 s1_curr_y < LEVEL_Y_START + 40 || s1_curr_y >=
    LEVEL_Y_START + 60) begin
437                 s2_level_bar_border <= 1;
438             end
439         end
440 
441 
442         // 1. Grid Logic
443         if (s1_is_grid) begin
444             // Grid Lines Detection
445             if (s1_block_pixel_x == 0 || s1_block_pixel_x == 31 ||
446                 s1_block_pixel_y == 0 || s1_block_pixel_y == 31) begin
447                 s2_is_grid_line <= 1;
448             end
```

```
449
450             // Check Grid Bounds & Data
451             if (s1_grid_col >= 0 && s1_grid_col < `FIELD_HORIZONTAL &&
452                 s1_grid_row >= 0 && s1_grid_row <
    `FIELD_VERTICAL_DISPLAY) begin
453
454                 // Main Block
455                 if (display.data[s1_grid_row + 2][s1_grid_col].data !=
    `TETROMINO_EMPTY) begin
456                     s2_cell_color_idx <= display.data[s1_grid_row + 2][
    s1_grid_col].data + 1;
457                     s2_sprite_addr_x <= s1_block_pixel_x[4:1];
458                     s2_sprite_addr_y <= s1_block_pixel_y[4:1];
459                 end
460                 // Ghost Piece
461                 else if (!game_over) begin
462                     if ((s1_grid_row + 2) >= ghost_y && (s1_grid_row +
    2) < ghost_y + 4 &&
463                         s1_grid_col >= t_curr.coordinate.x &&
    s1_grid_col < t_curr.coordinate.x + 4) begin
464
465                         if (t_curr.tetromino.data[t_curr.rotation][
    s1_grid_row + 2 - ghost_y][s1_grid_col - t_curr.coordinate.x]) begin
466                             s2_is_ghost <= 1;
467                             s2_sprite_addr_x <= s1_block_pixel_x[4:1];
468                             s2_sprite_addr_y <= s1_block_pixel_y[4:1];
469                         end
470                     end
471                 end
472             end
473         end
474
475         // 2. Next Piece Logic
476         else if (s1_is_next) begin
477             if (s1_next_rel_y < 20) begin
478                 s2_next_header <= 1;
479             end else begin
480                 // Draw Piece
481                 automatic logic [10:0] nx = s1_next_rel_x - 20;
482                 automatic logic [9:0] ny = s1_next_rel_y - 40;
483                 automatic logic [2:0] nr = ny / BLOCK_SIZE;
484                 automatic logic [2:0] nc = nx / BLOCK_SIZE;
485
486                 if (nx < 128 && ny < 128 && nr < 4 && nc < 4) begin
487                     if (t_next.tetromino.data[0][nr][nc]) begin
488                         s2_cell_color_idx <= t_next.idx.data + 1;
489                         s2_sprite_addr_x <= (nx % BLOCK_SIZE) >> 1;
490                         s2_sprite_addr_y <= (ny % BLOCK_SIZE) >> 1;
491                     end
492                 end
493             end
494         end
495
496         // 3. Hold Piece Logic
497         else if (s1_is_hold) begin
498             if (s1_hold_rel_y < 20) begin
499                 s2_hold_header <= 1;
500                 s2_hold_used_header <= hold_used;
```

```
            end else begin
                if (t_hold.idx.data != `TETROMINO_EMPTY) begin
                    automatic logic [10:0] hx = s1_hold_rel_x - 20;
                    automatic logic [9:0] hy = s1_hold_rel_y - 40;
                    automatic logic [2:0] hr = hy / BLOCK_SIZE;
                    automatic logic [2:0] hc = hx / BLOCK_SIZE;

                    if (hx < 128 && hy < 128 && hr < 4 && hc < 4) begin
                        if (t_hold.tetromino.data[0][hr][hc]) begin
                            s2_cell_color_idx <= t_hold.idx.data + 1;
                            s2_sprite_addr_x <= (hx % BLOCK_SIZE) >> 1;
                            s2_sprite_addr_y <= (hy % BLOCK_SIZE) >> 1;
                        end
                    end
                end
            end
        end

        // 4. Score Logic
        else if (s1_is_score) begin
            if (s1_curr_y < SCORE_Y_START + 20) begin
                s2_score_header <= 1;
            end
            s2_score_pixel <= s1b_score_pixel_on;  //     NEW: From
buffered register
        end

        // 5. Level Logic
        else if (s1_is_level) begin
            if (s1_curr_y < LEVEL_Y_START + 20) begin
                s2_level_header <= 1;
            end

            s2_level_text_pixel <= s1b_level_text_pixel_on;  //     NEW
: From buffered registeron;

            // // Level Bar Border (2px wide white border)
            // if (s1_curr_x >= SIDE_X_START - 2 && s1_curr_x <
SIDE_X_START + 202 &&
            //     s1_curr_y >= LEVEL_Y_START + 38 && s1_curr_y <
LEVEL_Y_START + 62) begin

            //     if (s1_curr_x < SIDE_X_START || s1_curr_x >=
SIDE_X_START + 200 ||
            //         s1_curr_y < LEVEL_Y_START + 40 || s1_curr_y >=
LEVEL_Y_START + 60) begin
            //         s2_level_bar_border <= 1;
            //     end
            // end

            // Level Bar
            if (s1_curr_x >= SIDE_X_START && s1_curr_x < SIDE_X_START +
 ((total_lines_cleared % 10) * 20) &&
                s1_curr_y >= LEVEL_Y_START + 40 && s1_curr_y <
LEVEL_Y_START + 60) begin
                s2_level_bar_pixel <= 1;
                if (current_level < 5) s2_level_bar_color_idx <= 0;
```

```
                else if (current_level < 10) s2_level_bar_color_idx <=
    1;
                else s2_level_bar_color_idx <= 2;
            end
        end

        // 6. Heartbeat
        if (s1_is_heartbeat && heartbeat_cnt[25] && game_over) begin
            s2_heartbeat_pixel <= 1;
        end
    end

    //
    ================================================================================

    // PIPELINE STAGE 3: Sprite Lookup Alignment
    //
    ================================================================================


    // Stage 3 intermediate for sprite lookup
    logic [3:0] s3_sprite_addr_x;
    logic [3:0] s3_sprite_addr_y;

    // Output sprite address for ROM lookup (1 cycle before color
    mapping)
    always_ff @(posedge clk) begin
        sprite_addr_x <= s2_sprite_addr_x;
        sprite_addr_y <= s2_sprite_addr_y;

        // Also pass these to next stage for alignment
        s3_sprite_addr_x <= s2_sprite_addr_x;
        s3_sprite_addr_y <= s2_sprite_addr_y;
    end

    // Stage 3 Registers (aligned with sprite_pixel output)
    logic s3_active_area;
    logic s3_hsync, s3_vsync;
    logic s3_is_border;
    logic s3_is_grid;
    logic s3_is_grid_line;
    logic s3_is_ghost;
    logic s3_is_next;
    logic s3_is_hold;
    logic [3:0] s3_cell_color_idx;
    logic s3_next_header;
    logic s3_hold_header;
    logic s3_hold_used_header;
    logic s3_score_header;
    logic s3_score_pixel;
    logic s3_level_header;
    logic s3_level_text_pixel;
    logic s3_level_bar_pixel;
    logic [3:0] s3_level_bar_color_idx;
    logic s3_heartbeat_pixel;
    logic s3_level_bar_border;

    always_ff @(posedge clk) begin
```

```
602         // Pass through all flags
603         s3_active_area <= s2_active_area;
604         s3_hsync <= s2_hsync;
605         s3_vsync <= s2_vsync;
606         s3_is_border <= s2_is_border;
607         s3_is_grid <= s2_is_grid;
608         s3_is_grid_line <= s2_is_grid_line;
609         s3_is_ghost <= s2_is_ghost;
610         s3_is_next <= s2_is_next;
611         s3_is_hold <= s2_is_hold;
612         s3_cell_color_idx <= s2_cell_color_idx;
613         s3_next_header <= s2_next_header;
614         s3_hold_header <= s2_hold_header;
615         s3_hold_used_header <= s2_hold_used_header;
616         s3_score_header <= s2_score_header;
617         s3_score_pixel <= s2_score_pixel;
618         s3_level_header <= s2_level_header;
619         s3_level_text_pixel <= s2_level_text_pixel;
620         s3_level_bar_pixel <= s2_level_bar_pixel;
621         s3_level_bar_color_idx <= s2_level_bar_color_idx;
622         s3_heartbeat_pixel <= s2_heartbeat_pixel;
623         s3_level_bar_border <= s2_level_bar_border;
624     end
625 
626     //
    ================================================================================
627     // PIPELINE STAGE 4: Color Mapping & Final Output
628     //
    ================================================================================
629 
630     always_ff @(posedge clk) begin
631         hsync_out <= s3_hsync;
632         vsync_out <= s3_vsync;
633 
634         // Default: BLACK background
635         vga_r <= 4'h0;
636         vga_g <= 4'h0;
637         vga_b <= 4'h0;
638 
639         if (s3_active_area) begin
640             // 1. Grid Border - White
641             if (s3_is_border) begin
642                 vga_r <= 4'hC; vga_g <= 4'hC; vga_b <= 4'hC;
643             end
644 
645             // 2. Grid Lines - Dark Grey
646             else if (s3_is_grid_line) begin
647                 vga_r <= 4'h3; vga_g <= 4'h3; vga_b <= 4'h3;
648             end
649 
650             // 3. Grid Content (Blocks and Ghost)
651             else if (s3_is_grid && (s3_cell_color_idx != 0 ||
    s3_is_ghost)) begin
652                 automatic logic [3:0] r, g, b;
653                 automatic logic [3:0] inten = sprite_pixel[7:4];
654 
```

```
                if (s3_is_ghost) begin
                    // Ghost piece - Semi-transparent white
                    r = 4'h5; g = 4'h5; b = 4'h5;
                end else begin
                    // Normal block rendering
                    if (game_over) begin
                        // Grey out blocks when game over
                        r = 4'h4; g = 4'h4; b = 4'h4;
                    end else begin
                        r = color_map[s3_cell_color_idx][11:8];
                        g = color_map[s3_cell_color_idx][7:4];
                        b = color_map[s3_cell_color_idx][3:0];
                    end
                end

                // Apply sprite shading
                if (inten != 4'hF) begin
                    vga_r <= r >> 1;
                    vga_g <= g >> 1;
                    vga_b <= b >> 1;
                end else begin
                    vga_r <= r;
                    vga_g <= g;
                    vga_b <= b;
                end
            end

            // 4. Next Piece Header - Bright White
            else if (s3_next_header) begin
                vga_r <= 4'hF; vga_g <= 4'hF; vga_b <= 4'hF;
            end
            // Next Piece Blocks
            else if (s3_is_next && s3_cell_color_idx != 0) begin
                automatic logic [3:0] r, g, b;
                automatic logic [3:0] inten = sprite_pixel[7:4];

                r = color_map[s3_cell_color_idx][11:8];
                g = color_map[s3_cell_color_idx][7:4];
                b = color_map[s3_cell_color_idx][3:0];

                if (inten != 4'hF) begin
                    vga_r <= r >> 1; vga_g <= g >> 1; vga_b <= b >> 1;
                end else begin
                    vga_r <= r; vga_g <= g; vga_b <= b;
                end
            end

            // 5. Hold Piece Header
            else if (s3_hold_header) begin
                if (s3_hold_used_header) begin
                    // Grayed out when used
                    vga_r <= 4'h5; vga_g <= 4'h5; vga_b <= 4'h5;
                end else begin
                    // Bright cyan when available
                    vga_r <= 4'h0; vga_g <= 4'hE; vga_b <= 4'hE;
                end
            end
            // Hold Piece Blocks
```

```
713             else if (s3_is_hold && s3_cell_color_idx != 0) begin
714                 automatic logic [3:0] r, g, b;
715                 automatic logic [3:0] inten = sprite_pixel[7:4];
716 
717                 if (hold_used) begin
718                     // Dim when used
719                     r = 4'h4; g = 4'h4; b = 4'h4;
720                 end else begin
721                     r = color_map[s3_cell_color_idx][11:8];
722                     g = color_map[s3_cell_color_idx][7:4];
723                     b = color_map[s3_cell_color_idx][3:0];
724                 end
725 
726                 if (inten != 4'hF) begin
727                     vga_r <= r >> 1; vga_g <= g >> 1; vga_b <= b >> 1;
728                 end else begin
729                     vga_r <= r; vga_g <= g; vga_b <= b;
730                 end
731             end
732 
733             // 6. Score Header - Yellow/Gold
734             else if (s3_score_header) begin
735                 vga_r <= 4'hF; vga_g <= 4'hC; vga_b <= 4'h0;
736             end
737             // Score Numbers - White
738             else if (s3_score_pixel) begin
739                 vga_r <= 4'hF; vga_g <= 4'hF; vga_b <= 4'hF;
740             end
741 
742             // 7. Level Text - Cyan
743             else if (s3_level_header && s3_level_text_pixel) begin
744                 vga_r <= 4'h0; vga_g <= 4'hD; vga_b <= 4'hD;
745             end
746             // Level Bar Border - Light grey
747             else if (s3_level_bar_border) begin
748                 vga_r <= 4'h8; vga_g <= 4'h8; vga_b <= 4'h8;
749             end
750             // Level Bar - Gradient based on level
751             else if (s3_level_bar_pixel) begin
752                 case (s3_level_bar_color_idx)
753                     0: begin vga_r <= 4'h0; vga_g <= 4'hE; vga_b <= 4'
    h0; end  // Bright Green (easy)
754                     1: begin vga_r <= 4'hF; vga_g <= 4'hC; vga_b <= 4'
    h0; end  // Orange (medium)
755                     2: begin vga_r <= 4'hF; vga_g <= 4'h0; vga_b <= 4'
    h0; end  // Red (hard)
756                     default: begin vga_r <= 4'h0; vga_g <= 4'hE; vga_b
    <= 4'h0; end
757                 endcase
758             end
759 
760             // 8. Heartbeat - Blinking white when game over
761             else if (s3_heartbeat_pixel) begin
762                 vga_r <= 4'hF; vga_g <= 4'hF; vga_b <= 4'hF;
763             end
764         end
765         // Outside active area, stays black (default)
766     end
```

```
767
768 endmodule
```

block_sprite.sv

Sprite ROM for tetromino blocks. Stores a 16×16 grayscale texture that is tinted with per-piece colors during rendering to give blocks a beveled 3D appearance.

```
1  `timescale 1ns / 1ps
2  // block_sprite: 16x16 ROM-backed sprite for a beveled block tile,
       returning
3  // a 12-bit RGB pixel for the requested x/y address.
4  module block_sprite(
5      input  wire logic clk,
6      input  wire logic [3:0] addr_x, // 0-15
7      input  wire logic [3:0] addr_y, // 0-15
8      output logic [11:0] pixel_out   // 12-bit RGB
9      );
10
11     (* rom_style = "block" *) logic [11:0] rom [0:255];
12
13     initial begin
14         int i, j;
15         for (i = 0; i < 16; i++) begin
16             for (j = 0; j < 16; j++) begin
17                 if (i == 0 || i == 15 || j == 0 || j == 15)
18                     rom[i*16 + j] = 12'h888; // Grey Border
19                 else if (i == 1 || i == 14 || j == 1 || j == 14)
20                     rom[i*16 + j] = 12'hAAA; // Lighter Border
21                 else
22                     rom[i*16 + j] = 12'hFFF; // White Center
23             end
24         end
25     end
26
27     always_ff @(posedge clk) begin
28         pixel_out <= rom[{addr_y, addr_x}];
29     end
30
31 endmodule
```

draw_number.sv

Numeric digit renderer. Draws score and level numbers on screen by mapping digit values to a font ROM and outputting pixel data.

```
1  // draw_number: renders up to eight hexadecimal digits at a given
       origin using
2  // a compact 4x5 bitmap font scaled for display overlays.
3  `include "../GLOBAL.sv"
4  module draw_number (
5      input  logic [10:0] curr_x,
6      input  logic [9:0]  curr_y,
7      input  logic [10:0] pos_x,
8      input  logic [9:0]  pos_y,
9      input  logic [31:0] number,
10     output logic        pixel_on
```

```
);
    logic [19:0] current_glyph;

    always_comb begin
        case (digit)
            0: current_glyph = 20'b0110_1001_1001_1001_0110;
            1: current_glyph = 20'b0010_0010_0010_0010_0010;
            2: current_glyph = 20'b0110_0001_0010_0100_1111;
            3: current_glyph = 20'b0110_0001_0010_0001_0110;
            4: current_glyph = 20'b1001_1001_1111_0001_0001;
            5: current_glyph = 20'b1111_1000_1110_0001_1110;
            6: current_glyph = 20'b0110_1000_1110_1001_0110;
            7: current_glyph = 20'b1111_0001_0010_0100_0100;
            8: current_glyph = 20'b0110_1001_0110_1001_0110;
            9: current_glyph = 20'b0110_1001_0111_0001_0110;
            default: current_glyph = '0;
        endcase
    end

    logic [3:0] digit;
    logic [2:0] dx, dy; // 0-4
    logic [3:0] digit_idx; // 0-7

    localparam SCALE = 4;
    localparam DIGIT_W = 4 * SCALE;
    localparam DIGIT_H = 5 * SCALE;
    localparam SPACING = 1 * SCALE;
    localparam TOTAL_W = DIGIT_W + SPACING;

    logic [10:0] rel_x;
    logic [9:0] rel_y;

    assign rel_x = curr_x - pos_x;
    assign rel_y = curr_y - pos_y;

    always_comb begin
        pixel_on = 0;
        if (curr_x >= pos_x && curr_x < pos_x + (8 * TOTAL_W) &&
            curr_y >= pos_y && curr_y < pos_y + DIGIT_H) begin

            digit_idx = rel_x / TOTAL_W;
            dx = (rel_x % TOTAL_W) / SCALE;
            dy = rel_y / SCALE;

            if (dx < 4 && dy < 5) begin
                case (7 - digit_idx)
                    0: digit = number[3:0];
                    1: digit = number[7:4];
                    2: digit = number[11:8];
                    3: digit = number[15:12];
                    4: digit = number[19:16];
                    5: digit = number[23:20];
                    6: digit = number[27:24];
                    7: digit = number[31:28];
                endcase

                if (digit < 10) begin
                   if (current_glyph[19 - (dy*4 + dx)]) pixel_on = 1;
```

```
                end
            end
        end
    end

endmodule
```

draw_string.sv

Text string renderer. Draws static text labels ("SCORE", "LEVEL", "NEXT", "HOLD") on the game UI using a character ROM.

```
// draw_string: renders a single ASCII character with an 8x8 bitmap font and
// optional scaling; font covers A-Z, 0-9, space, hyphen, and colon.
`include "../GLOBAL.sv"
module draw_string (
    input  logic [10:0] curr_x,
    input  logic [9:0]  curr_y,
    input  logic [10:0] pos_x,
    input  logic [9:0]  pos_y,
    input  logic [7:0]  char_code,  // ASCII character code
    input  logic [1:0]  scale,      // Scale factor (1, 2, 3...)
    output logic        pixel_on
);

    logic [63:0] font_data;

    always_comb begin
        case (char_code)
            // Space
            8'h20: font_data = 64'h0000000000000000;
            // Numbers
            8'h30: font_data = 64'h3C666E7666663C00; // 0
            8'h31: font_data = 64'h1818181818181800; // 1
            8'h32: font_data = 64'h3C060C1830607E00; // 2
            8'h33: font_data = 64'h3C06061C06063C00; // 3
            8'h34: font_data = 64'h060E1E667F060600; // 4
            8'h35: font_data = 64'h7E607C0606663C00; // 5
            8'h36: font_data = 64'h3C60607C66663C00; // 6
            8'h37: font_data = 64'h7E060C1818181800; // 7
            8'h38: font_data = 64'h3C66663C66663C00; // 8
            8'h39: font_data = 64'h3C66663E06063C00; // 9
            // Letters
            8'h41: font_data = 64'h3C66667E66666600; // A
            8'h42: font_data = 64'h7C66667C66667C00; // B
            8'h43: font_data = 64'h3C66606060663C00; // C
            8'h44: font_data = 64'h786C6666666C7800; // D
            8'h45: font_data = 64'h7E60607860607E00; // E
            8'h46: font_data = 64'h7E60607860606000; // F
            8'h47: font_data = 64'h3C66606E66663C00; // G
            8'h48: font_data = 64'h6666667E66666600; // H
            8'h49: font_data = 64'h3C18181818183C00; // I
            8'h4A: font_data = 64'h1E0C0C0C6C6C3800; // J
            8'h4B: font_data = 64'h666C7870786C6600; // K
            8'h4C: font_data = 64'h6060606060607E00; // L
            8'h4D: font_data = 64'h63777F6B63636300; // M
```

```
            8'h4E: font_data = 64'h66767E7E6E666600; // N
            8'h4F: font_data = 64'h3C66666666663C00; // O
            8'h50: font_data = 64'h7C66667C60606000; // P
            8'h51: font_data = 64'h3C66666E6C663A00; // Q
            8'h52: font_data = 64'h7C66667C786C6600; // R
            8'h53: font_data = 64'h3C60603C06063C00; // S
            8'h54: font_data = 64'h7E18181818181800; // T
            8'h55: font_data = 64'h6666666666663C00; // U
            8'h56: font_data = 64'h66666666663C1800; // V
            8'h57: font_data = 64'h63636B7F77636300; // W
            8'h58: font_data = 64'h66663C183C666600; // X
            8'h59: font_data = 64'h6666663C18181800; // Y
            8'h5A: font_data = 64'h7E060C1830607E00; // Z
            // Hyphen
            8'h2D: font_data = 64'h0000007E00000000; // -
            // Colon
            8'h3A: font_data = 64'h0000180000180000; // :
            default: font_data = 64'h0000000000000000;
        endcase
    end

    localparam CHAR_W = 8;
    localparam CHAR_H = 8;
    localparam CHAR_SPACING = 1;

    logic [10:0] rel_x, rel_y;
    logic [2:0] pixel_x, pixel_y;

    assign rel_x = curr_x - pos_x;
    assign rel_y = curr_y - pos_y;

    logic [10:0] scaled_rel_x, scaled_rel_y;
    assign scaled_rel_x = rel_x / scale;
    assign scaled_rel_y = rel_y / scale;

    assign pixel_x = scaled_rel_x[2:0];
    assign pixel_y = scaled_rel_y[2:0];

    always_comb begin
        pixel_on = 0;
        if (rel_x >= 0 && rel_x < (CHAR_W * scale) &&
            rel_y >= 0 && rel_y < (CHAR_H * scale)) begin
            if (font_data[63 - (pixel_y * 8 + pixel_x)]) begin
                pixel_on = 1;
            end
        end
    end

endmodule

// draw_string_line: renders a short string (up to 16 chars) by tiling
// draw_string instances with configurable scale and spacing.
module draw_string_line (
    input  logic [10:0] curr_x,
    input  logic [9:0]  curr_y,
    input  logic [10:0] pos_x,
    input  logic [9:0]  pos_y,
    input  logic [7:0]  str_chars [0:15],  // Up to 16 characters
```

```
    input  logic [3:0]  str_len,            // Actual string length
    input  logic [1:0]  scale,              // Scale factor
    output logic        pixel_on
);

    localparam CHAR_W = 8;
    localparam CHAR_SPACING = 1;
    // localparam CHAR_TOTAL_W = CHAR_W + CHAR_SPACING;

    logic [10:0] char_total_w;
    assign char_total_w = (CHAR_W + CHAR_SPACING) * scale;

    logic [10:0] rel_x, rel_y;
    logic [3:0] char_idx;
    logic [7:0] current_char;
    logic char_pixel_on;

    assign rel_x = curr_x - pos_x;
    assign rel_y = curr_y - pos_y;

    // Determine which character we're in
    assign char_idx = rel_x / char_total_w;

    // Get the character code for this position
    always_comb begin
        if (char_idx < str_len) begin
            current_char = str_chars[char_idx];
        end else begin
            current_char = 8'h20; // Space
        end
    end

    // Calculate position within character
    logic [10:0] char_pos_x;
    assign char_pos_x = pos_x + (char_idx * char_total_w);

    draw_string char_draw (
        .curr_x(curr_x),
        .curr_y(curr_y),
        .pos_x(char_pos_x),
        .pos_y(pos_y),
        .char_code(current_char),
        .scale(scale),
        .pixel_on(char_pixel_on)
    );

    assign pixel_on = char_pixel_on;

endmodule
```

seg7_key_display.sv

7-segment display driver. Shows debug information (current key code, game state) on the FPGA board's 8-digit 7-segment display using time-multiplexed digit scanning.

```
`timescale 1ns / 1ps
// seg7_key_display: multiplexed 7-segment driver that shows the active
    control
```

```
// key (with priority) and its scan code across four digits.
module seg7_key_display(
    input  logic        clk,           // System clock (100MHz)
    input  logic        rst,

    // Current key state
    input  logic [7:0]  scan_code,     // Current scan code
    input  logic        key_valid,     // Key is valid (pressed)

    // Active key indicators (directly from game)
    input  logic        key_left,
    input  logic        key_right,
    input  logic        key_down,
    input  logic        key_rotate_cw,
    input  logic        key_rotate_ccw,
    input  logic        key_drop,
    input  logic        key_hold,

    // 7-Segment outputs
    output logic [6:0]  SEG,           // Segment pattern (active low)
    output logic [7:0]  AN,            // Anode select (active low)
    output logic        DP             // Decimal point (active low)
);

    // Clock divider for multiplexing (need ~1kHz refresh rate)
    // 100MHz / 2^17     763 Hz
    logic [19:0] clk_div;
    logic [2:0] digit_sel;

    always_ff @(posedge clk) begin
        if (rst)
            clk_div <= 0;
        else
            clk_div <= clk_div + 1;
    end

    assign digit_sel = clk_div[19:17];

    // Determine what to display based on active key
    // Priority: Drop > Hold > Rotate > Left > Right > Down
    logic [3:0] display_char [0:7];  // Character for each digit

    // Character encoding for 7-segment (active low)
    // Segments: gfedcba
    function logic [6:0] char_to_seg(input [3:0] c);
        case (c)
            // Numbers
            4'h0: return 7'b1000000; // 0
            4'h1: return 7'b1111001; // 1
            4'h2: return 7'b0100100; // 2
            4'h3: return 7'b0110000; // 3
            4'h4: return 7'b0011001; // 4
            4'h5: return 7'b0010010; // 5
            4'h6: return 7'b0000010; // 6
            4'h7: return 7'b1111000; // 7
            4'h8: return 7'b0000000; // 8
            4'h9: return 7'b0010000; // 9
            4'hA: return 7'b0001000; // A
```

```
            4'hB: return 7'b0000011; // b
            4'hC: return 7'b1000110; // C
            4'hD: return 7'b0100001; // d
            4'hE: return 7'b0000110; // E
            4'hF: return 7'b0001110; // F
            default: return 7'b1111111; // blank
        endcase
    endfunction

    // Special characters for key names
    localparam [6:0] SEG_L = 7'b1000111; // L
    localparam [6:0] SEG_r = 7'b0101111; // r (lowercase)
    localparam [6:0] SEG_d = 7'b0100001; // d (lowercase)
    localparam [6:0] SEG_U = 7'b1000001; // U
    localparam [6:0] SEG_H = 7'b0001001; // H
    localparam [6:0] SEG_S = 7'b0010010; // S (same as 5)
    localparam [6:0] SEG_P = 7'b0001100; // P
    localparam [6:0] SEG_o = 7'b0100011; // o (lowercase)
    localparam [6:0] SEG_t = 7'b0000111; // t (lowercase)
    localparam [6:0] SEG_n = 7'b0101011; // n (lowercase)
    localparam [6:0] SEG_DASH = 7'b0111111; // -
    localparam [6:0] SEG_BLANK = 7'b1111111; // blank

    // What to display on each digit
    logic [6:0] seg_data [0:7];

    always_comb begin
        // Default: show dashes (no key)
        seg_data[7] = SEG_DASH;  // Leftmost
        seg_data[6] = SEG_DASH;
        seg_data[5] = SEG_DASH;
        seg_data[4] = SEG_DASH;
        seg_data[3] = SEG_BLANK;
        seg_data[2] = SEG_BLANK;
        seg_data[1] = SEG_BLANK;
        seg_data[0] = SEG_BLANK; // Rightmost

        // Show key name on left 4 digits, scan code on right 2 digits
        if (key_drop) begin
            // "droP" for drop/space
            seg_data[7] = SEG_d;
            seg_data[6] = SEG_r;
            seg_data[5] = SEG_o;
            seg_data[4] = SEG_P;
            // Scan code 0x29
            seg_data[1] = 7'b0100100; // 2
            seg_data[0] = 7'b0010000; // 9
        end
        else if (key_hold) begin
            // "HoLd" for hold
            seg_data[7] = SEG_H;
            seg_data[6] = SEG_o;
            seg_data[5] = SEG_L;
            seg_data[4] = SEG_d;
            // Scan code 0x12
            seg_data[1] = 7'b1111001; // 1
            seg_data[0] = 7'b0100100; // 2
        end
```

```
        else if (key_rotate_cw) begin
            // "rot r" for rotate CW
            seg_data[7] = SEG_r;
            seg_data[6] = SEG_o;
            seg_data[5] = SEG_t;
            seg_data[4] = SEG_r;
            // Scan code (generic or specific)
            seg_data[1] = 7'b1111000; // 7
            seg_data[0] = 7'b0010010; // 5
        end
        else if (key_rotate_ccw) begin
            // "rot L" for rotate CCW
            seg_data[7] = SEG_r;
            seg_data[6] = SEG_o;
            seg_data[5] = SEG_t;
            seg_data[4] = SEG_L;
            // Scan code
            seg_data[1] = 7'b1111001; // 1
            seg_data[0] = 7'b0001000; // A (1A for Z)
        end
        else if (key_left) begin
            // "LEFt" for left
            seg_data[7] = SEG_L;
            seg_data[6] = 7'b0000110; // E
            seg_data[5] = 7'b0001110; // F
            seg_data[4] = SEG_t;
            // Scan code 0x6B
            seg_data[1] = 7'b0000010; // 6
            seg_data[0] = 7'b0000011; // b
        end
        else if (key_right) begin
            // "rGHt" for right
            seg_data[7] = SEG_r;
            seg_data[6] = 7'b1000010; // G (approximation)
            seg_data[5] = SEG_H;
            seg_data[4] = SEG_t;
            // Scan code 0x74
            seg_data[1] = 7'b1111000; // 7
            seg_data[0] = 7'b0011001; // 4
        end
        else if (key_down) begin
            // "doUn" for down
            seg_data[7] = SEG_d;
            seg_data[6] = SEG_o;
            seg_data[5] = SEG_U;
            seg_data[4] = SEG_n;
            // Scan code 0x72
            seg_data[1] = 7'b1111000; // 7
            seg_data[0] = 7'b0100100; // 2
        end
    end

    // Multiplex outputs
    always_comb begin
        // All anodes off by default
        AN = 8'b11111111;
        SEG = SEG_BLANK;
        DP = 1'b1; // DP off
```

```
177
178         // Enable selected digit
179         AN[digit_sel] = 1'b0;
180         SEG = seg_data[digit_sel];
181
182         // Add decimal point between key name and scan code
183         if (digit_sel == 4)
184             DP = 1'b0;
185     end
186
187 endmodule
```

A.4 Game Logic Modules

game_control.sv

Central game state machine. Implements the main FSM with 15 states governing piece spawning, movement, rotation (with SRS wall kicks), gravity, line clearing, hold functionality, and game over detection.

```
1  /* game_control
2   * Central gameplay FSM: spawns pieces, processes movement/rotation
       with kicks,
3   * manages holds, scoring/levels, line clears, ghost piece, and exposes
        display
4   * signals plus status outputs.
5   */
6  `include "../GLOBAL.sv"
7
8  module game_control (
9      input   logic           clk,
10     input   logic           rst,
11     input   logic           tick_game,
12
13     // Inputs (Single Pulse)
14     input   logic           key_left,
15     input   logic           key_right,
16     input   logic           key_down,
17     input   logic           key_rotate_cw,
18     input   logic           key_rotate_ccw,
19     input   logic           key_drop,
20     input   logic           key_hold,
21     input   logic           key_drop_held,
22
23     output  field_t         display,
24     output  logic [31:0]    score,
25     output  logic           game_over,
26     output  tetromino_ctrl  t_next_disp,
27     output  tetromino_ctrl  t_hold_disp,
28     output  logic           hold_used_out,
29     output  logic [3:0]     current_level_out,
30     output  logic signed [`FIELD_VERTICAL_WIDTH : 0] ghost_y,
31     output  tetromino_ctrl  t_curr_out,
32
33     output  logic [7:0]     total_lines_cleared_out // Exposed for
       level bar
34   );
```

```
35
36  typedef enum logic [3:0] {
37    GEN,
38    GEN_WAIT,
39    IDLE,
40    MOVE_LEFT,
41    MOVE_RIGHT,
42    ROTATE,
43    DOWN,
44    HARD_DROP,
45    HOLD,
46    HOLD_SWAP,
47    CLEAN,
48    GAME_OVER_STATE,
49    RESET_GAME,
50    WARMUP,
51    DROP_LOCKOUT
52  } state_t;
53
54  state_t ps, ns;
55
56  tetromino_ctrl t_curr, t_curr_cand, t_gen, t_gen_next;
57  tetromino_ctrl t_check;
58  tetromino_ctrl t_hold;
59  tetromino_ctrl t_hold_old;
60  logic hold_used;
61  logic hold_empty;
62
63  logic [7:0] total_lines_cleared;
64  logic [3:0] consecutive_clears;
65
66  field_t f_curr, f_disp, f_cleaned;
67
68  logic valid;
69  logic rotate_done, rotate_success;
70  logic clean_done;
71  logic [2:0] lines_cleared;
72
73  logic [2:0] kick_attempt;
74  logic rotate_direction;
75  logic [1:0] rotation_from;
76  logic last_move_was_rotation;
77  logic [2:0] kick_used;
78
79  logic is_t_spin;
80  logic is_t_spin_mini;
81  logic is_s_spin;
82  logic is_z_spin;
83  logic is_j_spin;
84  logic is_l_spin;
85  logic is_i_spin;
86
87  logic [3:0] current_level = 0;
88  logic [31:0] drop_speed_frames;
89
90  // Derive current level from total cleared lines (capped at 15) and
     map to
```

```
  // the drop delay expressed in game ticks. Keep this combinational to avoid
  // hidden latches on the level or speed calculation.
  always_comb begin

    if (current_level != 15) current_level = total_lines_cleared / 10;

    case (current_level)
        0:  drop_speed_frames = 40;
        1:  drop_speed_frames = 37;
        2:  drop_speed_frames = 34;
        3:  drop_speed_frames = 30;
        4:  drop_speed_frames = 26;
        5:  drop_speed_frames = 21;
        6:  drop_speed_frames = 15;
        7:  drop_speed_frames = 12;
        8:  drop_speed_frames = 8;
        9:  drop_speed_frames = 6;
        10: drop_speed_frames = 5;
        11: drop_speed_frames = 5;
        12: drop_speed_frames = 5;
        13: drop_speed_frames = 4;
        14: drop_speed_frames = 3;
        15: drop_speed_frames = 2;
        default: drop_speed_frames = 2; // Impossible mode!
    endcase
  end

  // SRS KICK TABLES (Corrected for Y-down coordinate system)

localparam kick_offset_t JLSTZ_KICKS_CW[4][5] = '{
    // 0  R
    '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{-3'sd1, -3'sd1}, '{3'sd0, 3'sd2}, '{-3'sd1, 3'sd2}},
    //  R 2
    '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{3'sd1, 3'sd1}, '{3'sd0, -3'sd2}, '{3'sd1, -3'sd2}},
    // 2  L
    '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{3'sd1, -3'sd1}, '{3'sd0, 3'sd2}, '{3'sd1, 3'sd2}},
    //  L 0
    '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{-3'sd1, 3'sd1}, '{3'sd0, -3'sd2}, '{-3'sd1, -3'sd2}}
};

localparam kick_offset_t JLSTZ_KICKS_CCW[4][5] = '{
    // 0  L
    '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{3'sd1, -3'sd1}, '{3'sd0, 3'sd2}, '{3'sd1, 3'sd2}},
    //  R 0
    '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{3'sd1, 3'sd1}, '{3'sd0, -3'sd2}, '{3'sd1, -3'sd2}},
    // 2  R
    '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{-3'sd1, -3'sd1}, '{3'sd0, 3'sd2}, '{-3'sd1, 3'sd2}},
    //  L 2
    '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{-3'sd1, 3'sd1}, '{3'sd0, -3'sd2}, '{-3'sd1, -3'sd2}}
```

```
};

localparam kick_offset_t I_KICKS_CW[4][5] = '{
    // 0  R
    '{'{3'sd0, 3'sd0}, '{-3'sd2, 3'sd0}, '{3'sd1, 3'sd0}, '{-3'sd2, 3'sd1}, '{3'sd1, -3'sd2}},
    //  R 2
    '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{3'sd2, 3'sd0}, '{-3'sd1, -3'sd2}, '{3'sd2, 3'sd1}},
    // 2  L
    '{'{3'sd0, 3'sd0}, '{3'sd2, 3'sd0}, '{-3'sd1, 3'sd0}, '{3'sd2, -3'sd1}, '{-3'sd1, 3'sd2}},
    //  L 0
    '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{-3'sd2, 3'sd0}, '{3'sd1, 3'sd2}, '{-3'sd2, -3'sd1}}
};

localparam kick_offset_t I_KICKS_CCW[4][5] = '{
    // 0  L
    '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{3'sd2, 3'sd0}, '{-3'sd1, -3'sd2}, '{3'sd2, 3'sd1}},
    //  R 0
    '{'{3'sd0, 3'sd0}, '{3'sd2, 3'sd0}, '{-3'sd1, 3'sd0}, '{3'sd2, -3'sd1}, '{-3'sd1, 3'sd2}},
    // 2  R
    '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{-3'sd2, 3'sd0}, '{3'sd1, 3'sd2}, '{-3'sd2, -3'sd1}},
    //  L 2
    '{'{3'sd0, 3'sd0}, '{-3'sd2, 3'sd0}, '{3'sd1, 3'sd0}, '{-3'sd2, 3'sd1}, '{3'sd1, -3'sd2}}
};

  kick_offset_t current_kick;
  always_comb begin
      current_kick = '{3'sd0, 3'sd0};

      if (ps == ROTATE) begin
          case (t_curr.idx.data)
              `TETROMINO_I_IDX: begin
                  if (rotate_direction)
                      current_kick = I_KICKS_CW[rotation_from][kick_attempt];
                  else
                      current_kick = I_KICKS_CCW[rotation_from][kick_attempt];
              end
              `TETROMINO_O_IDX: begin
                  current_kick = '{3'sd0, 3'sd0};
              end
              default: begin
                  if (rotate_direction)
                      current_kick = JLSTZ_KICKS_CW[rotation_from][kick_attempt];
                  else
                      current_kick = JLSTZ_KICKS_CCW[rotation_from][kick_attempt];
              end
          endcase
```

```
        end
    end

    // ================================================================
    // SPIN DETECTION LOGIC
    // ================================================================
    // Checks if piece is surrounded by blocks/walls (3-corner rule for T-spins)
    // ================================================================

    spin_detector spin_detect (
        .t_ctrl(t_curr),
        .f(f_curr),
        .last_move_was_rotation(last_move_was_rotation),
        .kick_used(kick_used),
        .is_t_spin(is_t_spin),
        .is_t_spin_mini(is_t_spin_mini),
        .is_s_spin(is_s_spin),
        .is_z_spin(is_z_spin),
        .is_j_spin(is_j_spin),
        .is_l_spin(is_l_spin),
        .is_i_spin(is_i_spin)
    );

    // Timers
    integer drop_timer;
    integer lock_timer; // NEW: Lock delay timer

    localparam LOCK_DELAY_MAX = 30; // ~0.5s at 60Hz

    // Submodules
    generate_tetromino gen (
      .clk(clk),
      .rst(rst),
      .enable(ps == GEN || ps == WARMUP),
      .t_out(t_gen),
      .t_next_out(t_gen_next)
    );

    tetromino_ctrl t_rotated;
    rotate_tetromino rot (
      .clk(clk),
      .enable(ps == ROTATE && kick_attempt == 0),
      .clockwise(rotate_direction),
      .t_in(t_curr),
      .t_out(t_rotated),
      .success(rotate_success),
      .done(rotate_done)
    );

    check_valid validator (
      .t_ctrl(t_check),
      .f(f_curr),
      .isValid(valid)
    );

    create_field merger (
      .t_ctrl(t_curr),
```

```
    .f(f_curr),
    .f_out(f_disp)
  );

  clean_field cleaner (
    .clk(clk),
    .enable(ps == CLEAN),
    .f_in(f_curr),
    .f_out(f_cleaned),
    .lines_cleared(lines_cleared),
    .done(clean_done)
  );

  ghost_calc ghost (
    .t_curr(t_curr),
    .f(f_curr),
    .ghost_y(ghost_y)
  );

  assign t_curr_out = t_curr;
  assign display = f_disp;
  assign t_next_disp = t_gen_next;
  assign t_hold_disp = t_hold;
  assign hold_used_out = hold_used;
  assign current_level_out = current_level;
  assign total_lines_cleared_out = total_lines_cleared;

  // Candidate Logic for Validation
  always_comb begin
    t_check = t_curr;
    case (ps)
        GEN_WAIT: t_check = t_gen;
        MOVE_LEFT: t_check.coordinate.x = t_curr.coordinate.x - 1;
        MOVE_RIGHT: t_check.coordinate.x = t_curr.coordinate.x + 1;
        DOWN: t_check.coordinate.y = t_curr.coordinate.y + 1;
        HARD_DROP: t_check.coordinate.y = t_curr.coordinate.y + 1;
        IDLE: t_check.coordinate.y = t_curr.coordinate.y + 1;
        ROTATE: begin
            t_check = t_rotated;
            t_check.coordinate.x = t_rotated.coordinate.x +
   current_kick.x;
            t_check.coordinate.y = t_rotated.coordinate.y +
   current_kick.y;
        end
        HOLD_SWAP: begin
            t_check = t_hold_old;
            t_check.coordinate.x = 3;
            t_check.coordinate.y = 0;
            t_check.rotation = 0;
        end
        default: t_check = t_curr;
    endcase
  end

  // FSM Next State Logic
  always_comb begin
    ns = ps;
    case (ps)
```

```
299         GEN: ns = GEN_WAIT;
300 
301         GEN_WAIT: begin
302             if (valid) ns = IDLE;
303             else ns = GAME_OVER_STATE;
304         end
305 
306         IDLE: begin
307             if (key_drop) ns = HARD_DROP;
308             else if (key_hold && !hold_used) ns = HOLD;
309             else if (key_rotate_cw || key_rotate_ccw) ns = ROTATE;
310             else if (key_left) ns = MOVE_LEFT;
311             else if (key_right) ns = MOVE_RIGHT;
312             else if (drop_timer >= drop_speed_frames || key_down) ns =
    DOWN;
313             // Lock Delay Logic: If on ground (!valid) and timer
    expired -> Lock
314             else if (!valid && lock_timer >= LOCK_DELAY_MAX) ns = CLEAN
    ;
315         end
316 
317         MOVE_LEFT: ns = IDLE;
318         MOVE_RIGHT: ns = IDLE;
319 
320         ROTATE: begin
321             if (kick_attempt == 0 && !rotate_done) begin
322                 ns = ROTATE;
323             end else if (valid) begin
324                 ns = IDLE;
325             end else if (kick_attempt < 4) begin
326                 ns = ROTATE;
327             end else begin
328                 ns = IDLE;
329             end
330         end
331 
332         DOWN: begin
333             if (valid) begin
334                 ns = IDLE;
335             end else begin
336                 ns = IDLE;
337             end
338         end
339 
340         HARD_DROP: begin
341             if (valid) ns = HARD_DROP;
342             else ns = CLEAN;
343         end
344 
345         HOLD: begin
346             if (hold_empty) ns = GEN;
347             else ns = HOLD_SWAP;
348         end
349 
350         HOLD_SWAP: begin
351             if (valid) ns = IDLE;
352             else ns = GAME_OVER_STATE;
353         end
```

```
354
355         CLEAN: begin
356             if (clean_done) ns = DROP_LOCKOUT;
357         end
358
359         GAME_OVER_STATE: begin
360             if (key_drop) ns = RESET_GAME;
361         end
362
363         RESET_GAME: ns = WARMUP;
364         WARMUP: ns = GEN;
365
366         DROP_LOCKOUT: begin
367             if (!key_drop_held) ns = GEN;
368         end
369       endcase
370   end
371
372   // ================================================================
373   // SCORING WITH SPINS
374   // ================================================================
375   logic [11:0] base_score_raw;
376   logic [4:0]  level_mult;
377   logic [15:0] base_total;
378   logic [9:0]  streak_bonus;
379   logic [15:0] spin_bonus;
380   logic [15:0] points_to_add;
381
382   always_comb begin
383       // Base line clear scores (NES style)
384       case (lines_cleared)
385           3'd1: base_score_raw = 12'd40;
386           3'd2: base_score_raw = 12'd100;
387           3'd3: base_score_raw = 12'd300;
388           3'd4: base_score_raw = 12'd1200;  // Tetris!
389           default: base_score_raw = 12'd0;
390       endcase
391
392       level_mult = (current_level < 15) ? (current_level + 1) : 5'd16;
393       base_total = base_score_raw * level_mult;
394
395       // Streak bonus
396       if (lines_cleared != 0 && consecutive_clears > 0) begin
397           if (consecutive_clears >= 5)
398               streak_bonus = 10'd500;
399           else
400               streak_bonus = consecutive_clears * 10'd100;
401       end else begin
402           streak_bonus = 10'd0;
403       end
404
405       //
    ================================================================
406       // SPIN BONUSES (Modern Tetris Guidelines)
407       //
    ================================================================
408       spin_bonus = 16'd0;
409
```

```
410         // T-Spin scoring (most valuable)
411         if (is_t_spin && !is_t_spin_mini) begin
412             case (lines_cleared)
413                 3'd0: spin_bonus = 16'd400 * level_mult;   // T-Spin no
    lines
414                 3'd1: spin_bonus = 16'd800 * level_mult;   // T-Spin
    Single
415                 3'd2: spin_bonus = 16'd1200 * level_mult;  // T-Spin
    Double
416                 3'd3: spin_bonus = 16'd1600 * level_mult;  // T-Spin
    Triple
417                 default: spin_bonus = 16'd0;
418             endcase
419         end
420         // T-Spin Mini (less valuable)
421         else if (is_t_spin_mini) begin
422             case (lines_cleared)
423                 3'd0: spin_bonus = 16'd100 * level_mult;   // Mini T-Spin
     no lines
424                 3'd1: spin_bonus = 16'd200 * level_mult;   // Mini T-Spin
     Single
425                 default: spin_bonus = 16'd0;
426             endcase
427         end
428         // S/Z-Spin scoring (moderate bonus)
429         else if (is_s_spin || is_z_spin) begin
430             case (lines_cleared)
431                 3'd1: spin_bonus = 16'd400 * level_mult;
432                 3'd2: spin_bonus = 16'd800 * level_mult;
433                 3'd3: spin_bonus = 16'd1200 * level_mult;
434                 default: spin_bonus = 16'd0;
435             endcase
436         end
437         // J/L-Spin scoring (moderate bonus)
438         else if (is_j_spin || is_l_spin) begin
439             case (lines_cleared)
440                 3'd1: spin_bonus = 16'd400 * level_mult;
441                 3'd2: spin_bonus = 16'd800 * level_mult;
442                 3'd3: spin_bonus = 16'd1200 * level_mult;
443                 default: spin_bonus = 16'd0;
444             endcase
445         end
446         // I-Spin (rare, moderate bonus)
447         else if (is_i_spin) begin
448             case (lines_cleared)
449                 3'd1: spin_bonus = 16'd400 * level_mult;
450                 3'd2: spin_bonus = 16'd800 * level_mult;
451                 3'd4: spin_bonus = 16'd1600 * level_mult;  // I-Spin
    Tetris
452                 default: spin_bonus = 16'd0;
453             endcase
454         end
455 
456         // Total points
457         points_to_add = base_total + streak_bonus + spin_bonus;
458     end
459 
460     // BCD Conversion
```

```
461  logic [3:0] bcd_digits[4:0];
462  bin_to_bcd #(
463      .BINARY_WIDTH(16),
464      .BCD_DIGITS(5)
465  ) bcd_converter (
466      .binary(points_to_add),
467      .bcd(bcd_digits)
468  );
469
470  logic [3:0] bcd_ones, bcd_tens, bcd_hundreds, bcd_thousands,
      bcd_tenthousands;
471  always_comb begin
472      bcd_ones = bcd_digits[0];
473      bcd_tens = bcd_digits[1];
474      bcd_hundreds = bcd_digits[2];
475      bcd_thousands = bcd_digits[3];
476      bcd_tenthousands = bcd_digits[4];
477  end
478
479  // State Update & Data Path
480  always_ff @(posedge clk) begin
481    if (rst) begin
482        ps <= WARMUP;
483        f_curr.data <= '1;
484        t_curr.idx.data <= `TETROMINO_EMPTY;
485        t_curr.tetromino.data <= '0;
486        t_hold.idx.data <= `TETROMINO_EMPTY;
487        t_hold.tetromino.data <= '0;
488        t_hold_old.idx.data <= `TETROMINO_EMPTY;
489        t_hold_old.tetromino.data <= '0;
490        hold_empty <= 1;
491        hold_used <= 0;
492        total_lines_cleared <= 0;
493        consecutive_clears <= 0;
494        score <= 0;
495        game_over <= 0;
496        drop_timer <= 0;
497        kick_attempt <= 0;
498        rotate_direction <= 1;
499        rotation_from <= 0;
500        last_move_was_rotation <= 0;
501        kick_used <= 0;
502        lock_timer <= 0;
503    end else begin
504        ps <= ns;
505
506        if (tick_game) begin
507            if (ps == IDLE) begin
508                if (valid) begin
509                    drop_timer <= drop_timer + 1;
510                end
511
512                // Lock timer: increment when on ground, reset when in
      air
513                if (!valid) begin
514                    if (lock_timer < LOCK_DELAY_MAX)
515                        lock_timer <= lock_timer + 1;
516                end else begin
```

```
                    lock_timer <= 0;
                end
            end
        end

        if (ps == DOWN || ps == GEN) drop_timer <= 0;

        // Data Path Updates
        case (ps)
            IDLE: begin
                kick_attempt <= 0;
                if (key_rotate_cw) begin
                    rotate_direction <= 1;
                    rotation_from <= t_curr.rotation;
                end else if (key_rotate_ccw) begin
                    rotate_direction <= 0;
                    rotation_from <= t_curr.rotation;
                end

                // Clear last move flag if moving left/right/down
                if (key_left || key_right || key_down) begin
                    last_move_was_rotation <= 0;
                end

                // Lock piece when timer expires (right before transitioning to CLEAN)
                if (ns == CLEAN) begin
                    f_curr <= f_disp;
                end
            end


            GEN_WAIT: begin
                t_curr <= t_gen;
                lock_timer <= 0;
                hold_used <= 0;      // new piece allows hold again
            end

            DROP_LOCKOUT: begin
                hold_used <= 0;
                // Flags are now cleared by msg_timer
            end

            MOVE_LEFT: begin
                if (valid) begin
                    t_curr.coordinate.x <= t_curr.coordinate.x - 1;
                    last_move_was_rotation <= 0;  // Movement cancels rotation flag
                    lock_timer <= 0; // Reset lock timer on successful move
                end
            end

            MOVE_RIGHT: begin
                if (valid) begin
                    t_curr.coordinate.x <= t_curr.coordinate.x + 1;
                    last_move_was_rotation <= 0;  // Movement cancels rotation flag
```

```
                    lock_timer <= 0; // Reset lock timer on successful move
                end
            end

            ROTATE: begin
                if (kick_attempt == 0) begin
                    if (rotate_done) begin
                        if (valid) begin
                            t_curr <= t_check;
                            kick_attempt <= 0;
                            last_move_was_rotation <= 1;  // Mark as rotation move
                            kick_used <= 0;  // No kick needed
                            lock_timer <= 0; // Reset lock timer
                        end else begin
                            kick_attempt <= 1;
                        end
                    end
                end else begin
                    if (valid) begin
                        t_curr <= t_check;
                        kick_attempt <= 0;
                        last_move_was_rotation <= 1;  // Mark as rotation move
                        kick_used <= kick_attempt;  // Record which kick worked
                        lock_timer <= 0; // Reset lock timer
                    end else if (kick_attempt < 4) begin
                        kick_attempt <= kick_attempt + 1;
                    end else begin
                        kick_attempt <= 0;
                        last_move_was_rotation <= 0;  // Rotation failed
                    end
                end
            end

            DOWN: begin
                if (valid) begin
                    t_curr.coordinate.y <= t_curr.coordinate.y + 1;
                    last_move_was_rotation <= 0;  // Down movement cancels rotation
                    lock_timer <= 0; // Reset lock timer on successful move
                end
            end

            HARD_DROP: begin
                if (valid) begin
                    t_curr.coordinate.y <= t_curr.coordinate.y + 1;
                end else begin
                    f_curr <= f_disp;
                    // Hard drop cancels rotation flag (debatable - you can change this)
                    last_move_was_rotation <= 0;
                end
            end
```

```
621
622             HOLD: begin
623                 t_hold_old <= t_hold;
624                 t_hold.idx <= t_curr.idx;
625                 t_hold.tetromino <= t_curr.tetromino;
626                 t_hold.rotation <= 0;
627                 t_hold.coordinate.x <= 3;
628                 t_hold.coordinate.y <= 0;
629                 hold_used <= 1;
630                 last_move_was_rotation <= 0;
631 
632                 if (hold_empty) begin
633                     hold_empty <= 0;
634                 end
635             end
636 
637             HOLD_SWAP: begin
638                 t_curr.idx <= t_hold_old.idx;
639                 t_curr.tetromino <= t_hold_old.tetromino;
640                 t_curr.rotation <= 0;
641                 t_curr.coordinate.x <= 3;
642                 t_curr.coordinate.y <= 0;
643                 last_move_was_rotation <= 0;
644             end
645 
646             CLEAN: begin
647                 if (clean_done) begin
648                     f_curr <= f_cleaned;
649 
650                     if (lines_cleared != 0) begin
651                         total_lines_cleared <= total_lines_cleared +
    lines_cleared;
652                     end
653 
654                     if (lines_cleared != 0) begin
655                         if (consecutive_clears < 15)
656                             consecutive_clears <= consecutive_clears +
    1;
657                     end else begin
658                         consecutive_clears <= 0;
659                     end
660 
661 
662                     // BCD Addition
663                     if (lines_cleared != 0) begin
664                         automatic logic [3:0] new_ones, new_tens,
    new_hundreds, new_thousands, new_tenthousands;
665                         automatic logic c0, c1, c2, c3, c4;
666 
667                         new_ones = score[3:0] + bcd_ones;
668                         if (new_ones >= 10) begin
669                             score[3:0] <= new_ones - 10;
670                             c0 = 1;
671                         end else begin
672                             score[3:0] <= new_ones;
673                             c0 = 0;
674                         end
675 
```

```
                        new_tens = score[7:4] + bcd_tens + c0;
                        if (new_tens >= 10) begin
                            score[7:4] <= new_tens - 10;
                            c1 = 1;
                        end else begin
                            score[7:4] <= new_tens;
                            c1 = 0;
                        end

                        new_hundreds = score[11:8] + bcd_hundreds + c1;
                        if (new_hundreds >= 10) begin
                            score[11:8] <= new_hundreds - 10;
                            c2 = 1;
                        end else begin
                            score[11:8] <= new_hundreds;
                            c2 = 0;
                        end

                        new_thousands = score[15:12] + bcd_thousands +
    c2;
                        if (new_thousands >= 10) begin
                            score[15:12] <= new_thousands - 10;
                            c3 = 1;
                        end else begin
                            score[15:12] <= new_thousands;
                            c3 = 0;
                        end

                        new_tenthousands = score[19:16] +
    bcd_tenthousands + c3;
                        if (new_tenthousands >= 10) begin
                            score[19:16] <= new_tenthousands - 10;
                            c4 = 1;
                        end else begin
                            score[19:16] <= new_tenthousands;
                            c4 = 0;
                        end

                        if (c4) begin
                            if (score[23:20] == 9) begin
                                score[23:20] <= 0;
                                if (score[27:24] == 9) begin
                                    score[27:24] <= 0;
                                    score[31:28] <= score[31:28] + 1;
                                end else begin
                                    score[27:24] <= score[27:24] + 1;
                                end
                            end else begin
                                score[23:20] <= score[23:20] + 1;
                            end
                        end
                    end
                end
            end

            GAME_OVER_STATE: begin
                game_over <= 1;
            end
```

```
732
733             RESET_GAME: begin
734                 f_curr.data <= '1;
735                 t_hold.idx.data <= `TETROMINO_EMPTY;
736                 t_hold.tetromino.data <= '0;
737                 hold_empty <= 1;
738                 hold_used <= 0;
739                 total_lines_cleared <= 0;
740                 consecutive_clears <= 0;
741                 score <= 0;
742                 game_over <= 0;
743                 kick_attempt <= 0;
744                 last_move_was_rotation <= 0;
745                 kick_used <= 0;
746             end
747         endcase
748     end
749   end
750
751 endmodule
```

generate_tetromino.sv

7-Bag randomizer. Generates tetromino pieces using the standard 7-bag algorithm, ensuring all 7 piece types appear exactly once before repeating.

```
1  /* generate_tetromino
2   * 7-bag generator driven by an LFSR: maintains current and next
       tetromino
3   * controls and refreshes the bag when emptied.
4   */
5  `include "../GLOBAL.sv"
6
7  module generate_tetromino (
8      input   logic           clk,
9      input   logic           rst,
10     input   logic           enable,
11     output  tetromino_ctrl  t_out,
12     output  tetromino_ctrl  t_next_out // Optional: Next piece preview
13   );
14
15   logic [15:0] lfsr;
16   logic [2:0]  rand_idx;
17
18   always_ff @(posedge clk) begin
19     if (rst) begin
20         lfsr <= 16'hACE1;
21     end else begin
22         lfsr <= {lfsr[14:0], lfsr[15] ^ lfsr[13] ^ lfsr[12] ^ lfsr
       [10]};
23     end
24   end
25
26   assign rand_idx = lfsr[2:0] % `NUMBER_OF_TETROMINO;
27
28   tetromino_t current_shape;
29
```

```
30  logic [6:0] bag;
31  logic [2:0] selected_idx;
32  logic [6:0] next_bag;
33
34  always_comb begin
35    selected_idx = 0;
36
37    if (bag[rand_idx]) selected_idx = rand_idx;
38    else if (bag[(rand_idx + 1) % 7]) selected_idx = (rand_idx + 1) %
     7;
39    else if (bag[(rand_idx + 2) % 7]) selected_idx = (rand_idx + 2) %
     7;
40    else if (bag[(rand_idx + 3) % 7]) selected_idx = (rand_idx + 3) %
     7;
41    else if (bag[(rand_idx + 4) % 7]) selected_idx = (rand_idx + 4) %
     7;
42    else if (bag[(rand_idx + 5) % 7]) selected_idx = (rand_idx + 5) %
     7;
43    else if (bag[(rand_idx + 6) % 7]) selected_idx = (rand_idx + 6) %
     7;
44
45    next_bag = bag & ~(1 << selected_idx);
46
47    if (next_bag == 0) next_bag = 7'b1111111;
48  end
49
50  always_comb begin
51    case (selected_idx)
52        `TETROMINO_I_IDX: begin
53            current_shape.data[0] = {4'b0000, 4'b1111, 4'b0000, 4'b0000
     };
54            current_shape.data[1] = {4'b0010, 4'b0010, 4'b0010, 4'b0010
     };
55            current_shape.data[2] = {4'b0000, 4'b1111, 4'b0000, 4'b0000
     };
56            current_shape.data[3] = {4'b0010, 4'b0010, 4'b0010, 4'b0010
     };
57        end
58        `TETROMINO_J_IDX: begin
59            current_shape.data[0] = {4'b1000, 4'b1110, 4'b0000, 4'b0000
     };
60            current_shape.data[1] = {4'b0110, 4'b0100, 4'b0100, 4'b0000
     };
61            current_shape.data[2] = {4'b0000, 4'b1110, 4'b0010, 4'b0000
     };
62            current_shape.data[3] = {4'b0010, 4'b0010, 4'b0110, 4'b0000
     };
63        end
64        `TETROMINO_L_IDX: begin
65            current_shape.data[0] = {4'b0010, 4'b1110, 4'b0000, 4'b0000
     };
66            current_shape.data[1] = {4'b0100, 4'b0100, 4'b0110, 4'b0000
     };
67            current_shape.data[2] = {4'b0000, 4'b1110, 4'b1000, 4'b0000
     };
68            current_shape.data[3] = {4'b1100, 4'b0100, 4'b0100, 4'b0000
     };
69        end
```

```
70          `TETROMINO_O_IDX: begin
71              current_shape.data[0] = {4'b0110, 4'b0110, 4'b0000, 4'b0000
    };
72              current_shape.data[1] = {4'b0110, 4'b0110, 4'b0000, 4'b0000
    };
73              current_shape.data[2] = {4'b0110, 4'b0110, 4'b0000, 4'b0000
    };
74              current_shape.data[3] = {4'b0110, 4'b0110, 4'b0000, 4'b0000
    };
75          end
76          `TETROMINO_S_IDX: begin
77              current_shape.data[0] = {4'b0110, 4'b1100, 4'b0000, 4'b0000
    };
78              current_shape.data[1] = {4'b0100, 4'b0110, 4'b0010, 4'b0000
    };
79              current_shape.data[2] = {4'b0000, 4'b0110, 4'b1100, 4'b0000
    };
80              current_shape.data[3] = {4'b1000, 4'b1100, 4'b0100, 4'b0000
    };
81          end
82          `TETROMINO_T_IDX: begin
83              current_shape.data[0] = {4'b0100, 4'b1110, 4'b0000, 4'b0000
    };
84              current_shape.data[1] = {4'b0100, 4'b0110, 4'b0100, 4'b0000
    };
85              current_shape.data[2] = {4'b0000, 4'b1110, 4'b0100, 4'b0000
    };
86              current_shape.data[3] = {4'b0100, 4'b1100, 4'b0100, 4'b0000
    };
87          end
88          `TETROMINO_Z_IDX: begin
89              current_shape.data[0] = {4'b1100, 4'b0110, 4'b0000, 4'b0000
    };
90              current_shape.data[1] = {4'b0010, 4'b0110, 4'b0100, 4'b0000
    };
91              current_shape.data[2] = {4'b0000, 4'b1100, 4'b0110, 4'b0000
    };
92              current_shape.data[3] = {4'b0100, 4'b1100, 4'b1000, 4'b0000
    };
93          end
94          default: current_shape = '0;
95      endcase
96  end
97
98  always_ff @(posedge clk) begin
99      if (rst) begin
100         bag <= 7'b1111111;
101
102         t_out.idx.data <= `TETROMINO_EMPTY;
103         t_out.rotation <= 0;
104         t_out.coordinate.x <= 3;
105         t_out.coordinate.y <= 0;
106         t_out.tetromino.data <= '0;
107
108         t_next_out.idx.data <= 0; // Default I
109         t_next_out.tetromino <= '0;
110         t_next_out.rotation <= 0;
111         t_next_out.coordinate.x <= 3;
```

```
        t_next_out.coordinate.y <= 0;

    end else if (enable) begin
        t_out <= t_next_out;

        t_next_out.idx.data <= selected_idx;
        t_next_out.tetromino <= current_shape;
        t_next_out.rotation <= 0;
        t_next_out.coordinate.x <= 3;
        t_next_out.coordinate.y <= 0;

        bag <= next_bag;
    end
  end

endmodule
```

check_valid.sv

Collision detection. Checks if a proposed piece position is valid by testing for collisions with the field boundaries and locked blocks.

```
/* check_valid
 * Combinational gatekeeper: returns 1 only when the candidate
   tetromino
 * stays inside the field and does not overlap placed blocks.
 */
`include "../GLOBAL.sv"

module check_valid (
    input   tetromino_ctrl  t_ctrl,
    input   field_t         f,
    output  logic           isValid
  );

  logic [1:0] currRotation;
  assign currRotation = t_ctrl.rotation;

  integer signed i, j;
  integer signed tx, ty;

  always_comb begin
    isValid = 1'b1;
    for (i = 0; i < 4; ++i) begin
      for (j = 0; j < 4; ++j) begin
        if (t_ctrl.tetromino.data[currRotation][i][j] == 1) begin
          tx = t_ctrl.coordinate.x + j;
          ty = t_ctrl.coordinate.y + i;

          if (tx < 0 || tx >= `FIELD_HORIZONTAL ||
              ty < 0 || ty >= `FIELD_VERTICAL) begin
            isValid = 1'b0;
          end
          else begin
             if (f.data[ty][tx].data != `TETROMINO_EMPTY) begin
                isValid = 1'b0;
             end
```

```
          end
        end
      end
    end
  end
endmodule
```

clean_field.sv

Line clearing logic. Detects complete rows, removes them, shifts all rows above downward, and returns the count of lines cleared for scoring.

```
/* clean_field
 * Sequential row clearer: scans for full rows, shifts above content
   downward,
 * counts cleared lines, and exposes the updated field when done.
 */
`include "../GLOBAL.sv"

module clean_field (
    input   logic       clk,
    input   logic       enable,
    input   field_t     f_in,
    output  field_t     f_out,
    output  logic [2:0] lines_cleared,
    output  logic       done
  );

  typedef enum logic [1:0] {IDLE, CHECK, SHIFT, FINISH} state_t;
  state_t state = IDLE;

  integer row, col, k;
  logic row_full;

  field_t f_temp;

  always_ff @(posedge clk) begin
    if (!enable) begin
        state <= IDLE;
        done <= 0;
        lines_cleared <= 0;
    end else begin
        case (state)
            IDLE: begin
                f_temp <= f_in;
                row <= `FIELD_VERTICAL - 1;
                lines_cleared <= 0;
                done <= 0;
                state <= CHECK;
            end

            CHECK: begin
            if (row < 0) begin
                    state <= FINISH;
                end else begin
                    row_full = 1;
                    for (col = 0; col < `FIELD_HORIZONTAL; col++) begin
```

```
                        if (f_temp.data[row][col].data ==
    `TETROMINO_EMPTY) begin
                            row_full = 0;
                        end
                    end

                    if (row_full) begin
                        lines_cleared <= lines_cleared + 1;
                        state <= SHIFT;
                    end else begin
                        row <= row - 1;
                    end
                end
            end

            SHIFT: begin
                for (k = `FIELD_VERTICAL - 1; k > 0; k--) begin
                    if (k <= row) begin
                        f_temp.data[k] <= f_temp.data[k-1];
                    end
                end
                for (col = 0; col < `FIELD_HORIZONTAL; col++) begin
                    f_temp.data[0][col].data <= `TETROMINO_EMPTY;
                end

                state <= CHECK;
            end

            FINISH: begin
                f_out <= f_temp;
                done <= 1;
            end
        endcase
    end
  end

endmodule
```

create_field.sv

Field manipulation helper. Merges the current falling piece into the game field array when it locks in place.

```
/* create_field
 * Combines the settled field with the active tetromino to yield a
    display
 * snapshot without mutating the stored field.
 */
`include "../GLOBAL.sv"

module create_field (
    input   tetromino_ctrl  t_ctrl,
    input   field_t         f,
    output  field_t         f_out
  );

  logic [1:0] currRotation;
```

```
  assign currRotation = t_ctrl.rotation;

  integer signed i, j;
  integer signed tx, ty;

  always_comb begin
    f_out = f;

    for (i = 0; i < 4; ++i) begin
      for (j = 0; j < 4; ++j) begin
        if (t_ctrl.tetromino.data[currRotation][i][j] == 1) begin
          tx = t_ctrl.coordinate.x + j;
          ty = t_ctrl.coordinate.y + i;

          if (tx >= 0 && tx < `FIELD_HORIZONTAL &&
              ty >= 0 && ty < `FIELD_VERTICAL) begin
            f_out.data[ty][tx] = t_ctrl.idx;
          end
        end
      end
    end
  end
endmodule
```

rotate_tetromino.sv

Rotation with wall kicks. Implements the Super Rotation System (SRS) by attempting standard rotation first, then trying up to 4 alternative kick offsets if collision occurs.

```
/* rotate_tetromino
 * Produces a rotated tetromino candidate (CW or CCW) and flags
   completion;
 * rotation wrap-around relies on 2-bit arithmetic.
 */
`include "../GLOBAL.sv"

module rotate_tetromino (
    input   logic           clk,
    input   logic           enable,
    input   logic           clockwise,
    input   tetromino_ctrl  t_in,
    output  tetromino_ctrl  t_out,
    output  logic           success,
    output  logic           done
);

    always_ff @(posedge clk) begin
        if (!enable) begin
            done <= 0;
            success <= 0;
        end else begin
            t_out <= t_in;
            if (clockwise) begin
                t_out.rotation <= t_in.rotation + 1;
            end else begin
                t_out.rotation <= t_in.rotation - 1;
            end
```

```
            success <= 1;
            done <= 1;
        end
    end

endmodule
```

rotate_clockwise.sv

Basic rotation matrix. Performs a simple 90-degree clockwise rotation of a 4×4 tetromino shape matrix.

```
/* rotate_clockwise
 * Reference helper that advances rotation clockwise and flags
   completion;
 * caller is responsible for validity and kicks.
 */
`include "../GLOBAL.sv"

module rotate_clockwise (
    input   logic           clk,
    input   logic           enable,
    input   tetromino_ctrl  t_in,
    output  tetromino_ctrl  t_out,
    output  logic           success,
    output  logic           done
  );

  always_ff @(posedge clk) begin
    if (!enable) begin
        done <= 0;
        success <= 0;
    end else begin
        t_out <= t_in;
        t_out.rotation <= t_in.rotation + 1;
        success <= 1;
        done <= 1;
    end
  end

endmodule
```

ghost_calc.sv

Ghost piece calculator. Projects the current piece downward to find where it would land, enabling the semi-transparent "ghost" preview.

```
/* ghost_calc
 * Computes the ghost landing row by scanning downward until collision.
 */
`include "../GLOBAL.sv"

module ghost_calc (
    input   tetromino_ctrl  t_curr,
    input   field_t         f,
    output  logic signed [`FIELD_VERTICAL_WIDTH : 0] ghost_y
  );
```

```
11
12    logic isValid;
13    integer offset;
14
15    logic [1:0] currRotation;
16    assign currRotation = t_curr.rotation;
17
18    integer i, j;
19    integer tx, ty;
20    logic collision;
21
22    always_comb begin
23      ghost_y = t_curr.coordinate.y; // Default to current position
24
25      // Iterate downwards from current position
26      // Max drop is FIELD_VERTICAL (22)
27      for (offset = 1; offset < `FIELD_VERTICAL; offset++) begin
28          // Check if t_curr moved down by 'offset' is valid
29          collision = 0;
30
31          for (i = 0; i < 4; i++) begin
32              for (j = 0; j < 4; j++) begin
33                  if (t_curr.tetromino.data[currRotation][i][j]) begin
34                      tx = t_curr.coordinate.x + j;
35                      ty = t_curr.coordinate.y + i + offset;
36
37                      // Check Bounds
38                      if (tx < 0 || tx >= `FIELD_HORIZONTAL || ty >=
    `FIELD_VERTICAL) begin
39                          collision = 1;
40                      end
41                      // Check Field Collision
42                      else if (ty >= 0 && f.data[ty][tx].data !=
    `TETROMINO_EMPTY) begin
43                          collision = 1;
44                      end
45                  end
46              end
47          end
48
49          if (collision) begin
50              // If collision at 'offset', then 'offset-1' was the last
    valid position
51              // But we are looking for the position *before* collision.
52              // So ghost_y = t_curr.y + (offset - 1)
53              ghost_y = t_curr.coordinate.y + (offset - 1);
54              break;
55          end else begin
56              // If no collision, this position is valid. Keep searching.
57              ghost_y = t_curr.coordinate.y + offset;
58          end
59      end
60    end
61
62  endmodule
```

spin_detector.sv

T-Spin detection. Checks the 3-corner rule after T-piece rotations to determine if a T-Spin occurred for bonus scoring.

```
/* spin_detector
 * Detects spin events (T, S, Z, J, L, I) using the last rotation flag,
 * applied kick, and surrounding field occupancy.
 */
`include "../GLOBAL.sv"
module spin_detector (
    input   tetromino_ctrl  t_ctrl,
    input   field_t         f,
    input   logic           last_move_was_rotation,
    input   logic [2:0]     kick_used,

    output  logic           is_t_spin,
    output  logic           is_t_spin_mini,
    output  logic           is_s_spin,
    output  logic           is_z_spin,
    output  logic           is_j_spin,
    output  logic           is_l_spin,
    output  logic           is_i_spin
);
    logic signed [`FIELD_HORIZONTAL_WIDTH:0] cx;
    logic signed [`FIELD_VERTICAL_WIDTH:0] cy;

    assign cx = t_ctrl.coordinate.x + 1;
    assign cy = t_ctrl.coordinate.y + 1;

    function automatic logic is_blocked(
        input logic signed [`FIELD_HORIZONTAL_WIDTH:0] x,
        input logic signed [`FIELD_VERTICAL_WIDTH:0] y
    );
        if (x < 0 || x >= `FIELD_HORIZONTAL || y < 0 || y >=
   `FIELD_VERTICAL)
            return 1'b1;
        else
            return (f.data[y][x] != `TETROMINO_EMPTY);
    endfunction

    always_comb begin
        is_t_spin = 0;
        is_t_spin_mini = 0;
        is_s_spin = 0;
        is_z_spin = 0;
        is_j_spin = 0;
        is_l_spin = 0;
        is_i_spin = 0;

        if (last_move_was_rotation) begin
            case (t_ctrl.idx.data)
                `TETROMINO_T_IDX: begin
                    logic [3:0] corners_filled;
                    logic [1:0] front_corners_filled;

                    corners_filled[0] = is_blocked(cx - 1, cy - 1);
                    corners_filled[1] = is_blocked(cx + 1, cy - 1);
```

```
53                      corners_filled[2] = is_blocked(cx - 1, cy + 1);
54                      corners_filled[3] = is_blocked(cx + 1, cy + 1);
55
56                      case (t_ctrl.rotation)
57                          2'b00: front_corners_filled = corners_filled
    [1:0];
58                          2'b01: front_corners_filled = {corners_filled
    [1], corners_filled[3]};
59                          2'b10: front_corners_filled = corners_filled
    [3:2];
60                          2'b11: front_corners_filled = {corners_filled
    [0], corners_filled[2]};
61                      endcase
62
63                      if ((corners_filled[0] + corners_filled[1] +
    corners_filled[2] + corners_filled[3]) >= 3) begin
64                          if (front_corners_filled == 2'b11) begin
65                              is_t_spin = 1;
66                          end else begin
67                              is_t_spin_mini = 1;
68                          end
69                      end
70                  end
71
72                  `TETROMINO_S_IDX: begin
73                      logic [3:0] corners_filled;
74                      corners_filled[0] = is_blocked(cx - 1, cy - 1);
75                      corners_filled[1] = is_blocked(cx + 1, cy - 1);
76                      corners_filled[2] = is_blocked(cx - 1, cy + 1);
77                      corners_filled[3] = is_blocked(cx + 1, cy + 1);
78
79                      if ((corners_filled[0] + corners_filled[1] +
    corners_filled[2] + corners_filled[3]) >= 3) begin
80                          is_s_spin = 1;
81                      end
82                  end
83
84                  `TETROMINO_Z_IDX: begin
85                      logic [3:0] corners_filled;
86                      corners_filled[0] = is_blocked(cx - 1, cy - 1);
87                      corners_filled[1] = is_blocked(cx + 1, cy - 1);
88                      corners_filled[2] = is_blocked(cx - 1, cy + 1);
89                      corners_filled[3] = is_blocked(cx + 1, cy + 1);
90
91                      if ((corners_filled[0] + corners_filled[1] +
    corners_filled[2] + corners_filled[3]) >= 3) begin
92                          is_z_spin = 1;
93                      end
94                  end
95
96                  `TETROMINO_J_IDX: begin
97                      logic [3:0] corners_filled;
98                      corners_filled[0] = is_blocked(cx - 1, cy - 1);
99                      corners_filled[1] = is_blocked(cx + 1, cy - 1);
100                     corners_filled[2] = is_blocked(cx - 1, cy + 1);
101                     corners_filled[3] = is_blocked(cx + 1, cy + 1);
102
```

```
                    if ((corners_filled[0] + corners_filled[1] +
    corners_filled[2] + corners_filled[3]) >= 3) begin
                        is_j_spin = 1;
                    end
                end

                `TETROMINO_L_IDX: begin
                    logic [3:0] corners_filled;
                    corners_filled[0] = is_blocked(cx - 1, cy - 1);
                    corners_filled[1] = is_blocked(cx + 1, cy - 1);
                    corners_filled[2] = is_blocked(cx - 1, cy + 1);
                    corners_filled[3] = is_blocked(cx + 1, cy + 1);

                    if ((corners_filled[0] + corners_filled[1] +
    corners_filled[2] + corners_filled[3]) >= 3) begin
                        is_l_spin = 1;
                    end
                end

                `TETROMINO_I_IDX: begin
                    if (kick_used > 0) begin
                        is_i_spin = 1;
                    end
                end
            endcase
        end
    end
endmodule
```

bin_to_bcd.sv

Binary to BCD converter. Converts binary score values to Binary-Coded Decimal for display on the 7-segment display.

```
/* bin_to_bcd
 * Converts a binary input to packed BCD digits via the double-dabble
    method.
 */
`include "../GLOBAL.sv"

module bin_to_bcd #(
    parameter BINARY_WIDTH = 16,
    parameter BCD_DIGITS = 5
)(
    input  logic [BINARY_WIDTH-1:0] binary,
    output logic [3:0]              bcd [BCD_DIGITS-1:0]
);
    integer i, j;
    logic [BINARY_WIDTH + BCD_DIGITS*4 - 1:0] shift_reg;

    always_comb begin
        shift_reg = {{(BCD_DIGITS*4){1'b0}}, binary};

        for (i = 0; i < BINARY_WIDTH; i = i + 1) begin
            for (j = 0; j < BCD_DIGITS; j = j + 1) begin
                if (shift_reg[BINARY_WIDTH + j*4 +: 4] >= 5)
                    shift_reg[BINARY_WIDTH + j*4 +: 4] = shift_reg[
    BINARY_WIDTH + j*4 +: 4] + 3;
```

```
            end
            shift_reg = shift_reg << 1;
        end

        for (j = 0; j < BCD_DIGITS; j = j + 1) begin
            bcd[j] = shift_reg[BINARY_WIDTH + j*4 +: 4];
        end
    end
endmodule
```

A.5 Input Processing Modules

ps2_keyboard.sv

PS/2 keyboard decoder. Processes raw scan codes from PS2Receiver, handles make/break prefixes (0xF0) and extended codes (0xE0), and outputs key events.

```
`timescale 1ns / 1ps
//
    ////////////////////////////////////////////////////////////////////////////////
// PS2 Keyboard Controller
// Processes raw PS2 scan codes and provides make/break status for each
    key event.
// Handles both normal and extended (E0-prefixed) scan codes.
//
    ////////////////////////////////////////////////////////////////////////////////


module ps2_keyboard(
    input  wire logic clk,
    input  wire logic rst,
    input  wire logic ps2_clk,
    input  wire logic ps2_data,
    output logic [7:0] current_scan_code,
    output logic       current_make_break, // 1 = Make (Press), 0 =
   Break (Release)
    output logic       key_event_valid     // Pulse high for one cycle
   when a valid key event occurs
    );

    logic [31:0] keycode;
    logic [31:0] prev_keycode;

    PS2Receiver ps2_rx (
        .clk(clk),
        .kclk(ps2_clk),
        .kdata(ps2_data),
        .keycodeout(keycode)
    );

    // Extract bytes from keycode buffer
    logic [7:0] new_byte;
    logic [7:0] prev_byte;
    logic [7:0] prev_prev_byte;

    assign new_byte = keycode[7:0];
```

```
34      assign prev_byte = keycode[15:8];
35      assign prev_prev_byte = keycode[23:16];
36
37      // Extend key_event_valid pulse to ensure CDC captures it
38      // At 50MHz -> 25MHz, we need at least 2 cycles, use 4 for safety
39      logic [2:0] event_pulse_counter;
40      logic event_detected;
41
42      always_ff @(posedge clk) begin
43          if (rst) begin
44              current_scan_code <= 8'h00;
45              current_make_break <= 1'b0;
46              key_event_valid <= 1'b0;
47              prev_keycode <= 32'h0;
48              event_pulse_counter <= 0;
49              event_detected <= 0;
50          end else begin
51              // Extend pulse for CDC: keep high for 4 cycles
52              if (event_detected) begin
53                  if (event_pulse_counter < 4) begin
54                      key_event_valid <= 1'b1;
55                      event_pulse_counter <= event_pulse_counter + 1;
56                  end else begin
57                      key_event_valid <= 1'b0;
58                      event_detected <= 0;
59                      event_pulse_counter <= 0;
60                  end
61              end else begin
62                  key_event_valid <= 1'b0;
63              end
64
65              // Check if keycode buffer has changed
66              if (keycode != prev_keycode) begin
67                  prev_keycode <= keycode;
68
69                  // Skip if we just received a prefix byte (E0 or F0)
70                  if (new_byte == 8'hE0 || new_byte == 8'hF0) begin
71                      // Prefix received, wait for actual scan code
72                      // Don't trigger event
73                  end
74                  // Extended key release: E0 F0 XX
75                  else if (prev_byte == 8'hF0 && prev_prev_byte == 8'hE0)
    begin
76                      current_make_break <= 1'b0;  // Release
77                      current_scan_code <= new_byte;
78                      event_detected <= 1;
79                      event_pulse_counter <= 0;
80                  end
81                  // Normal key release: F0 XX
82                  else if (prev_byte == 8'hF0) begin
83                      current_make_break <= 1'b0;  // Release
84                      current_scan_code <= new_byte;
85                      event_detected <= 1;
86                      event_pulse_counter <= 0;
87                  end
88                  // Extended key press: E0 XX (but not E0 F0)
89                  else if (prev_byte == 8'hE0) begin
90                      current_make_break <= 1'b1;  // Press
```

```
                    current_scan_code <= new_byte;
                    event_detected <= 1;
                    event_pulse_counter <= 0;
                end
                // Normal key press: just XX (not E0, not F0, and prev wasn't F0)
                else begin
                    current_make_break <= 1'b1;  // Press
                    current_scan_code <= new_byte;
                    event_detected <= 1;
                    event_pulse_counter <= 0;
                end
            end
        end
    end

endmodule
```

PS2Receiver.sv

Low-level PS/2 protocol handler. Samples the PS/2 clock and data lines to deserialize 11-bit frames (start, 8 data, parity, stop) into raw scan codes.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////

// Company: Digilent Inc.
// Engineer: Thomas Kappenman
//
// Create Date: 03/03/2015 09:33:36 PM
// Design Name:
// Module Name: PS2Receiver
// Project Name: Nexys4DDR Keyboard Demo
// Target Devices: Nexys4DDR
// Tool Versions:
// Description: PS2 Receiver module used to shift in keycodes from a keyboard plugged into the PS2 port
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module PS2Receiver(
    input clk,
    input kclk,
    input kdata,
    output [31:0] keycodeout
    );

```

```
30
31      wire kclkf, kdataf;
32      reg [7:0]datacur;
33      reg [7:0]dataprev;
34      reg [3:0]cnt;
35      reg [31:0]keycode;
36      reg flag;
37
38      initial begin
39          keycode[31:0] <= 32'h00000000;
40          cnt <= 4'b0000;
41          flag <= 1'b0;
42          datacur <= 8'h00;
43          dataprev <= 8'h00;
44      end
45
46  debouncer debounce(
47      .clk(clk),
48      .I0(kclk),
49      .I1(kdata),
50      .O0(kclkf),
51      .O1(kdataf)
52  );
53
54  always@(negedge(kclkf))begin
55      case(cnt)
56      0:;//Start bit
57      1:datacur[0]<=kdataf;
58      2:datacur[1]<=kdataf;
59      3:datacur[2]<=kdataf;
60      4:datacur[3]<=kdataf;
61      5:datacur[4]<=kdataf;
62      6:datacur[5]<=kdataf;
63      7:datacur[6]<=kdataf;
64      8:datacur[7]<=kdataf;
65      9:flag<=1'b1;
66      10:flag<=1'b0;
67
68      endcase
69          if(cnt<=9) cnt<=cnt+1;
70          else if(cnt==10) cnt<=0;
71
72  end
73
74  always @(posedge flag)begin
75      // if (dataprev!=datacur)begin
76      keycode[31:24]<=keycode[23:16];
77      keycode[23:16]<=keycode[15:8];
78      keycode[15:8]<=dataprev;
79      keycode[7:0]<=datacur;
80      dataprev<=datacur;
81  // end
82  end
83
84  assign keycodeout=keycode;
85
86  endmodule
```

input_manager.sv

Input processing with DAS. Implements Delayed Auto Shift for smooth left/right movement and one-shot behavior for rotation, drop, and hold commands.

```
`include "../GLOBAL.sv"

module input_manager (
    input  logic clk,
    input  logic rst,
    input  logic tick_game, // 60Hz tick

    // Raw Inputs (Level)
    input  logic raw_left,
    input  logic raw_right,
    input  logic raw_down,
    input  logic raw_rotate_cw,
    input  logic raw_rotate_ccw,
    input  logic raw_drop,
    input  logic raw_hold,

    // Processed Outputs
    output logic cmd_left,   // Pulse (DAS)
    output logic cmd_right,  // Pulse (DAS)
    output logic cmd_down,   // Pulse (DAS or continuous?)
    output logic cmd_rotate_cw, // Pulse (One-shot)
    output logic cmd_rotate_ccw, // Pulse (One-shot)
    output logic cmd_drop,   // Pulse (One-shot)
    output logic cmd_hold    // Pulse (One-shot)
  );

  // Parameters for DAS (Delayed Auto Shift) - modern Tetris feel
  localparam DAS_DELAY = 6; // Frames before auto-repeat
  localparam DAS_SPEED = 2;  // Frames between repeats
  localparam SOFT_DROP_SPEED = 0;  // 0 = every frame (instant)

  // Timers and edge detectors
  logic [5:0] timer_left, timer_right, timer_down;
  logic prev_left, prev_right, prev_down, prev_rotate_cw,
   prev_rotate_ccw, prev_drop, prev_hold;

  always_ff @(posedge clk) begin
    if (rst) begin
        cmd_left <= 0; cmd_right <= 0; cmd_down <= 0;
        cmd_rotate_cw <= 0; cmd_rotate_ccw <= 0; cmd_drop <= 0;
   cmd_hold <= 0;
        timer_left <= 0; timer_right <= 0; timer_down <= 0;
        prev_left <= 0; prev_right <= 0; prev_down <= 0;
        prev_rotate_cw <= 0; prev_rotate_ccw <= 0; prev_drop <= 0;
   prev_hold <= 0;
    end else begin
        // Default low
        cmd_left <= 0;
        cmd_right <= 0;
        cmd_down <= 0;
        cmd_rotate_cw <= 0;
        cmd_rotate_ccw <= 0;
        cmd_drop <= 0;
```

```
        cmd_hold <= 0;

        // --- LEFT ---
        if (raw_left) begin
            if (!prev_left) begin
                // Initial Press
                cmd_left <= 1;
                timer_left <= 0;
            end else if (tick_game) begin
                // Holding
                if (timer_left < DAS_DELAY) begin
                    timer_left <= timer_left + 1;
                end else begin
                    // Auto Repeat
                    if (timer_left >= DAS_DELAY + DAS_SPEED) begin
                        cmd_left <= 1;
                        timer_left <= DAS_DELAY; // Reset to delay base
                    end else begin
                        timer_left <= timer_left + 1;
                    end
                end
            end
        end else begin
            timer_left <= 0;
        end
        prev_left <= raw_left;

        // --- RIGHT ---
        if (raw_right) begin
            if (!prev_right) begin
                cmd_right <= 1;
                timer_right <= 0;
            end else if (tick_game) begin
                if (timer_right < DAS_DELAY) begin
                    timer_right <= timer_right + 1;
                end else begin
                    if (timer_right >= DAS_DELAY + DAS_SPEED) begin
                        cmd_right <= 1;
                        timer_right <= DAS_DELAY;
                    end else begin
                        timer_right <= timer_right + 1;
                    end
                end
            end
        end else begin
            timer_right <= 0;
        end
        prev_right <= raw_right;

        if (raw_down) begin
            if (!prev_down) begin
                cmd_down <= 1;
                timer_down <= 0;
            end else if (tick_game) begin
                if (timer_down >= SOFT_DROP_SPEED) begin
                    cmd_down <= 1;
                    timer_down <= 0;
                end else begin
```

```
                    timer_down <= timer_down + 1;
                end
            end
        end else begin
            timer_down <= 0;
        end
        prev_down <= raw_down;

        // --- ROTATE CW (One Shot) ---
        if (raw_rotate_cw && !prev_rotate_cw) begin
            cmd_rotate_cw <= 1;
        end
        prev_rotate_cw <= raw_rotate_cw;

        // --- ROTATE CCW (One Shot) ---
        if (raw_rotate_ccw && !prev_rotate_ccw) begin
            cmd_rotate_ccw <= 1;
        end
        prev_rotate_ccw <= raw_rotate_ccw;

        // --- DROP (One Shot) ---
        if (raw_drop && !prev_drop) begin
            cmd_drop <= 1;
        end
        prev_drop <= raw_drop;

        // --- HOLD (One Shot) ---
        if (raw_hold && !prev_hold) begin
            cmd_hold <= 1;
        end
        prev_hold <= raw_hold;
    end
  end

endmodule
```

debouncer.sv

Button debouncer. Removes mechanical bounce from physical push-button inputs using a shift register and stability counter.

```
`timescale 1ns / 1ps
//
    //////////////////////////////////////////////////////////////////////////////////

// PS2 Signal Debouncer
// Based on Digilent Nexys-A7-100T-Keyboard reference implementation
// Uses fast debouncing (~20 cycles) suitable for PS2 protocol timing
//
    //////////////////////////////////////////////////////////////////////////////////


module debouncer(
    input clk,
    input I0,
    input I1,
    output reg O0,
```

```
    output reg O1
    );

    // Use 5-bit counters for fast PS2 debouncing (~20 cycles)
    // At 50MHz: 20 cycles = 400ns, well within PS2 timing requirements
    reg [4:0] cnt0 = 0, cnt1 = 0;
    reg Iv0 = 0, Iv1 = 0;

    // Initialize outputs to match expected idle state (PS2 lines idle high)
    initial begin
        O0 = 1'b1;
        O1 = 1'b1;
    end

    localparam CNT_MAX = 19; // Fast debounce for PS2 signals

always @(posedge clk) begin
    // Debounce I0
    if (I0 == Iv0) begin
        if (cnt0 == CNT_MAX)
            O0 <= I0;
        else
            cnt0 <= cnt0 + 1;
    end else begin
        cnt0 <= 5'b00000;
        Iv0 <= I0;
    end

    // Debounce I1
    if (I1 == Iv1) begin
        if (cnt1 == CNT_MAX)
            O1 <= I1;
        else
            cnt1 <= cnt1 + 1;
    end else begin
        cnt1 <= 5'b00000;
        Iv1 <= I1;
    end
end

endmodule
```

A.6 Testbenches and Simulation Logs

tb_game_control.sv

Core FSM testbench. Verifies piece spawning, movement, rotation, hold functionality, hard drop, ghost position, and game over detection.

```
`timescale 1ns / 1ps
`include "GLOBAL.sv"

/* tb_game_control
 * Integration testbench covering spawn, movement, rotation, hold, drop, and ghost.
 */
```

```
module tb_game_control;

    // Inputs
    logic clk;
    logic rst;
    logic tick_game;
    logic key_left;
    logic key_right;
    logic key_down;
    logic key_rotate_cw;
    logic key_rotate_ccw;
    logic key_drop;
    logic key_hold;
    logic key_drop_held;

    field_t display;
    logic [31:0] score;
    logic game_over;
    tetromino_ctrl t_next_disp;
    tetromino_ctrl t_hold_disp;
    logic hold_used_out;
    logic [3:0] current_level_out;
    logic signed [`FIELD_VERTICAL_WIDTH : 0] ghost_y;
    tetromino_ctrl t_curr_out;
    logic [7:0] total_lines_cleared_out;

    int pass_count = 0;
    int fail_count = 0;

    // Instantiate the Unit Under Test (UUT)
    game_control uut (
        .clk(clk),
        .rst(rst),
        .tick_game(tick_game),
        .key_left(key_left),
        .key_right(key_right),
        .key_down(key_down),
        .key_rotate_cw(key_rotate_cw),
        .key_rotate_ccw(key_rotate_ccw),
        .key_drop(key_drop),
        .key_hold(key_hold),
        .key_drop_held(key_drop_held),
        .display(display),
        .score(score),
        .game_over(game_over),
        .t_next_disp(t_next_disp),
        .t_hold_disp(t_hold_disp),
        .hold_used_out(hold_used_out),
        .current_level_out(current_level_out),
        .ghost_y(ghost_y),
        .t_curr_out(t_curr_out),
        .total_lines_cleared_out(total_lines_cleared_out)
    );

    // Clock Generation
    always #5 clk = ~clk; // 100MHz clock (10ns period)

    // Helper Task: Tick Game
```

```
65      task tick;
66          input int frames;
67          begin
68              repeat(frames) begin
69                  tick_game = 1;
70                  @(posedge clk);
71                  tick_game = 0;
72                  repeat(10) @(posedge clk);
73              end
74          end
75      endtask
76
77      // Helper Task: Pulse Key
78      task press_left;
79          begin
80              key_left = 1;
81              @(posedge clk);
82              key_left = 0;
83              @(posedge clk);
84          end
85      endtask
86
87      task press_right;
88          begin
89              key_right = 1;
90              @(posedge clk);
91              key_right = 0;
92              @(posedge clk);
93          end
94      endtask
95
96      task press_rotate_cw;
97          begin
98              key_rotate_cw = 1;
99              @(posedge clk);
100             key_rotate_cw = 0;
101             @(posedge clk);
102         end
103     endtask
104
105     task press_rotate_ccw;
106         begin
107             key_rotate_ccw = 1;
108             @(posedge clk);
109             key_rotate_ccw = 0;
110             @(posedge clk);
111         end
112     endtask
113
114     task press_drop;
115         begin
116             key_drop = 1;
117             key_drop_held = 1;
118             @(posedge clk);
119             key_drop = 0;
120             @(posedge clk);
121             // Release held after a few cycles
122             repeat(5) @(posedge clk);
```

```
            key_drop_held = 0;
            @(posedge clk);
        end
    endtask

    task press_hold;
        begin
            key_hold = 1;
            @(posedge clk);
            key_hold = 0;
            @(posedge clk);
        end
    endtask

    task press_down;
        begin
            key_down = 1;
            @(posedge clk);
            key_down = 0;
            @(posedge clk);
        end
    endtask

    initial begin
        // Initialize Inputs
        clk = 0;
        rst = 1;
        tick_game = 0;
        key_left = 0;
        key_right = 0;
        key_down = 0;
        key_rotate_cw = 0;
        key_rotate_ccw = 0;
        key_drop = 0;
        key_hold = 0;
        key_drop_held = 0;

        $display("=== Game Control Testbench ===");
        $display("Testing game mechanics including HOLD feature\n");

        // Wait 100 ns for global reset to finish
        #100;
        rst = 0;

        // Wait for Generator to spawn first piece
        repeat(20) @(posedge clk);

        //
    ================================================================
        // Test 1: Initialization
        //
    ================================================================
        $display("Test 1: Initialization");
        if (game_over == 0) begin
            $display("  PASS: Game not over at start");
            pass_count++;
        end else begin
            $display("  FAIL: Game over at start");
```

```
179             fail_count++;
180         end
181 
182         if (t_curr_out.idx.data != `TETROMINO_EMPTY) begin
183             $display("  PASS: First piece spawned (idx=%d)", t_curr_out
    .idx.data);
184             pass_count++;
185         end else begin
186             $display("  FAIL: No piece spawned");
187             fail_count++;
188         end
189 
190         //
    ================================================================
191         // Test 2: Movement
192         //
    ================================================================
193         $display("\nTest 2: Movement");
194         begin
195         logic signed [`FIELD_HORIZONTAL_WIDTH:0] start_x;
196         start_x = t_curr_out.coordinate.x;
197 
198         // Move Left
199         press_left();
200         repeat(5) @(posedge clk);
201 
202         if (t_curr_out.coordinate.x == start_x - 1) begin
203             $display("  PASS: Moved left (x: %d -> %d)", start_x,
    t_curr_out.coordinate.x);
204             pass_count++;
205         end else begin
206             $display("  FAIL: Left move failed (x: %d -> %d)", start_x,
     t_curr_out.coordinate.x);
207             fail_count++;
208         end
209 
210         // Move Right twice
211         press_right();
212         repeat(3) @(posedge clk);
213         press_right();
214         repeat(3) @(posedge clk);
215 
216         if (t_curr_out.coordinate.x == start_x + 1) begin
217             $display("  PASS: Moved right twice (x: %d)", t_curr_out.
    coordinate.x);
218             pass_count++;
219         end else begin
220             $display("  INFO: Right move result (x: %d)", t_curr_out.
    coordinate.x);
221         end
222         end
223 
224         //
    ================================================================
225         // Test 3: Rotation
226         //
    ================================================================
227         $display("\nTest 3: Rotation");
```

```
228         begin
229         logic [1:0] start_rot;
230         start_rot = t_curr_out.rotation;
231 
232         press_rotate_cw();
233         repeat(10) @(posedge clk); // Wait for rotation to complete
234 
235         // O-piece doesn't rotate visibly, but rotation state still
    changes
236         $display("  INFO: Rotation state: %d -> %d", start_rot,
    t_curr_out.rotation);
237         pass_count++;  // Rotation test is informational
238         end
239 
240         //
    ================================================================
241         // Test 4: Hold Feature (First Hold - Empty)
242         //
    ================================================================
243         $display("\nTest 4: Hold Feature - First Hold");
244         begin
245 
246         tetromino_idx_t first_piece_idx;
247         first_piece_idx = t_curr_out.idx;
248 
249         if (t_hold_disp.idx.data == `TETROMINO_EMPTY) begin
250             $display("  INFO: Hold slot is empty before hold");
251         end
252 
253         press_hold();
254         repeat(20) @(posedge clk);
255 
256         if (t_hold_disp.idx.data == first_piece_idx.data) begin
257             $display("  PASS: First piece stored in hold (idx=%d)",
    t_hold_disp.idx.data);
258             pass_count++;
259         end else begin
260             $display("  FAIL: Hold did not store piece correctly");
261             fail_count++;
262         end
263         end
264         //
    ================================================================
265         // Test 5: Hold Available After New Piece
266         //
    ================================================================
267         $display("\nTest 5: Hold Available After New Piece");
268         begin
269 
270             tetromino_idx_t piece_before_second_hold;
271             piece_before_second_hold = t_curr_out.idx;
272             begin
273             tetromino_idx_t hold_before_second_hold;
274             hold_before_second_hold = t_hold_disp.idx;
275 
276             press_hold();
277             repeat(20) @(posedge clk);
278 
```

```
279             // After second hold, pieces should swap: current becomes
    previous hold, hold becomes previous current
280             if (t_curr_out.idx.data == hold_before_second_hold.data &&
281                 t_hold_disp.idx.data == piece_before_second_hold.data)
    begin
282                 $display("  PASS: Second hold swaps current and hold
    pieces");
283                 pass_count++;
284             end else begin
285                 $display("  FAIL: Second hold did not swap as expected
    (curr=%d hold=%d)", t_curr_out.idx.data, t_hold_disp.idx.data);
286                 fail_count++;
287             end
288             end
289         end
290         //
    ================================================================
291         // Test 6: Hard Drop & Lock
292         //
    ================================================================
293         $display("\nTest 6: Hard Drop & Lock");
294         press_drop();
295         repeat(30) @(posedge clk); // Wait for drop and lockout
296
297         if (game_over == 0) begin
298             $display("  PASS: Game continues after drop");
299             pass_count++;
300         end else begin
301             $display("  FAIL: Game ended unexpectedly");
302             fail_count++;
303         end
304
305
306         //
    ================================================================
307         // Test 7: Ghost Y Position
308         //
    ================================================================
309         $display("\nTest 7: Ghost Position");
310
311         if (ghost_y >= t_curr_out.coordinate.y) begin
312             $display("  PASS: Ghost Y (%d) >= Current Y (%d)", ghost_y,
     t_curr_out.coordinate.y);
313             pass_count++;
314         end else begin
315             $display("  FAIL: Ghost Y invalid");
316             fail_count++;
317         end
318
319         //
    ================================================================
320         // Summary
321         //
    ================================================================
322         $display("\n=== Test Summary ===");
323         $display("Passed: %0d", pass_count);
324         $display("Failed: %0d", fail_count);
325         $display("Simulation Finished");
```

```
        $finish;
    end

endmodule
```

Simulation Log:

```
source tb_game_control.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform Configuration' or type 'create_wave_config' in the TCL console."
#   }
# }
# run 1000ns
=== Game Control Testbench ===
Testing game mechanics including HOLD feature

Test 1: Initialization
  PASS: Game not over at start
  PASS: First piece spawned (idx=1)

Test 2: Movement
  PASS: Moved left (x:   3 ->   2)
  PASS: Moved right twice (x:   4)

Test 3: Rotation
  INFO: Rotation state: 0 -> 1

Test 4: Hold Feature - First Hold
  INFO: Hold slot is empty before hold
  PASS: First piece stored in hold (idx=1)

Test 5: Hold Available After New Piece
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_game_control_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:12 ; elapsed = 00:00:09 . Memory (MB): peak = 8644.832 ; gain = 15.008 ; free physical = 1012901 ; free virtual = 1028180
run all
  PASS: Second hold swaps current and hold pieces

Test 6: Hard Drop & Lock
  PASS: Game continues after drop

Test 7: Ghost Position
  PASS: Ghost Y ( 20) >= Current Y ( 20)

=== Test Summary ===
Passed: 9
Failed: 0
Simulation Finished
```

```
$finish called at time : 1405 ns : File "/home/alkhalmr/gtprj/ag-tetris/test/tb_game_control.sv" Line 326
```

tb_input_manager.sv

Input processing testbench. Tests Delayed Auto Shift (DAS) timing and one-shot behavior for rotation, drop, and hold commands.

```
`timescale 1ns / 1ps
`include "GLOBAL.sv"

/* tb_input_manager
 * Verifies one-shot and DAS behavior for controller inputs.
 */
module tb_input_manager;

    logic clk;
    logic rst;
    logic tick_game;
    logic raw_left, raw_right, raw_down, raw_rotate_cw, raw_rotate_ccw, raw_drop, raw_hold;
    logic cmd_left, cmd_right, cmd_down, cmd_rotate_cw, cmd_rotate_ccw, cmd_drop, cmd_hold;

    int pass_count = 0;
    int fail_count = 0;

    input_manager uut (
        .clk(clk),
        .rst(rst),
        .tick_game(tick_game),
        .raw_left(raw_left),
        .raw_right(raw_right),
        .raw_down(raw_down),
        .raw_rotate_cw(raw_rotate_cw),
        .raw_rotate_ccw(raw_rotate_ccw),
        .raw_drop(raw_drop),
        .raw_hold(raw_hold),
        .cmd_left(cmd_left),
        .cmd_right(cmd_right),
        .cmd_down(cmd_down),
        .cmd_rotate_cw(cmd_rotate_cw),
        .cmd_rotate_ccw(cmd_rotate_ccw),
        .cmd_drop(cmd_drop),
        .cmd_hold(cmd_hold)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0; rst = 1; tick_game = 0;
        raw_left = 0; raw_right = 0; raw_down = 0;
        raw_rotate_cw = 0; raw_rotate_ccw = 0; raw_drop = 0; raw_hold = 0;

        $display("=== Input Manager Testbench ===");
        $display("Testing DAS and One-Shot behavior\n");
```

```
47
48          #20 rst = 0;
49
50          $display("Test 1: Rotate CW One-Shot");
51          @(negedge clk);  // Setup time before clock edge
52          raw_rotate_cw = 1;
53          @(posedge clk);
54          #1;
55          if (cmd_rotate_cw) begin
56              $display("  PASS: Rotate CW Triggered on press");
57              pass_count++;
58          end else begin
59              $display("  FAIL: Rotate CW did not trigger");
60              fail_count++;
61          end
62
63          @(posedge clk);
64          #1;
65          if (!cmd_rotate_cw) begin
66              $display("  PASS: Rotate CW Pulse Ended after one cycle");
67              pass_count++;
68          end else begin
69              $display("  FAIL: Rotate CW Pulse lasted too long");
70              fail_count++;
71          end
72
73          repeat(10) @(posedge clk);
74          if (!cmd_rotate_cw) begin
75              $display("  PASS: Rotate CW did not re-trigger while
    holding");
76              pass_count++;
77          end else begin
78              $display("  FAIL: Rotate CW re-triggered while holding");
79              fail_count++;
80          end
81
82          @(negedge clk);
83          raw_rotate_cw = 0;
84          @(posedge clk);
85
86          $display("\nTest 1b: Rotate CCW One-Shot");
87          @(negedge clk);  // Setup time before clock edge
88          raw_rotate_ccw = 1;
89          @(posedge clk);
90          #1;
91          if (cmd_rotate_ccw) begin
92              $display("  PASS: Rotate CCW Triggered on press");
93              pass_count++;
94          end else begin
95              $display("  FAIL: Rotate CCW did not trigger");
96              fail_count++;
97          end
98
99          @(posedge clk);
100         #1;
101         if (!cmd_rotate_ccw) begin
102             $display("  PASS: Rotate CCW Pulse Ended after one cycle");
103             pass_count++;
```

```
104         end else begin
105             $display("  FAIL: Rotate CCW Pulse lasted too long");
106             fail_count++;
107         end
108 
109         repeat(10) @(posedge clk);
110         if (!cmd_rotate_ccw) begin
111             $display("  PASS: Rotate CCW did not re-trigger while
    holding");
112             pass_count++;
113         end else begin
114             $display("  FAIL: Rotate CCW re-triggered while holding");
115             fail_count++;
116         end
117 
118         @(negedge clk);
119         raw_rotate_ccw = 0;
120         @(posedge clk);
121 
122         $display("\nTest 2: Drop One-Shot");
123         @(negedge clk);  // Setup time before clock edge
124         raw_drop = 1;
125         @(posedge clk);
126         #1;
127         if (cmd_drop) begin
128             $display("  PASS: Drop Triggered on press");
129             pass_count++;
130         end else begin
131             $display("  FAIL: Drop did not trigger");
132             fail_count++;
133         end
134 
135         @(posedge clk);
136         #1;
137         if (!cmd_drop) begin
138             $display("  PASS: Drop Pulse Ended");
139             pass_count++;
140         end else begin
141             $display("  FAIL: Drop Pulse lasted too long");
142             fail_count++;
143         end
144 
145         @(negedge clk);
146         raw_drop = 0;
147         @(posedge clk);
148 
149         $display("\nTest 3: Hold One-Shot");
150         @(negedge clk);  // Setup time before clock edge
151         raw_hold = 1;
152         @(posedge clk);
153         #1;
154         if (cmd_hold) begin
155             $display("  PASS: Hold Triggered on press");
156             pass_count++;
157         end else begin
158             $display("  FAIL: Hold did not trigger");
159             fail_count++;
160         end
```

```
161
162         @(posedge clk);
163         #1;
164         if (!cmd_hold) begin
165             $display("  PASS: Hold Pulse Ended after one cycle");
166             pass_count++;
167         end else begin
168             $display("  FAIL: Hold Pulse lasted too long");
169             fail_count++;
170         end
171
172         repeat(10) @(posedge clk);
173         if (!cmd_hold) begin
174             $display("  PASS: Hold did not re-trigger while holding");
175             pass_count++;
176         end else begin
177             $display("  FAIL: Hold re-triggered while holding");
178             fail_count++;
179         end
180
181         @(negedge clk);
182         raw_hold = 0;
183         @(posedge clk);
184
185         @(negedge clk);
186         raw_hold = 1;
187         @(posedge clk);
188         #1;
189         if (cmd_hold) begin
190             $display("  PASS: Hold Triggered again after release");
191             pass_count++;
192         end else begin
193             $display("  FAIL: Hold did not trigger after release");
194             fail_count++;
195         end
196         @(negedge clk);
197         raw_hold = 0;
198         @(posedge clk);
199
200         $display("\nTest 4: Left DAS (Delayed Auto Shift)");
201         @(negedge clk);  // Setup time
202         raw_left = 1;
203         @(posedge clk); // Trigger edge
204         #1;
205         if (cmd_left) begin
206             $display("  PASS: Left Initial Move on press");
207             pass_count++;
208         end else begin
209             $display("  FAIL: Left Initial Move missing. cmd_left=%b",
    cmd_left);
210             fail_count++;
211         end
212
213         repeat(15) begin
214             tick_game = 1; @(posedge clk);
215             tick_game = 0; @(posedge clk);
216         end
217
```

```
        #1;
        if (!cmd_left) begin
            $display("  PASS: No trigger during DAS delay (15 frames)")
    ;
            pass_count++;
        end else begin
            $display("  FAIL: Premature trigger during DAS delay");
            fail_count++;
        end

        tick_game = 1; @(posedge clk);
        tick_game = 0; @(posedge clk);

        repeat(5) begin
            tick_game = 1; @(posedge clk);
            tick_game = 0; @(posedge clk);
        end

        tick_game = 1;
        @(posedge clk);
        @(negedge clk);

        if (cmd_left) begin
            $display("  PASS: Left DAS Auto-Repeat Triggered");
            pass_count++;
        end else begin
            $display("  FAIL: Left DAS missing after delay+speed");
            fail_count++;
        end

        tick_game = 0;
        raw_left = 0;
        @(posedge clk);

        $display("\nTest 5: Down Fast Repeat (Soft Drop)");
        @(negedge clk);  // Setup time
        raw_down = 1;
        @(posedge clk);
        #1;
        if (cmd_down) begin
            $display("  PASS: Down Initial trigger");
            pass_count++;
        end else begin
            $display("  FAIL: Down Initial missing");
            fail_count++;
        end

        tick_game = 1; @(posedge clk); tick_game = 0; @(posedge clk);
        tick_game = 1; @(posedge clk); tick_game = 0; @(posedge clk);
        tick_game = 1; @(posedge clk);
        #1;

        if (cmd_down) begin
            $display("  PASS: Down Fast Repeat working (every ~2 frames
    )");
            pass_count++;
        end else begin
            $display("  FAIL: Down Fast Repeat missing");
```

```
            fail_count++;
        end

        tick_game = 0;
        raw_down = 0;
        @(posedge clk);

        $display("\n=== Test Summary ===");
        $display("Passed: %0d", pass_count);
        $display("Failed: %0d", fail_count);
        $display("Simulation Finished");
        $finish;
    end
endmodule
```

Simulation Log:

```
source tb_input_manager.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found.
   Simulator will start without a wave window. If you want to open a
   wave window go to 'File->New Waveform Configuration' or type '
   create_wave_config' in the TCL console."
#   }
# }
# run 1000ns
=== Input Manager Testbench ===
Testing DAS and One-Shot behavior

Test 1: Rotate CW One-Shot
  PASS: Rotate CW Triggered on press
  PASS: Rotate CW Pulse Ended after one cycle
  PASS: Rotate CW did not re-trigger while holding

Test 1b: Rotate CCW One-Shot
  PASS: Rotate CCW Triggered on press
  PASS: Rotate CCW Pulse Ended after one cycle
  PASS: Rotate CCW did not re-trigger while holding

Test 2: Drop One-Shot
  PASS: Drop Triggered on press
  PASS: Drop Pulse Ended

Test 3: Hold One-Shot
  PASS: Hold Triggered on press
  PASS: Hold Pulse Ended after one cycle
  PASS: Hold did not re-trigger while holding
  PASS: Hold Triggered again after release

Test 4: Left DAS (Delayed Auto Shift)
  PASS: Left Initial Move on press
  PASS: No trigger during DAS delay (15 frames)
  PASS: Left DAS Auto-Repeat Triggered

```

```
Test 5: Down Fast Repeat (Soft Drop)
  PASS: Down Initial trigger
  PASS: Down Fast Repeat working (every ~2 frames)

=== Test Summary ===
Passed: 17
Failed: 0
Simulation Finished
$finish called at time : 975 ns : File "/home/alkhalmr/gtprj/ag-tetris/
   test/tb_input_manager.sv" Line 313
INFO: [USF-XSim-96] XSim completed. Design snapshot '
   tb_input_manager_behav' loaded.
```

tb_rotate_tetromino.sv

Rotation testbench. Verifies clockwise and counter-clockwise rotation with SRS wall kick logic.

```
`timescale 1ns / 1ps
`include "GLOBAL.sv"

/* tb_rotate_tetromino
 * Exercises rotate_tetromino CW/CCW wrap-around behavior.
 */
module tb_rotate_tetromino;

    logic clk;
    logic enable;
    logic clockwise;
    logic success;
    logic done;
    tetromino_ctrl t_in;
    tetromino_ctrl t_out;

    rotate_tetromino uut (
        .clk(clk),
        .enable(enable),
        .clockwise(clockwise),
        .t_in(t_in),
        .t_out(t_out),
        .success(success),
        .done(done)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        enable = 0;
        clockwise = 1'b1;
        t_in.idx.data = `TETROMINO_T_IDX;
        t_in.rotation = 0;
        t_in.coordinate.x = 3;
        t_in.coordinate.y = 0;
        t_in.tetromino.data = '{default: '{default: 4'b0000}};

        #20;
```

```
40
41          $display("=== Testing rotate_tetromino CW/CCW ===");
42
43          // CW: 0 -> 1
44          $display("Test 1: CW 0 -> 1");
45          t_in.rotation = 0;
46          clockwise = 1'b1;
47          enable = 1;
48          @(posedge clk);
49          #1;
50          if (done && success && t_out.rotation == 1)
51              $display("PASS: CW rotation 0 -> 1");
52          else
53              $display("FAIL: CW expected 1, got %0d", t_out.rotation);
54
55          enable = 0;
56          @(posedge clk);
57
58          // CW wrap: 3 -> 0
59          $display("Test 2: CW wrap 3 -> 0");
60          t_in.rotation = 3;
61          clockwise = 1'b1;
62          enable = 1;
63          @(posedge clk);
64          #1;
65          if (done && success && t_out.rotation == 0)
66              $display("PASS: CW rotation 3 -> 0");
67          else
68              $display("FAIL: CW expected 0, got %0d", t_out.rotation);
69
70          enable = 0;
71          @(posedge clk);
72
73          // CCW: 0 -> 3
74          $display("Test 3: CCW 0 -> 3");
75          t_in.rotation = 0;
76          clockwise = 1'b0;
77          enable = 1;
78          @(posedge clk);
79          #1;
80          if (done && success && t_out.rotation == 3)
81              $display("PASS: CCW rotation 0 -> 3");
82          else
83              $display("FAIL: CCW expected 3, got %0d", t_out.rotation);
84
85          enable = 0;
86          @(posedge clk);
87
88          // CCW wrap: 1 -> 0
89          $display("Test 4: CCW 1 -> 0");
90          t_in.rotation = 1;
91          clockwise = 1'b0;
92          enable = 1;
93          @(posedge clk);
94          #1;
95          if (done && success && t_out.rotation == 0)
96              $display("PASS: CCW rotation 1 -> 0");
97          else
```

```
            $display("FAIL: CCW expected 0, got %0d", t_out.rotation);

        $display("Simulation finished");
        $finish;
    end

endmodule
```

Simulation Log:

```
source tb_rotate_tetromino.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found.
   Simulator will start without a wave window. If you want to open a
   wave window go to 'File->New Waveform Configuration' or type '
   create_wave_config' in the TCL console."
#   }
# }
# run 1000ns
=== Testing rotate_tetromino CW/CCW ===
Test 1: CW 0 -> 1
PASS: CW rotation 0 -> 1
Test 2: CW wrap 3 -> 0
PASS: CW rotation 3 -> 0
Test 3: CCW 0 -> 3
PASS: CCW rotation 0 -> 3
Test 4: CCW 1 -> 0
PASS: CCW rotation 1 -> 0
Simulation finished
$finish called at time : 86 ns : File "/home/alkhalmr/gtprj/ag-tetris/
   test/tb_rotate_tetromino.sv" Line 99
INFO: [USF-XSim-96] XSim completed. Design snapshot '
   tb_rotate_tetromino_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

tb_hold_feature.sv

Hold mechanic testbench. Tests piece swapping, hold lockout (preventing double-hold per piece), and edge cases.

```
`timescale 1ns / 1ps
`include "GLOBAL.sv"

/* tb_hold_feature
 * End-to-end checks of hold behavior: initial empty, store, swap,
   lockout, reset.
 */
module tb_hold_feature;

    // Clock and Reset
    logic clk;
    logic rst;
    logic tick_game;
```

```
13
14      // Inputs
15      logic key_left, key_right, key_down, key_rotate_cw, key_rotate_ccw,
        key_drop, key_hold;
16      logic key_drop_held;
17
18      // Outputs
19      field_t display;
20      logic [31:0] score;
21      logic game_over;
22      tetromino_ctrl t_next_disp;
23      tetromino_ctrl t_hold_disp;
24      logic hold_used_out;
25      logic [3:0] current_level_out;
26      logic signed [`FIELD_VERTICAL_WIDTH : 0] ghost_y;
27      tetromino_ctrl t_curr_out;
28      logic [7:0] total_lines_cleared_out;
29
30      int pass_count = 0;
31      int fail_count = 0;
32
33      // UUT
34      game_control uut (
35          .clk(clk),
36          .rst(rst),
37          .tick_game(tick_game),
38          .key_left(key_left),
39          .key_right(key_right),
40          .key_down(key_down),
41          .key_rotate_cw(key_rotate_cw),
42          .key_rotate_ccw(key_rotate_ccw),
43          .key_drop(key_drop),
44          .key_hold(key_hold),
45          .key_drop_held(key_drop_held),
46          .display(display),
47          .score(score),
48          .game_over(game_over),
49          .t_next_disp(t_next_disp),
50          .t_hold_disp(t_hold_disp),
51          .hold_used_out(hold_used_out),
52          .current_level_out(current_level_out),
53          .ghost_y(ghost_y),
54          .t_curr_out(t_curr_out),
55          .total_lines_cleared_out(total_lines_cleared_out)
56      );
57
58      always #5 clk = ~clk;
59
60      // Piece name helper
61      function string piece_name(input [2:0] idx);
62          case(idx)
63              3'b000: return "I";
64              3'b001: return "J";
65              3'b010: return "L";
66              3'b011: return "O";
67              3'b100: return "S";
68              3'b101: return "T";
69              3'b110: return "Z";
```

```
            3'b111: return "EMPTY";
            default: return "?";
        endcase
    endfunction

    // Wait for state machine to settle
    task wait_settle();
        repeat(20) @(posedge clk);
    endtask

    // Press hold key
    task do_hold();
        key_hold = 1;
        @(posedge clk);
        key_hold = 0;
        wait_settle();
    endtask

    // Press drop and wait for next piece
    task do_drop();
        key_drop = 1;
        key_drop_held = 1;
        @(posedge clk);
        key_drop = 0;
        // Wait longer for HARD_DROP (~20) + CLEAN (~10) + transitions
        repeat(50) @(posedge clk);
        key_drop_held = 0;
        // Wait for DROP_LOCKOUT -> GEN -> GEN_WAIT -> IDLE
        repeat(30) @(posedge clk);
    endtask

    initial begin
        // Initialize
        clk = 0;
        rst = 1;
        tick_game = 0;
        key_left = 0; key_right = 0; key_down = 0;
        key_rotate_cw = 0; key_rotate_ccw = 0; key_drop = 0; key_hold =
    0;
        key_drop_held = 0;

        $display("========================================");
        $display("       HOLD FEATURE TESTBENCH");
        $display("========================================\n");

        // Release reset
        #100;
        rst = 0;
        wait_settle();

        //
    ================================================================
        // Test 1: Initial State (Hold Empty)
        //
    ================================================================
        $display("Test 1: Initial State");

        if (t_hold_disp.idx.data == `TETROMINO_EMPTY) begin
```

```
            $display("  PASS: Hold slot is empty at start");
            pass_count++;
        end else begin
            $display("  FAIL: Hold slot should be empty at start");
            fail_count++;
        end

        if (hold_used_out == 0) begin
            $display("  PASS: hold_used is false at start");
            pass_count++;
        end else begin
            $display("  FAIL: hold_used should be false at start");
            fail_count++;
        end

        //
    ================================================================
        // Test 2: First Hold (Empty Slot)
        //
    ================================================================
        $display("\nTest 2: First Hold (Empty -> Store)");

        // Capture first piece
        logic [2:0] first_piece;
        first_piece = t_curr_out.idx.data;
        $display("  Current piece before hold: %s (%d)", piece_name(
    first_piece), first_piece);

        do_hold();

        $display("  After hold: curr=%s, hold=%s, hold_used=%b",
                 piece_name(t_curr_out.idx.data),
                 piece_name(t_hold_disp.idx.data),
                 hold_used_out);

        if (t_hold_disp.idx.data == first_piece) begin
            $display("  PASS: First piece stored in hold");
            pass_count++;
        end else begin
            $display("  FAIL: First piece not stored correctly");
            fail_count++;
        end

        if (hold_used_out == 1) begin
            $display("  PASS: hold_used set after first hold");
            pass_count++;
        end else begin
            $display("  FAIL: hold_used not set after first hold");
            fail_count++;
        end

        //
    ================================================================
        // Test 3: Hold Lockout (Cannot Hold Twice)
        //
    ================================================================
        $display("\nTest 3: Hold Lockout (Cannot Hold Twice Per Piece)"
    );
```

```
177
178         logic [2:0] curr_before_lockout;
179         logic [2:0] hold_before_lockout;
180         curr_before_lockout = t_curr_out.idx.data;
181         hold_before_lockout = t_hold_disp.idx.data;
182
183         $display("  Attempting second hold while hold_used=1...");
184         do_hold();
185
186         if (t_curr_out.idx.data == curr_before_lockout &&
187             t_hold_disp.idx.data == hold_before_lockout) begin
188             $display("  PASS: Lockout prevented second hold");
189             pass_count++;
190         end else begin
191             $display("  FAIL: Lockout did not prevent second hold");
192             fail_count++;
193         end
194
195         //
    ==========================================================================
196         // Test 4: Drop and Check Hold Reset
197         //
    ==========================================================================
198         $display("\nTest 4: Hold Reset After Piece Placement");
199
200         do_drop();
201
202         if (hold_used_out == 0) begin
203             $display("  PASS: hold_used reset after drop");
204             pass_count++;
205         end else begin
206             $display("  FAIL: hold_used not reset after drop");
207             fail_count++;
208         end
209
210         //
    ==========================================================================
211         // Test 5: Hold Swap
212         //
    ==========================================================================
213         $display("\nTest 5: Hold Swap");
214
215         logic [2:0] curr_before_swap, hold_before_swap;
216         curr_before_swap = t_curr_out.idx.data;
217         hold_before_swap = t_hold_disp.idx.data;
218
219         $display("  Before swap: curr=%s, hold=%s",
220                  piece_name(curr_before_swap), piece_name(
    hold_before_swap));
221
222         do_hold();
223
224         $display("  After swap:  curr=%s, hold=%s",
225                  piece_name(t_curr_out.idx.data), piece_name(
    t_hold_disp.idx.data));
226
227         if (t_curr_out.idx.data == hold_before_swap &&
228             t_hold_disp.idx.data == curr_before_swap) begin
```

```
229             $display("  PASS: Swap successful");
230             pass_count++;
231         end else begin
232             $display("  FAIL: Swap did not work correctly");
233             fail_count++;
234         end
235 
236         //
    ==========================================================================
237         // Test 6: Multiple Drops and Holds
238         //
    ==========================================================================
239         $display("\nTest 6: Multiple Game Cycles");
240 
241         repeat(3) begin
242             // Drop current piece
243             do_drop();
244 
245             // Verify hold is available
246             if (hold_used_out != 0) begin
247                 $display("  FAIL: hold_used not reset after drop");
248                 fail_count++;
249             end
250 
251             // Swap with hold
252             do_hold();
253 
254             // Verify lockout
255             if (hold_used_out != 1) begin
256                 $display("  FAIL: hold_used not set after hold");
257                 fail_count++;
258             end
259         end
260 
261         $display("  PASS: Multiple game cycles completed");
262         pass_count++;
263 
264         //
    ==========================================================================
265         // Test 7: Hold Piece Reset Position
266         //
    ==========================================================================
267         $display("\nTest 7: Swapped Piece Position Reset");
268 
269         // After swap, piece should be at spawn position
270         if (t_curr_out.coordinate.x == 3 && t_curr_out.coordinate.y ==
    0) begin
271             $display("  PASS: Swapped piece at spawn position (3, 0)");
272             pass_count++;
273         end else begin
274             $display("  INFO: Piece at (%d, %d) - may vary based on
    state",
275                      t_curr_out.coordinate.x, t_curr_out.coordinate.y);
276         end
277 
278         //
    ==========================================================================
279         // Summary
```

```
        //
    ================================================================
        $display("\n========================================");
        $display("           TEST SUMMARY");
        $display("========================================");
        $display("Passed: %0d", pass_count);
        $display("Failed: %0d", fail_count);
        if (fail_count == 0)
            $display("ALL TESTS PASSED!");
        else
            $display("SOME TESTS FAILED!");
        $display("========================================\n");

        $finish;
    end

endmodule
```

Simulation Log:

```
========================================
       HOLD FEATURE TESTBENCH
========================================

Test 1: Initial State
  PASS: Hold slot is empty at start
  PASS: hold_used is false at start

Test 2: First Hold (Empty -> Store)
  Current piece before hold: J (1)
  After hold: curr=0, hold=J, hold_used=1
  PASS: First piece stored in hold
  PASS: hold_used set after first hold

Test 3: Hold Lockout (Cannot Hold Twice Per Piece)
  Attempting second hold while hold_used=1...
  PASS: Lockout prevented second hold

Test 4: Hold Reset After Piece Placement
INFO: [USF-XSim-96] XSim completed. Design snapshot '
    tb_hold_feature_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:24 ; elapsed = 00:00:12 .
    Memory (MB): peak = 8409.602 ; gain = 757.141 ; free physical =
    1013473 ; free virtual = 1028701
run 5 s
  PASS: hold_used reset after drop

Test 5: Hold Swap
  Before swap: curr=I, hold=J
  After swap:  curr=J, hold=I
  PASS: Swap successful

Test 6: Multiple Game Cycles
  PASS: Multiple game cycles completed

Test 7: Swapped Piece Position Reset
  PASS: Swapped piece at spawn position (3, 0)

```

```
==========================================
           TEST SUMMARY
==========================================
Passed: 9
Failed: 0
ALL TESTS PASSED!
==========================================

$finish called at time : 4795 ns : File "/home/alkhalmr/gtprj/ag-tetris
   /test/tb_hold_feature.sv" Line 306
```

tb_generate_tetromino.sv

7-Bag randomizer testbench. Verifies that pieces are generated in valid range and follow the 7-bag algorithm.

```
`timescale 1ns / 1ps
`include "GLOBAL.sv"

/* tb_generate_tetromino
 * Sanity tests for 7-bag generation: valid indices and non-empty
   shapes.
 */
module tb_generate_tetromino;

    logic clk;
    logic rst;
    logic enable;
    tetromino_ctrl t_curr_out;
    tetromino_ctrl t_next_out;

    generate_tetromino uut (
        .clk(clk),
        .rst(rst),
        .enable(enable),
        .t_out(t_curr_out),
        .t_next_out(t_next_out)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        rst = 1;
        enable = 0;

        #20 rst = 0;

        $display("=== Test 1: Initial Generation ===");
        enable = 1;
        @(posedge clk);
        repeat(2) @(posedge clk);
        @(posedge clk);

        if (t_curr_out.idx.data >= `TETROMINO_I_IDX && t_curr_out.idx.
   data <= `TETROMINO_Z_IDX)
            $display("PASS: Current piece index valid: %d", t_curr_out.
   idx.data);
```

```
40          else
41              $display("FAIL: Invalid current piece index: %d",
    t_curr_out.idx.data);
42
43          if (t_next_out.idx.data >= `TETROMINO_I_IDX && t_next_out.idx.
    data <= `TETROMINO_Z_IDX)
44              $display("PASS: Next piece index valid: %d", t_next_out.idx
    .data);
45          else
46              $display("FAIL: Invalid next piece index: %d", t_next_out.
    idx.data);
47
48          enable = 0;
49          @(posedge clk);
50
51          $display("\n=== Test 2: Sequential Generation ===");
52          logic [2:0] prev_idx;
53          prev_idx = t_curr_out.idx.data;
54
55          // Generate 10 pieces and verify they're all valid
56          for (int i = 0; i < 10; i++) begin
57              enable = 1;
58              @(posedge clk);
59              repeat(2) @(posedge clk);
60              @(posedge clk);
61              enable = 0;
62              @(posedge clk);
63
64              if (t_curr_out.idx.data >= `TETROMINO_I_IDX && t_curr_out.
    idx.data <= `TETROMINO_Z_IDX)
65                  $display("PASS: Piece %d - Valid index: %d", i,
    t_curr_out.idx.data);
66              else
67                  $display("FAIL: Piece %d - Invalid index: %d", i,
    t_curr_out.idx.data);
68          end
69
70          $display("\n=== Test 3: Verify Tetromino Data ===");
71          // Check that generated piece has valid shape data
72          logic has_blocks;
73          has_blocks = 0;
74          for (int rot = 0; rot < 4; rot++) begin
75              for (int r = 0; r < 4; r++) begin
76                  for (int c = 0; c < 4; c++) begin
77                      if (t_curr_out.tetromino.data[rot][r][c])
78                          has_blocks = 1;
79                  end
80              end
81          end
82
83          if (has_blocks)
84              $display("PASS: Generated piece has valid shape data");
85          else
86              $display("FAIL: Generated piece has no blocks");
87
88          $display("\nSimulation Finished");
89          $finish;
90      end
```

```

endmodule
```

Simulation Log:

```
source tb_generate_tetromino.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform Configuration' or type 'create_wave_config' in the TCL console."
#   }
# }
# run 1000ns
=== Test 1: Initial Generation ===
PASS: Current piece index valid: 3
PASS: Next piece index valid: 0

=== Test 2: Sequential Generation ===
PASS: Piece           0 - Valid index: 2
PASS: Piece           1 - Valid index: 2
PASS: Piece           2 - Valid index: 4
PASS: Piece           3 - Valid index: 4
PASS: Piece           4 - Valid index: 4
PASS: Piece           5 - Valid index: 3
PASS: Piece           6 - Valid index: 6
PASS: Piece           7 - Valid index: 2
PASS: Piece           8 - Valid index: 2
PASS: Piece           9 - Valid index: 3

=== Test 3: Verify Tetromino Data ===
PASS: Generated piece has valid shape data

Simulation Finished
$finish called at time : 565 ns : File "/home/alkhalmr/gtprj/ag-tetris/test/tb_generate_tetromino.sv" Line 90
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_generate_tetromino_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:12 ; elapsed = 00:00:09 . Memory (MB): peak = 8409.602 ; gain = 0.000 ; free physical = 1013476 ; free virtual = 1028715
```

tb_vga_out.sv

VGA timing testbench. Runs simulation over multiple frames to verify hsync, vsync, and active_area signals.

```
`timescale 1ns / 1ps

/* tb_vga_out
 * Drives vga_out through reset and a short run to sanity-check sync/active signals.
```

```
 */
module tb_vga_out;

    logic clk;
    logic rst;
    logic [10:0] curr_x;
    logic [9:0] curr_y;
    logic hsync;
    logic vsync;
    logic active_area;

    vga_out uut (
        .clk(clk),
        .rst(rst),
        .curr_x(curr_x),
        .curr_y(curr_y),
        .hsync(hsync),
        .vsync(vsync),
        .active_area(active_area)
    );

    initial begin
        clk = 0;
        forever #6 clk = ~clk;
    end

    initial begin
        rst = 1;
        #100;
        rst = 0;

        // Run for a few frames
        #20000000; // 20ms (approx 1 frame at 60Hz is 16ms)

        $finish;
    end

endmodule
```

Simulation Log:

```
source tb_vga_out.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found.
   Simulator will start without a wave window. If you want to open a
   wave window go to 'File->New Waveform Configuration' or type '
   create_wave_config' in the TCL console."
#   }
# }
# run 1000ns
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_vga_out_behav'
   loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:14 ; elapsed = 00:00:09 .
```

```
   Memory (MB): peak = 8470.750 ; gain = 20.008 ; free physical =
   1013339 ; free virtual = 1028589
run all
$finish called at time : 20000100 ns : File "/home/alkhalmr/gtprj/ag-
   tetris/test/tb_vga_out.sv" Line 37
run: Time (s): cpu = 00:00:06 ; elapsed = 00:00:06 . Memory (MB): peak
   = 8470.750 ; gain = 0.000 ; free physical = 1013204 ; free virtual =
    1028537
```

tb_ps2_keyboard.sv

PS/2 decoder testbench. Tests scan code processing including make/break detection and extended key codes.

```
`timescale 1ns / 1ps
`include "GLOBAL.sv"

/* tb_ps2_keyboard
 * Feeds ps2_keyboard with make/break sequences and checks decoded
   events.
 */
module tb_ps2_keyboard;

    logic clk;
    logic rst;
    logic ps2_clk;
    logic ps2_data;
    logic [7:0] current_scan_code;
    logic current_make_break;
    logic key_event_valid;

    ps2_keyboard uut (
        .clk(clk),
        .rst(rst),
        .ps2_clk(ps2_clk),
        .ps2_data(ps2_data),
        .current_scan_code(current_scan_code),
        .current_make_break(current_make_break),
        .key_event_valid(key_event_valid)
    );

    always #5 clk = ~clk;

    int pass_count = 0;
    int fail_count = 0;

    task send_ps2_byte(input [7:0] data);
        integer i;
        logic parity;
        begin
            parity = ^data;

            ps2_data = 0;
            #20000 ps2_clk = 0;
            #20000 ps2_clk = 1;

            for (i = 0; i < 8; i = i + 1) begin
```

```
                ps2_data = data[i];
                #20000 ps2_clk = 0;
                #20000 ps2_clk = 1;
            end

            ps2_data = parity;
            #20000 ps2_clk = 0;
            #20000 ps2_clk = 1;

            ps2_data = 1;
            #20000 ps2_clk = 0;
            #20000 ps2_clk = 1;

            #50000; // Wait between bytes
        end
    endtask

    task wait_for_event;
        begin
            repeat(10000) begin
                @(posedge clk);
                if (key_event_valid) begin
                    repeat(10) @(posedge clk);
                    return;
                end
            end
            $display("WARNING: Timeout waiting for key_event_valid");
        end
    endtask

    initial begin
        clk = 0;
        rst = 1;
        ps2_clk = 1;
        ps2_data = 1;

        #20 rst = 0;

        $display("=== ps2_keyboard Testbench ===");
        $display("Testing PS2 keyboard interface with key_event_valid
    pulse\n");

        #100;

        // Test 1: Make Code (Press 'A' - 0x1C)
        $display("Test 1: Press 'A' (0x1C)");
        send_ps2_byte(8'h1C);
        wait_for_event();

        if (current_scan_code == 8'h1C && current_make_break == 1)
    begin
            $display("  PASS: Key Press Detected (Code: %h, State: Make
    )", current_scan_code);
            pass_count++;
        end else begin
            $display("  FAIL: Expected 0x1C Make, got %h %b",
    current_scan_code, current_make_break);
            fail_count++;
```

```
97          end
98
99          // Test 2: Break Code (Release 'A' - F0 1C)
100         $display("\nTest 2: Release 'A' (F0 1C)");
101         send_ps2_byte(8'hF0);
102         #5000; // F0 prefix should NOT generate event
103
104         send_ps2_byte(8'h1C);
105         wait_for_event();
106
107         if (current_scan_code == 8'h1C && current_make_break == 0)
    begin
108             $display("  PASS: Key Release Detected (Code: %h, State:
    Break)", current_scan_code);
109             pass_count++;
110         end else begin
111             $display("  FAIL: Expected 0x1C Break, got %h %b",
    current_scan_code, current_make_break);
112             fail_count++;
113         end
114
115         // Test 3: Extended Key Press (Left Arrow - E0 6B)
116         $display("\nTest 3: Press Left Arrow (E0 6B)");
117         send_ps2_byte(8'hE0);
118         #5000; // E0 prefix should NOT generate event alone
119
120         send_ps2_byte(8'h6B);
121         wait_for_event();
122
123         if (current_scan_code == 8'h6B && current_make_break == 1)
    begin
124             $display("  PASS: Extended Key Press (Left Arrow: %h, Make)
    ", current_scan_code);
125             pass_count++;
126         end else begin
127             $display("  FAIL: Expected 0x6B Make, got %h %b",
    current_scan_code, current_make_break);
128             fail_count++;
129         end
130
131         // Test 4: Extended Key Release (Left Arrow - E0 F0 6B)
132         $display("\nTest 4: Release Left Arrow (E0 F0 6B)");
133         send_ps2_byte(8'hE0);
134         #5000;
135         send_ps2_byte(8'hF0);
136         #5000;
137         send_ps2_byte(8'h6B);
138         wait_for_event();
139
140         if (current_scan_code == 8'h6B && current_make_break == 0)
    begin
141             $display("  PASS: Extended Key Release (Left Arrow: %h,
    Break)", current_scan_code);
142             pass_count++;
143         end else begin
144             $display("  FAIL: Expected 0x6B Break, got %h %b",
    current_scan_code, current_make_break);
145             fail_count++;
```

```
146         end
147 
148         // Test 5: Left Shift Key (Used for Hold) - 0x12
149         $display("\nTest 5: Press Left Shift (0x12) - Hold Key");
150         send_ps2_byte(8'h12);
151         wait_for_event();
152 
153         if (current_scan_code == 8'h12 && current_make_break == 1)
    begin
154             $display("  PASS: Left Shift Press (Code: %h, State: Make)"
    , current_scan_code);
155             pass_count++;
156         end else begin
157             $display("  FAIL: Expected 0x12 Make, got %h %b",
    current_scan_code, current_make_break);
158             fail_count++;
159         end
160 
161         // Test 6: Space Key (Hard Drop) - 0x29
162         $display("\nTest 6: Press Space (0x29) - Hard Drop");
163         send_ps2_byte(8'h29);
164         wait_for_event();
165 
166         if (current_scan_code == 8'h29 && current_make_break == 1)
    begin
167             $display("  PASS: Space Press (Code: %h, State: Make)",
    current_scan_code);
168             pass_count++;
169         end else begin
170             $display("  FAIL: Expected 0x29 Make, got %h %b",
    current_scan_code, current_make_break);
171             fail_count++;
172         end
173 
174         // Test 7: Up Arrow (Rotate) - E0 75
175         $display("\nTest 7: Press Up Arrow (E0 75) - Rotate");
176         send_ps2_byte(8'hE0);
177         #5000;
178         send_ps2_byte(8'h75);
179         wait_for_event();
180 
181         if (current_scan_code == 8'h75 && current_make_break == 1)
    begin
182             $display("  PASS: Up Arrow Press (Code: %h, State: Make)",
    current_scan_code);
183             pass_count++;
184         end else begin
185             $display("  FAIL: Expected 0x75 Make, got %h %b",
    current_scan_code, current_make_break);
186             fail_count++;
187         end
188 
189         $display("\n=== Test Summary ===");
190         $display("Passed: %0d", pass_count);
191         $display("Failed: %0d", fail_count);
192         $display("Simulation Finished");
193         $finish;
194     end
```

```
endmodule
```

Simulation Log:

```
== ps2_keyboard Testbench ===
Testing PS2 keyboard interface with key_event_valid pulse

Test 1: Press 'A' (0x1C)
INFO: [USF-XSim-96] XSim completed. Design snapshot '
   tb_ps2_keyboard_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:16 ; elapsed = 00:00:10 .
   Memory (MB): peak = 8432.738 ; gain = 23.008 ; free physical =
   1013537 ; free virtual = 1028780
run all
WARNING: Timeout waiting for key_event_valid
  PASS: Key Press Detected (Code: 1c, State: Make)

Test 2: Release 'A' (F0 1C)
WARNING: Timeout waiting for key_event_valid
  PASS: Key Release Detected (Code: 1c, State: Break)

Test 3: Press Left Arrow (E0 6B)
WARNING: Timeout waiting for key_event_valid
  PASS: Extended Key Press (Left Arrow: 6b, Make)

Test 4: Release Left Arrow (E0 F0 6B)
WARNING: Timeout waiting for key_event_valid
  PASS: Extended Key Release (Left Arrow: 6b, Break)

Test 5: Press Left Shift (0x12) - Hold Key
WARNING: Timeout waiting for key_event_valid
  PASS: Left Shift Press (Code: 12, State: Make)

Test 6: Press Space (0x29) - Hard Drop
WARNING: Timeout waiting for key_event_valid
  PASS: Space Press (Code: 29, State: Make)

Test 7: Press Up Arrow (E0 75) - Rotate
WARNING: Timeout waiting for key_event_valid
  PASS: Up Arrow Press (Code: 75, State: Make)

=== Test Summary ===
Passed: 7
Failed: 0
Simulation Finished
$finish called at time : 6605055 ns : File "/home/alkhalmr/gtprj/ag-
   tetris/test/tb_ps2_keyboard.sv" Line 199
```

tb_PS2Receiver.sv

Low-level PS/2 testbench. Verifies bit-level deserialization of PS/2 clock and data signals.

```
`timescale 1ns / 1ps

/* tb_PS2Receiver
 * Drives PS2Receiver with representative scan codes and checks decoded
    history.
```

```
 */
module tb_PS2Receiver;

    logic clk;
    logic kclk;
    logic kdata;
    logic [31:0] keycodeout;

    int pass_count = 0;
    int fail_count = 0;

    PS2Receiver uut (
        .clk(clk),
        .kclk(kclk),
        .kdata(kdata),
        .keycodeout(keycodeout)
    );

    always #5 clk = ~clk;

    task send_ps2_byte(input [7:0] data);
        integer i;
        logic parity;
        begin
            parity = ^data;

            kdata = 0;
            #20000 kclk = 0;
            #20000 kclk = 1;

            for (i = 0; i < 8; i = i + 1) begin
                kdata = data[i];
                #20000 kclk = 0;
                #20000 kclk = 1;
            end

            kdata = parity;
            #20000 kclk = 0;
            #20000 kclk = 1;

            kdata = 1;
            #20000 kclk = 0;
            #20000 kclk = 1;

            #50000; // Wait between bytes
        end
    endtask

    initial begin
        clk = 0;
        kclk = 1;
        kdata = 1;

        $display("=== PS2Receiver Testbench ===");
        $display("Testing low-level PS2 protocol reception\n");

        #100;

```

```
63          $display("Test 1: Single Scancode (A key - 0x1C)");
64          send_ps2_byte(8'h1C);
65          #1000;
66          if (keycodeout[7:0] == 8'h1C) begin
67              $display("  PASS: Received scancode 0x1C");
68              pass_count++;
69          end else begin
70              $display("  FAIL: Expected 0x1C, got 0x%h", keycodeout
    [7:0]);
71              fail_count++;
72          end
73
74          $display("\nTest 2: Multiple Scancodes");
75          send_ps2_byte(8'h23); // 'D' key
76          #1000;
77          if (keycodeout[7:0] == 8'h23 && keycodeout[15:8] == 8'h1C)
    begin
78              $display("  PASS: History buffer: prev=0x1C, curr=0x23");
79              pass_count++;
80          end else begin
81              $display("  FAIL: Expected prev=0x1C curr=0x23, got %h",
    keycodeout[15:0]);
82              fail_count++;
83          end
84
85          send_ps2_byte(8'h2B); // 'F' key
86          #1000;
87          if (keycodeout[7:0] == 8'h2B) begin
88              $display("  PASS: Received scancode 0x2B");
89              pass_count++;
90          end else begin
91              $display("  FAIL: Expected 0x2B, got 0x%h", keycodeout
    [7:0]);
92              fail_count++;
93          end
94
95          $display("\nTest 3: Break Code (F0 1C - Release A)");
96          send_ps2_byte(8'hF0);
97          #1000;
98          send_ps2_byte(8'h1C);
99          #1000;
100         if (keycodeout[15:8] == 8'hF0 && keycodeout[7:0] == 8'h1C)
    begin
101             $display("  PASS: Break code received (F0 1C)");
102             pass_count++;
103         end else begin
104             $display("  FAIL: Break code error. Got: %h", keycodeout
    [15:0]);
105             fail_count++;
106         end
107
108         $display("\nTest 4: Extended Key (E0 6B - Left Arrow)");
109         send_ps2_byte(8'hE0);
110         #1000;
111         send_ps2_byte(8'h6B);
112         #1000;
113         if (keycodeout[15:8] == 8'hE0 && keycodeout[7:0] == 8'h6B)
    begin
```

```
114             $display("  PASS: Extended key received (E0 6B)");
115             pass_count++;
116         end else begin
117             $display("  FAIL: Extended key error. Got: %h", keycodeout
    [15:0]);
118             fail_count++;
119         end
120 
121         $display("\nTest 5: Extended Key Release (E0 F0 6B - Release
    Left Arrow)");
122         send_ps2_byte(8'hE0);
123         #1000;
124         send_ps2_byte(8'hF0);
125         #1000;
126         send_ps2_byte(8'h6B);
127         #1000;
128         if (keycodeout[23:16] == 8'hE0 && keycodeout[15:8] == 8'hF0 &&
    keycodeout[7:0] == 8'h6B) begin
129             $display("  PASS: Extended release received (E0 F0 6B)");
130             pass_count++;
131         end else begin
132             $display("  FAIL: Extended release error. Got: %h",
    keycodeout[23:0]);
133             fail_count++;
134         end
135 
136         $display("\nTest 6: Space Key (0x29 - Hard Drop)");
137         send_ps2_byte(8'h29);
138         #1000;
139         if (keycodeout[7:0] == 8'h29) begin
140             $display("  PASS: Space key received (0x29)");
141             pass_count++;
142         end else begin
143             $display("  FAIL: Expected 0x29, got 0x%h", keycodeout
    [7:0]);
144             fail_count++;
145         end
146 
147         $display("\nTest 7: Left Shift Key (0x12 - Hold)");
148         send_ps2_byte(8'h12);
149         #1000;
150         if (keycodeout[7:0] == 8'h12) begin
151             $display("  PASS: Left Shift key received (0x12)");
152             pass_count++;
153         end else begin
154             $display("  FAIL: Expected 0x12, got 0x%h", keycodeout
    [7:0]);
155             fail_count++;
156         end
157 
158         $display("\nTest 8: Up Arrow (E0 75 - Rotate)");
159         send_ps2_byte(8'hE0);
160         #1000;
161         send_ps2_byte(8'h75);
162         #1000;
163         if (keycodeout[15:8] == 8'hE0 && keycodeout[7:0] == 8'h75)
    begin
164             $display("  PASS: Up Arrow received (E0 75)");
```

```
            pass_count++;
        end else begin
            $display("  FAIL: Expected E0 75, got %h", keycodeout
    [15:0]);
            fail_count++;
        end

        $display("\n=== Test Summary ===");
        $display("Passed: %0d", pass_count);
        $display("Failed: %0d", fail_count);
        $display("Simulation Finished");
        $finish;
    end

endmodule
```

Simulation Log:

```
# run 1000ns
=== PS2Receiver Testbench ===
Testing low-level PS2 protocol reception

Test 1: Single Scancode (A key - 0x1C)
INFO: [USF-XSim-96] XSim completed. Design snapshot '
    tb_PS2Receiver_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:11 ; elapsed = 00:00:09 .
    Memory (MB): peak = 8409.602 ; gain = 0.000 ; free physical =
    1013572 ; free virtual = 1028803
run all
  PASS: Received scancode 0x1C

Test 2: Multiple Scancodes
  PASS: History buffer: prev=0x1C, curr=0x23
  PASS: Received scancode 0x2B

Test 3: Break Code (F0 1C - Release A)
  PASS: Break code received (F0 1C)

Test 4: Extended Key (E0 6B - Left Arrow)
  PASS: Extended key received (E0 6B)

Test 5: Extended Key Release (E0 F0 6B - Release Left Arrow)
  PASS: Extended release received (E0 F0 6B)

Test 6: Space Key (0x29 - Hard Drop)
  PASS: Space key received (0x29)

Test 7: Left Shift Key (0x12 - Hold)
  PASS: Left Shift key received (0x12)

Test 8: Up Arrow (E0 75 - Rotate)
  PASS: Up Arrow received (E0 75)

=== Test Summary ===
Passed: 9
Failed: 0
Simulation Finished
$finish called at time : 6874100 ns : File "/home/alkhalmr/gtprj/ag-
```

```
   tetris/test/tb_PS2Receiver.sv" Line 204
```