

# CS256 Class Project Report: FPGA Tetris

Mohammad Alkhalfah

`mohammad.alkhalifah@kaust.edu.sa`

Mustafa Albahrani\*

`mustafa.albaharani@kaust.edu.sa`

December 11, 2025

**Project Repository:** <https://github.com/Mkhlf/ag-tetris>

---

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

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	What is Tetris?	4
1.2	Why Implement Tetris on an FPGA?	4
1.3	Project Overview	4
<b>2</b>	<b>VGA Signaling and Timing Generation</b>	<b>4</b>
2.1	Protocol Overview	5
2.2	Circuit Implementation	5
2.3	Determining Pixel Coordinates	5
<b>3</b>	<b>Module Architecture</b>	<b>6</b>
3.1	Resource Utilization	7
<b>4</b>	<b>Design Description and Drawing Logic</b>	<b>8</b>
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	16
<b>5</b>	<b>Testing and Verification</b>	<b>16</b>
5.1	Testbench Summary	16
5.2	Game Control Simulation ( <code>tb_game_control</code> )	16
5.3	Input Manager Test ( <code>tb_input_manager</code> )	17
5.4	Rotation Test ( <code>tb_rotate_tetromino</code> )	17
5.5	Hold Feature Test ( <code>tb_hold_feature</code> )	18
5.6	Randomizer Test ( <code>tb_generate_tetromino</code> )	18
5.7	Hardware Validation	18
<b>6</b>	<b>References</b>	<b>18</b>
<b>7</b>	<b>Reflection</b>	<b>19</b>
<b>A</b>	<b>Appendix: Verilog Code</b>	<b>20</b>
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  $10 \times 20$  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  $1280 \times 800$  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:

```
1 // README SPEC: hsync 0 when hcount 0-135 (Active Low)
2 assign hsync = ~(hcount <= H_SYNC_END); // H_SYNC_END = 135
3
4 // README SPEC: vsync 1 when vcount 0-2 (Active High)
5 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.

```
1 // Active Area Definition
2 assign active_area = (hcount >= H_VIS_START && hcount <= H_VIS_END) &&
3                     (vcount >= V_VIS_START && vcount <= V_VIS_END);
4
5 // Coordinate Calculation (Relative to Top-Left of Screen)
6 // Output coordinates relative to active area
7 always_comb begin
8     if (active_area) begin
9         curr_x = hcount - H_VIS_START; // Offset by 336
10        curr_y = vcount - V_VIS_START; // Offset by 27
```

```

11    end else begin
12        curr_x = 0; curr_y = 0;
13    end
14 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.

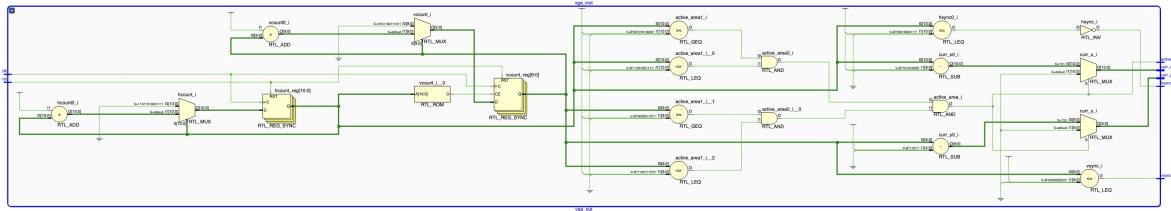


Figure 1: Synthesized Schematic of the ‘`vga_out`’ module showing counters and sync comparators.

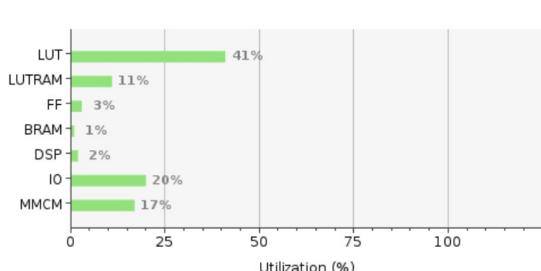
### 3 Module Architecture

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 <code>hsync</code> , <code>vsync</code> , and <code>active_area</code> signals required for the $1280 \times 800$ 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 $10 \times 22$ 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.

```
1 // Calculate Grid Indices
2 s1_grid_col <= (curr_x - GRID_X_START) / BLOCK_SIZE;
3 s1_grid_row <= (curr_y - GRID_Y_START) / BLOCK_SIZE;
4
5 // Check if block exists
6 // The +2 offset maps visible screen coordinates to the internal
7 // board array, which includes 2 hidden spawn rows at the top.
8 if (display.data[s1_grid_row + 2][s1_grid_col].data != 'TETROMINO_EMPTY')
    begin
9     s2_cell_color_idx <= display.data[s1_grid_row + 2][s1_grid_col].
    data + 1;
10 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.

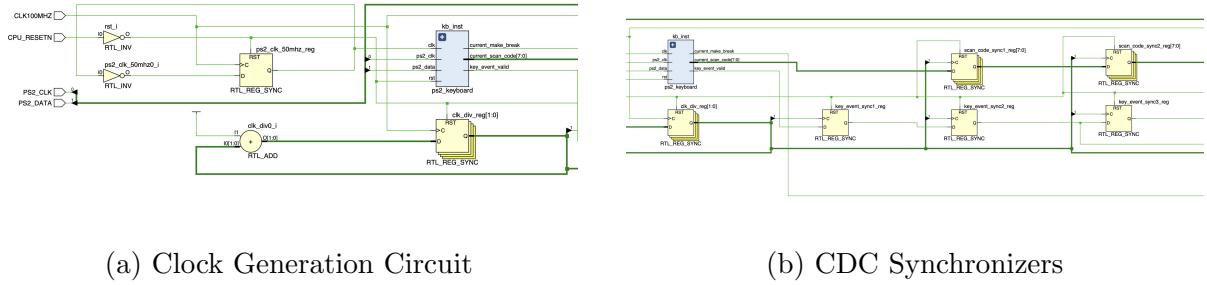


Figure 3: Clock Architecture and Domain Crossing Logic.

#### 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.

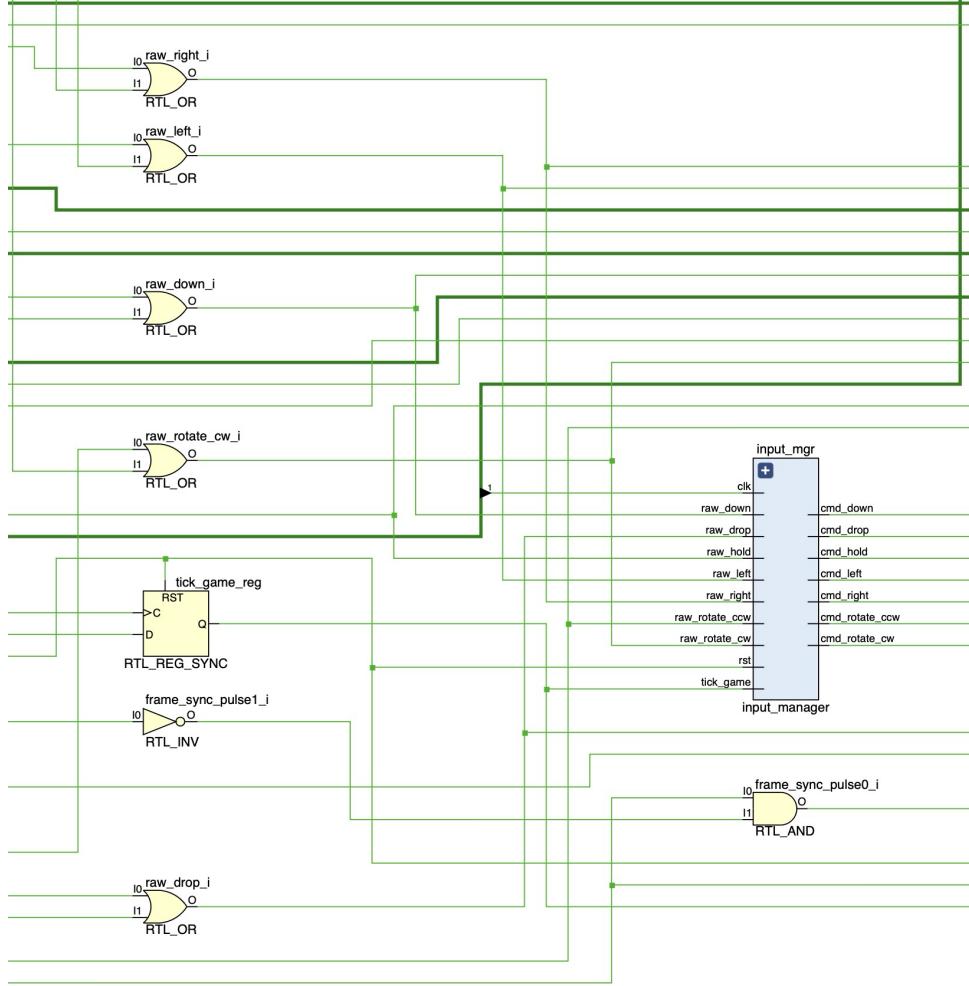
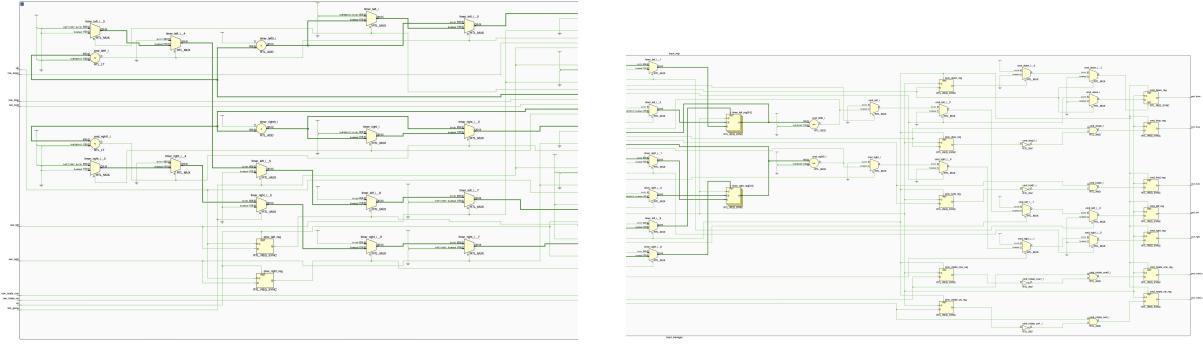


Figure 4: Input merging logic showing OR gates combining keyboard and physical button inputs.

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:  $40 \times (\text{Level} + 1)$
- Double:  $100 \times (\text{Level} + 1)$
- Triple:  $300 \times (\text{Level} + 1)$
- Tetris (4 lines):  $1200 \times (\text{Level} + 1)$

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.

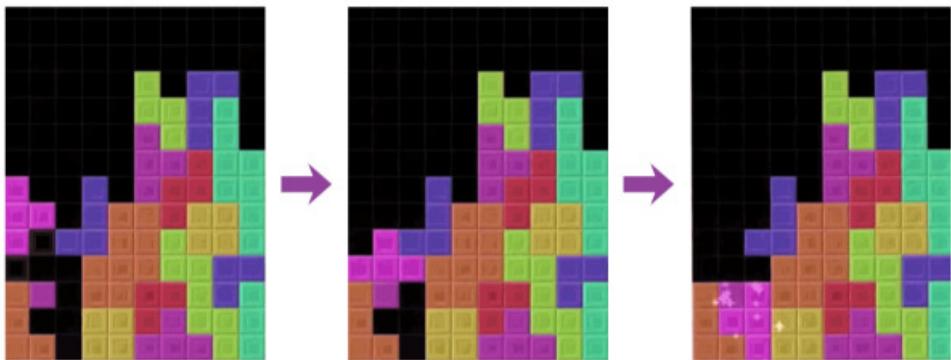


Figure 6: Example of a T-Spin: The T-piece rotates into a slot where it is immobile.

#### 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.

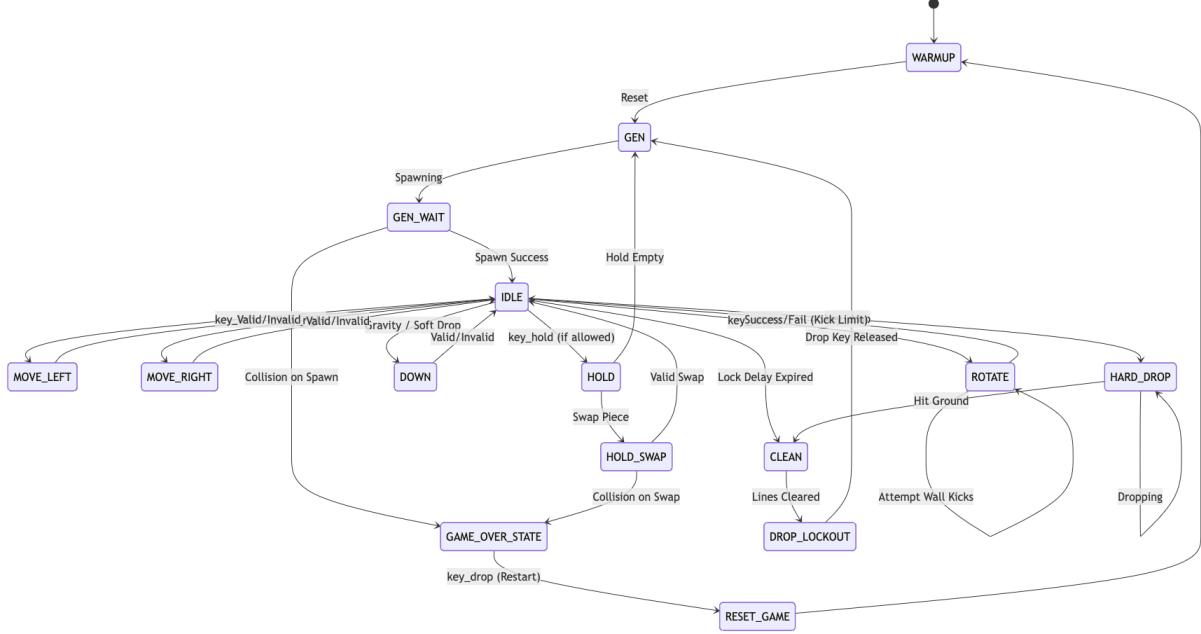


Figure 7: Game Control FSM Diagram showing major state transitions.

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.

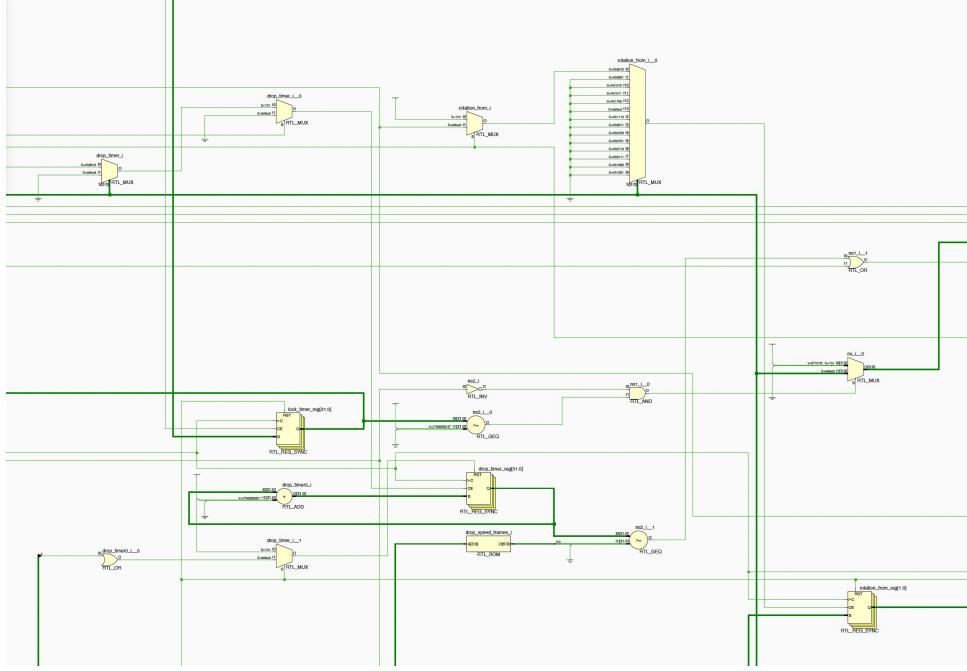


Figure 8: Logic for piece manipulation (Drop and Rotate).

## 4.6 Display Logic

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

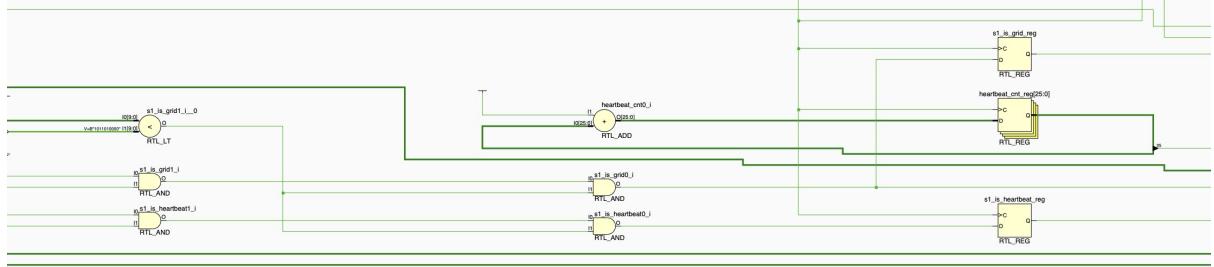
- **CDC for Display:** The game state (field, current piece position) is transferred from the game domain to the pixel domain at the start of each frame (VSync) to ensure a tear-free image.
- **Rendering:** ‘draw\_tetris’ calculates the color of the current pixel.
- **Sprites:** A ROM-based sprite system (‘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.



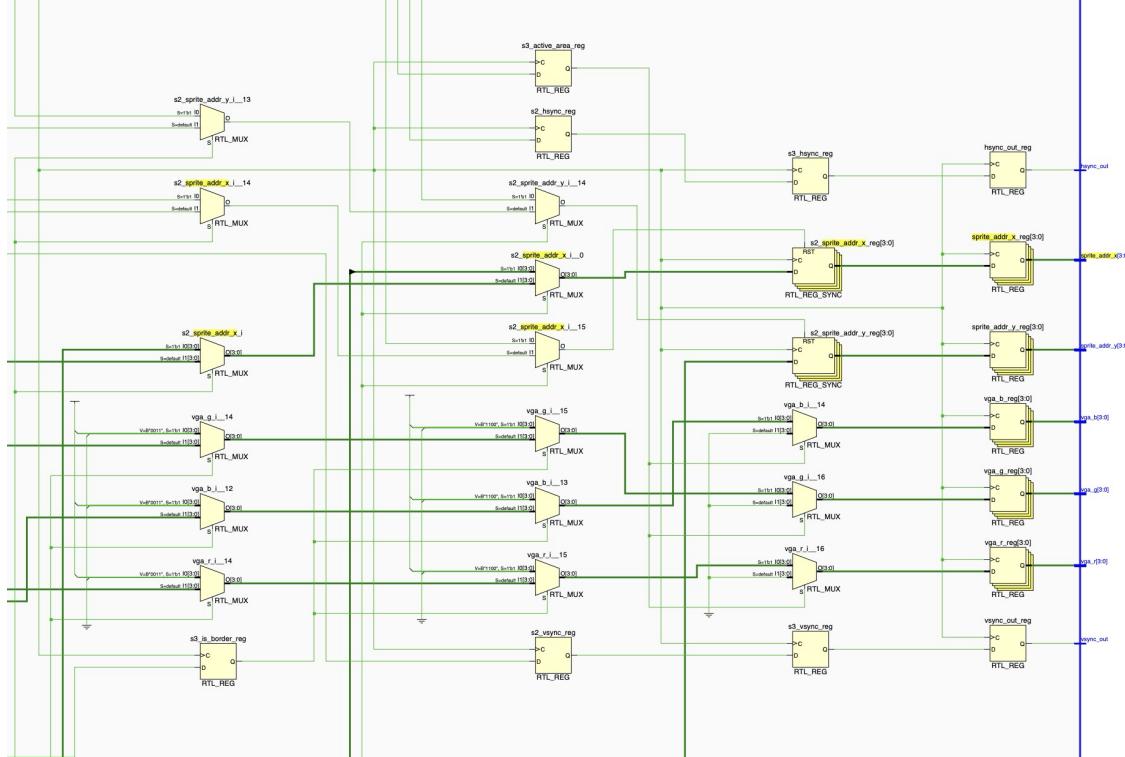
(a) Region Detection Logic (Stage 1)

current pixel coordinates, how you drew objects on screen, and the logic behind your game. You should also discuss how you tested your design. Your report should be written in a formal manner, and include excerpts of code where this would assist discussion. Screen captures may also be useful. Include all your Verilog code in an appendix, clearly stating what each source file is for. Do not paste screenshots of code – if you want syntax highlighting use an online syntax highlighting tool. The report should conclude with a reflective section about what you liked and disliked about the project, what you learned or couldn't understand, and suggestions for improvement in the future.

This report will be marked according to the following criteria:

Criteria	Maximum Marks
Introduction and Background discussion on displays/VGA	3 marks
Design description	5 marks
Testing description	3 marks
Code quality	4 marks
Presentation and references	3 marks
Reflection section	2 marks

The two component marks will be combined to produce the class project mark out of 40.



(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
<code>tb_game_control</code>	Core FSM (IDLE, MOVE, CLEAN)	9/9 PASS
<code>tb_input_manager</code>	DAS, one-shot logic	17/17 PASS
<code>tb_rotate_tetromino</code>	CW/CCW rotation matrix	4/4 PASS
<code>tb_hold_feature</code>	Hold swap, lockout	9/9 PASS
<code>tb_generate_tetromino</code>	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.

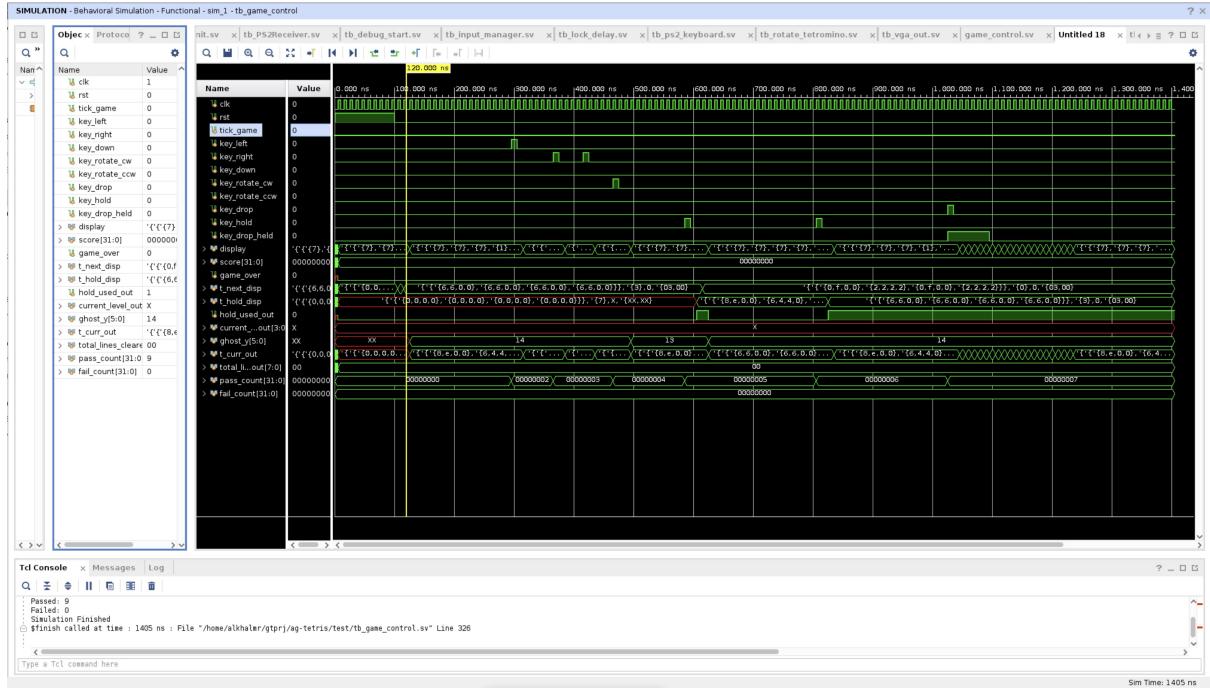


Figure 10: Simulation waveform from `tb_game_control` showing FSM state transitions, piece position updates, and control signals.

### 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 → 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,” *Tetris Wiki*. [Online]. Available: [https://tetris.wiki/Super\\_Rotation\\_System](https://tetris.wiki/Super_Rotation_System)
- [3] “Tetris Guideline,” *Hard Drop Tetris Wiki*. [Online]. Available: [https://harddrop.com/wiki/Tetris\\_Guideline](https://harddrop.com/wiki/Tetris_Guideline)
- [4] “Random Generator,” *Tetris Wiki*. [Online]. Available: [https://tetris.wiki/Random\\_Generator](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.

```
1  'include "src/GLOBAL.sv"
2
3 module game_top(
4     input  wire  logic CLK100MHZ ,
5     input  wire  logic CPU_RESETN , // Active Low
6     input  wire  logic PS2_CLK ,
7     input  wire  logic PS2_DATA ,
8     input  wire  logic btn_l ,
9     input  wire  logic btn_r ,
10    input  wire  logic btn_u ,
11    input  wire  logic btn_d ,
12    input  wire  logic btn_c ,
13    output logic [3:0] VGA_R ,
14    output logic [3:0] VGA_G ,
15    output logic [3:0] VGA_B ,
16    output logic VGA_HS ,
17    output logic VGA_VS ,
18    output logic [1:0] LED ,      // Debug LEDs
19    // 7-Segment Display
20    output logic [6:0] SEG ,      // Segment pattern (active low)
21    output logic [7:0] AN ,       // Anode select (active low)
22    output logic DP           // Decimal point (active low)
23 );
24
25 logic rst;
26 assign rst = ~CPU_RESETN;
27
28 // Debug LEDs - PS/2 Hardware Signals
29 // NOTE: PS/2 uses pull-ups, so lines are HIGH when idle/
30 // disconnected
31 // With keyboard active (typing), you'll see flickering/blinking
32 // LED[0] = PS2_CLK (will flicker when keyboard sends data)
33 // LED[1] = PS2_DATA (will change based on transmitted bits)
34 assign LED[0] = PS2_CLK;
35 assign LED[1] = PS2_DATA;
36
37 // Clock Generation
38 logic pix_clk; // 83.46 MHz (approx)
39 logic locked;
40
41 // Instantiate Clock Wizard
42 clk_wiz_0 clk_gen (
43     .clk_in1(CLK100MHZ),
44     .clk_out1(pix_clk),
```

```

44     .reset(rst),
45     .locked(locked)
46 );
47
48 // PS2 Clock Generation (50 MHz) - Better for PS2 protocol timing
49 // Divide 100MHz by 2
50 logic ps2_clk_50mhz;
51 always_ff @(posedge CLK100MHZ) begin
52   if (rst) ps2_clk_50mhz <= 0;
53   else ps2_clk_50mhz <= ~ps2_clk_50mhz;
54 end
55
56 // Game Clock Generation (25 MHz)
57 // Divide 100MHz by 4
58 logic [1:0] clk_div;
59 logic game_clk;
60
61 always_ff @(posedge CLK100MHZ) begin
62   if (rst) clk_div <= 0;
63   else clk_div <= clk_div + 1;
64 end
65 assign game_clk = clk_div[1]; // 25 MHz
66
67 // Game Tick Generation (60Hz)
68 // 25 MHz / 60 Hz ~= 416,666
69 logic [18:0] tick_counter;
70 logic tick_game;
71
72 always_ff @(posedge game_clk) begin
73   if (rst) begin
74     tick_counter <= 0;
75     tick_game <= 0;
76   end else begin
77     if (tick_counter == 416666) begin
78       tick_counter <= 0;
79       tick_game <= 1;
80     end else begin
81       tick_counter <= tick_counter + 1;
82       tick_game <= 0;
83     end
84   end
85 end
86
87 //=====
88 // PS2 Keyboard Input (50MHz domain)
89 //=====
90
91 logic [7:0] scan_code_50;
92 logic make_break_50;
93 logic key_event_valid_50;
94
95 ps2_keyboard kb_inst (
96   .clk(ps2_clk_50mhz),
97   .rst(rst),
98   .ps2_clk(PS2_CLK),
99

```

```

98     .ps2_data(PS2_DATA),
99     .current_scan_code(scan_code_50),
100    .current_make_break(make_break_50),
101    .key_event_valid(key_event_valid_50)
102  );
103
104  //
=====

105 // CDC: Clock Domain Crossing from 50MHz to 25MHz (game_clk)
106 //
=====

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

112 always_ff @(posedge game_clk) begin
113   if (rst) begin
114     key_event_sync1 <= 0;
115     key_event_sync2 <= 0;
116     key_event_sync3 <= 0;
117   end else begin
118     key_event_sync1 <= key_event_valid_50;
119     key_event_sync2 <= key_event_sync1;
120     key_event_sync3 <= key_event_sync2;
121   end
122 end
123
124 // Detect rising edge in game_clk domain
125 assign key_event_pulse = key_event_sync2 & ~key_event_sync3;
126
127 // Synchronize scan_code and make_break (they're stable during
128 extended pulse)
129 logic [7:0] scan_code_sync1, scan_code_sync2;
130 logic make_break_sync1, make_break_sync2;

131 always_ff @(posedge game_clk) begin
132   if (rst) begin
133     scan_code_sync1 <= 8'h00;
134     scan_code_sync2 <= 8'h00;
135     make_break_sync1 <= 1'b0;
136     make_break_sync2 <= 1'b0;
137   end else begin
138     scan_code_sync1 <= scan_code_50;
139     scan_code_sync2 <= scan_code_sync1;
140     make_break_sync1 <= make_break_50;
141     make_break_sync2 <= make_break_sync1;
142   end
143 end
144
145 // For 7-segment display use
146 logic [7:0] scan_code;
147 logic make_break;
148 assign scan_code = scan_code_sync2;
149 assign make_break = make_break_sync2;

```

```

150
151    //
152    //=====
153    // Decode Raw Levels (Held State) - Now in game_clk domain
154    // WITH WATCHDOG INTEGRATED (single driver!)
155    //
156    //=====
157
158    logic raw_left_kb, raw_right_kb, raw_down_kb, raw_rotate_cw_kb,
159    raw_rotate_ccw_kb, raw_drop_kb, raw_hold_kb;
160
161    // Watchdog counter (declared here, used in same block)
162    logic [19:0] watchdog_counter;
163    logic watchdog_timeout;
164
165    always_ff @(posedge game_clk) begin
166        if (rst) begin
167            raw_left_kb <= 0; raw_right_kb <= 0; raw_down_kb <= 0;
168            raw_rotate_cw_kb <= 0; raw_rotate_ccw_kb <= 0; raw_drop_kb
169            <= 0; raw_hold_kb <= 0;
170            watchdog_counter <= 0;
171            watchdog_timeout <= 0;
172        end else begin
173            // === WATCHDOG LOGIC ===
174            if (raw_left_kb || raw_right_kb || raw_down_kb ||
175                raw_rotate_cw_kb || raw_rotate_ccw_kb || raw_drop_kb ||
176                raw_hold_kb) begin
177                if (watchdog_counter < 20'd750000) begin
178                    watchdog_counter <= watchdog_counter + 1;
179                    watchdog_timeout <= 0;
180                end else begin
181                    watchdog_timeout <= 1;
182                end
183            end else begin
184                watchdog_counter <= 0;
185                watchdog_timeout <= 0;
186            end
187        end
188
189        // === KEYBOARD DECODE (only driver of raw_*_kb signals)
190        ===
191
192        if (watchdog_timeout) begin
193            // Watchdog fired - clear all keys
194            raw_left_kb <= 0;
195            raw_right_kb <= 0;
196            raw_down_kb <= 0;
197            raw_rotate_cw_kb <= 0;
198            raw_rotate_ccw_kb <= 0;
199            raw_drop_kb <= 0;
200            raw_hold_kb <= 0;
201        end else if (key_event_pulse) begin
202            // Normal keyboard events
203            case (scan_code_sync2)
204                'LEFT_ARROW_C: raw_left_kb <= make_break_sync2;
205                'RIGHT_ARROW_C: raw_right_kb <= make_break_sync2;
206                'DOWN_ARROW_C: raw_down_kb <= make_break_sync2;
207                'UP_ARROW_C: raw_rotate_cw_kb <=
208            make_break_sync2;

```

```

199          'X_KEY_C:           raw_rotate_cw_kb <=
200      make_break_sync2;
201          'Z_KEY_C:           raw_rotate_ccw_kb <=
202      make_break_sync2;
203          'SPACE_C:          raw_drop_kb    <= make_break_sync2;
204          'LSHIFT_C:          raw_hold_kb   <= make_break_sync2;
205          default: ; // No change for other keys
206      endcase
207  end
208
209 // Debounce Buttons (use separate debouncer for buttons, different
210 // timing)
211 logic btn_l_db, btn_r_db, btn_u_db, btn_d_db, btn_c_db;
212 logic unused_db;
213
214 // Button debouncer with longer timing for mechanical buttons
215 debouncer_btn db_lr (
216     .clk(game_clk),
217     .I0(btn_l), .I1(btn_r),
218     .O0(btn_l_db), .O1(btn_r_db)
219 );
220
221 debouncer_btn db_ud (
222     .clk(game_clk),
223     .I0(btn_u), .I1(btn_d),
224     .O0(btn_u_db), .O1(btn_d_db)
225 );
226
227 debouncer_btn db_c (
228     .clk(game_clk),
229     .I0(btn_c), .I1(1'b0),
230     .O0(btn_c_db), .O1(unused_db)
231 );
232
233 // Combine with Buttons (Active High)
234 logic raw_left, raw_right, raw_down, raw_rotate_cw, raw_rotate_ccw,
235 raw_drop, raw_hold;
236 assign raw_left = raw_left_kb | btn_l_db;
237 assign raw_right = raw_right_kb | btn_r_db;
238 assign raw_down = raw_down_kb | btn_d_db;
239 assign raw_rotate_cw = raw_rotate_cw_kb | btn_u_db;
240 assign raw_rotate_ccw = raw_rotate_ccw_kb;
241 assign raw_drop = raw_drop_kb | btn_c_db;
242 assign raw_hold = raw_hold_kb; // Hold only via keyboard (no
button)
243
244 // Input Manager (DAS & One-Shot)
245 logic key_left, key_right, key_down, key_rotate_cw, key_rotate_ccw,
246 key_drop, key_hold;
247
248 input_manager input_mgr (
249     .clk(game_clk),
250     .rst(rst),
251     .tick_game(tick_game),
252     .raw_left(raw_left),
253     .raw_right(raw_right),

```

```

251     .raw_down(raw_down),
252     .raw_rotate_cw(raw_rotate_cw),
253     .raw_rotate_ccw(raw_rotate_ccw),
254     .raw_drop(raw_drop),
255     .raw_hold(raw_hold),
256     .cmd_left(key_left),
257     .cmd_right(key_right),
258     .cmd_down(key_down),
259     .cmd_rotate_cw(key_rotate_cw),
260     .cmd_rotate_ccw(key_rotate_ccw),
261     .cmd_drop(key_drop),
262     .cmd_hold(key_hold)
263 );
264
265 // Game Logic
266 field_t display_field;
267 logic [31:0] score;
268 logic game_over;
269 tetromino_ctrl t_next; // Next piece signal
270 tetromino_ctrl t_hold; // Hold piece signal
271 logic hold_used; // Hold was used this piece
272 logic [3:0] current_level;
273 logic signed ['FIELD_VERTICAL_WIDTH : 0] ghost_y;
274 tetromino_ctrl t_curr;
275 logic [7:0] total_lines_cleared; // NEW: For level bar
276
277 game_control game_inst (
278     .clk(game_clk),
279     .rst(rst),
280     .tick_game(tick_game),
281     .key_left(key_left),
282     .key_right(key_right),
283     .key_down(key_down),
284     .key_rotate_cw(key_rotate_cw),
285     .key_rotate_ccw(key_rotate_ccw),
286     .key_drop(key_drop),
287     .key_hold(key_hold),
288     .key_drop_held(raw_drop), // Connect raw state for lockout
289     .display(display_field),
290     .score(score),
291     .game_over(game_over),
292     .t_nextDisp(t_next),
293     .t_holdDisp(t_hold),
294     .holdUsed_out(hold_used),
295     .currentLevel_out(current_level),
296     .ghost_y(ghost_y),
297     .t_curr_out(t_curr),
298     .totalLinesCleared_out(total_lines_cleared)
299 );
300
301 //
302 =====
303 // CDC: Game State Synchronization (game_clk      pix_clk)
304 // Frame-sync to reduce CDC traffic from 83MHz to 60Hz
305 // =====

```

```

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
326             edge of vsync
327         end
328     end
329
330     // Only sync game state once per frame (60 Hz instead of 83.46 MHz)
331     always_ff @(posedge pix_clk) begin
332         if (rst) begin
333             display_field_sync <= '0;
334             score_sync <= 0;
335             game_over_sync <= 0;
336             t_next_sync <= '0;
337             t_hold_sync <= '0;
338             t_curr_sync <= '0;
339             hold_used_sync <= 0;
340             current_level_sync <= 0;
341             total_lines_cleared_sync <= 0;
342             ghost_y_sync <= 0;
343         end else if (frame_sync_pulse) begin // KEY CHANGE: Only
344             update at frame boundary
345             display_field_sync <= display_field;
346             score_sync <= score;
347             game_over_sync <= game_over;
348             t_next_sync <= t_next;
349             t_hold_sync <= t_hold;
350             t_curr_sync <= t_curr;
351             hold_used_sync <= hold_used;
352             current_level_sync <= current_level;
353             total_lines_cleared_sync <= total_lines_cleared;
354             ghost_y_sync <= ghost_y;
355         end
356         // else: Hold previous values (no update)
357     end
358
359     // VGA Output (Raw)
360     logic [10:0] curr_x_raw;
361     logic [9:0] curr_y_raw;
362     logic active_area_raw;

```

```

361 logic hsync_raw, vsync_raw;
362
363 vga_out vga_inst (
364     .clk(pix_clk),
365     .rst(rst),
366     .curr_x(curr_x_raw),
367     .curr_y(curr_y_raw),
368     .hsync(hsync_raw),
369     .vsync(vsync_raw),
370     .active_area(active_area_raw)
371 );
372
373 // Sprite ROM
374 logic [3:0] sprite_addr_x;
375 logic [3:0] sprite_addr_y;
376 logic [11:0] sprite_pixel;
377
378 block_sprite sprite_inst (
379     .clk(pix_clk),
380     .addr_x(sprite_addr_x),
381     .addr_y(sprite_addr_y),
382     .pixel_out(sprite_pixel)
383 );
384
385 // Drawing Logic (Raw)
386 logic [3:0] vga_r_raw, vga_g_raw, vga_b_raw;
387 logic hsync_pipelined, vsync_pipelined;
388
389 draw_tetris draw_inst (
390     .clk(pix_clk),
391     .rst(rst),
392     .curr_x(curr_x_raw),
393     .curr_y(curr_y_raw),
394     .active_area(active_area_raw),
395     .hsync_in(hsync_raw),
396     .vsync_in(vsync_raw),
397
398     .display(display_field_sync),           // Use synchronized
399     .score(score_sync),                  // Use synchronized
400     .game_over(game_over_sync),          // Use synchronized
401     .t_next(t_next_sync),                // Use synchronized
402     .t_hold(t_hold_sync),                // Use synchronized
403     .hold_used(hold_used_sync),          // Use synchronized
404     .current_level(current_level_sync), // Use synchronized
405     .total_lines_cleared(total_lines_cleared_sync), // Use
406     synchronized
407     .ghost_y(ghost_y_sync),               // Use synchronized
408     .t_curr(t_curr_sync),                // Use synchronized
409
410     .sprite_addr_x(sprite_addr_x),
411     .sprite_addr_y(sprite_addr_y),
412     .sprite_pixel(sprite_pixel),
413
414     .vga_r(vga_r_raw),
415     .vga_g(vga_g_raw),
416     .vga_b(vga_b_raw),
417
418     .hsync_out(hsync_pipelined),

```

```

418     .vsync_out(vsync_pipelined)
419 );
420
421 // Output Pipeline (Final Stage)
422 // We keep this stage for clean output timing, effectively making
423 // it a 4-stage pipeline.
424 always_ff @(posedge pix_clk) begin
425     VGA_R <= vga_r_raw;
426     VGA_G <= vga_g_raw;
427     VGA_B <= vga_b_raw;
428     VGA_HS <= hsync_pipelined;
429     VGA_VS <= vsync_pipelined;
430 end
431
432 // =====
433 // 7-Segment Display for Keyboard Input Visualization
434 // =====
435
436 seg7_key_display seg7_inst (
437     .clk(CLK100MHZ),
438     .rst(rst),
439     .scan_code(scan_code),
440     .key_valid(key_event_pulse),
441     .key_left(raw_left),
442     .key_right(raw_right),
443     .key_down(raw_down),
444     .key_rotate_cw(raw_rotate_cw),
445     .key_rotate_ccw(raw_rotate_ccw),
446     .key_drop(raw_drop),
447     .key_hold(raw_hold),
448     .SEG(SEG),
449     .AN(AN),
450     .DP(DP)
451 );
452
453 // =====
454 // Button Debouncer (longer timing for mechanical buttons)
455 // =====
456 module debouncer_btn(
457     input clk,
458     input I0,
459     input I1,
460     output reg 00,
461     output reg 01
462 );
463
464 // Use larger counter for button debouncing (~10ms at 25MHz)
465 reg [17:0] cnt0, cnt1;
466 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 (IO == Iv0) begin
473         if (cnt0 == CNT_MAX)
474             O0 <= IO;
475         else
476             cnt0 <= cnt0 + 1;
477     end else begin
478         cnt0 <= 18'b0;
479         Iv0 <= IO;
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 ( $10 \times 20$ ), 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

```

```

21  'define FIELD_VERTICAL           22 // additional 2 blocks for spawning
22
23  'define FIELD_HORIZONTAL_WIDTH $clog2(`FIELD_HORIZONTAL - 1)
24  'define FIELD_VERTICAL_WIDTH   $clog2(`FIELD_VERTICAL - 1)
25
26
27 // TETROMINO TYPE
28 'define NUMBER_OF_TETROMINO 7
29
30 'define TETROMINO_I_IDX      3'b000
31 'define TETROMINO_J_IDX      3'b001
32 'define TETROMINO_L_IDX      3'b010
33 'define TETROMINO_O_IDX      3'b011
34 'define TETROMINO_S_IDX      3'b100
35 'define TETROMINO_T_IDX      3'b101
36 'define TETROMINO_Z_IDX      3'b110
37 'define TETROMINO_EMPTY     3'b111
38
39 // TETROMINO AND ALL OF ITS ROTATION
40 // [rotation][row][column]
41 typedef struct packed {
42     logic [0:3][0:3][0:3] data;
43 } tetromino_t;
44
45 // KICK OFF
46 // Add this typedef to GLOBAL.sv
47 typedef struct packed {
48     logic signed [2:0] x;
49     logic signed [2:0] y;
50 } kick_offset_t;
51
52 // COLOR TYPE (color_mapper for more detail)
53 'define C_BLACK      4'b0000
54 'define C_FREE1     4'b0001
55 'define C_FREE2     4'b0010
56 'define C_FREE3     4'b0011
57 'define C_FREE4     4'b0100
58 'define C_FREE5     4'b0101
59 'define C_FREE6     4'b0110
60 'define C_FREE7     4'b0111
61 'define COLOR_I    4'b1000
62 'define COLOR_J    4'b1001
63 'define COLOR_L    4'b1010
64 'define COLOR_O    4'b1011
65 'define COLOR_S    4'b1100
66 'define COLOR_T    4'b1101
67 'define COLOR_Z    4'b1110
68 'define C_WHITE     4'b1111
69
70 'define C_BORDER    'C_BLACK
71 'define C_EMPTY     'C_WHITE
72
73 'define C_BORDER_FULL 24'h00_00_00 // black
74 'define C_EMPTY_FULL 24'hb2_b2_b2 // darker grey
75 'define C_BLOCK_BORDER 24'h66_66_66 // lighter grey
76
77 typedef struct packed {
78     logic [3:0] data;

```

```

79     } color_t;
80
81 // TETROMINO COLOR ALWAYS HAS START BIT 1,
82 // CAN BE USED TO IDENTIFY TETROMINO
83 typedef struct packed {
84     logic [2:0] data;
85 } tetromino_idx_t;
86
87 // COORDINATE OF TETROMINO
88 typedef struct packed {
89     logic signed ['FIELD_HORIZONTAL_WIDTH : 0]      x;
90     logic signed ['FIELD_VERTICAL_WIDTH : 0]        y;
91 } coor_t;
92
93 // FIELD DISPLAY
94 typedef struct packed {
95     tetromino_idx_t [0 : 'FIELD_VERTICAL - 1][0 :
96         'FIELD_HORIZONTAL - 1] data;
97 } field_t;
98
99 // CONTROLLING TETROMINO
100 typedef struct packed {
101     tetromino_t          tetromino;
102     tetromino_idx_t      idx;
103     logic [1:0]          rotation; // spawn rotation
104     coor_t               coordinate;
105 } tetromino_ctrl;
106
107 // DRAWING START AND END COORDINATE, INCLUSIVE START, EXCLUSIVE END
108 `define BLOCK_PIXEL 32
109 `define FIELD_START_X 200
110 `define FIELD_END_X ('FIELD_START_X + 'FIELD_HORIZONTAL*'BLOCK_PIXEL)
111
112 `define FIELD_START_Y 40
113 `define FIELD_END_Y ('FIELD_START_Y + 'FIELD_VERTICAL_DISPLAY*
114     'BLOCK_PIXEL)
115
116 `define FIELD_BORDER_THICKNESS 8
117 `define FIELD_BORDER_START_X ('FIELD_START_X - 'FIELD_BORDER_THICKNESS)
118 `define FIELD_BORDER_END_X ('FIELD_END_X + 'FIELD_BORDER_THICKNESS)
119
120 `define FIELD_BORDER_START_Y ('FIELD_START_Y - 'FIELD_BORDER_THICKNESS)
121 `define FIELD_BORDER_END_Y ('FIELD_END_Y + 'FIELD_BORDER_THICKNESS)
122
123 `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.*

```

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

```

```

3 // producing hsync/vsync, active-area flag, and visible coordinates.
4 module vga_out(
5     input wire logic clk,
6     input wire logic rst,
7     output logic [10:0] curr_x, // 0-1279
8     output logic [9:0] curr_y, // 0-799
9     output logic hsync,
10    output logic vsync,
11    output logic active_area // High when in visible area
12 );
13
14 // Counters
15 logic [10:0] hcount;
16 logic [9:0] vcount;
17
18 // Parameters from README
19 localparam H_MAX = 1679;
20 localparam V_MAX = 827;
21
22 localparam H_SYNC_END = 135;
23 localparam V_SYNC_END = 2;
24
25 localparam H_VIS_START = 336;
26 localparam H_VIS_END = 1615;
27 localparam V_VIS_START = 27;
28 localparam V_VIS_END = 826;
29
30 always_ff @(posedge clk) begin
31     if (rst) begin
32         hcount <= 0;
33         vcount <= 0;
34     end else begin
35         if (hcount == H_MAX) begin
36             hcount <= 0;
37             if (vcount == V_MAX) begin
38                 vcount <= 0;
39             end else begin
40                 vcount <= vcount + 1;
41             end
42         end else begin
43             hcount <= hcount + 1;
44         end
45     end
46 end
47
48 // Sync signals
49 // README: hsync 0 when hcount 0-135, else 1. (Active Low)
50 assign hsync = ~(hcount <= H_SYNC_END);
51
52 // README: vsync 1 when vcount 0-2, else 0. (Active High)
53 assign vsync = (vcount <= V_SYNC_END);
54
55 // Active Area
56 assign active_area = (hcount >= H_VIS_START && hcount <= H_VIS_END)
57 && (vcount >= V_VIS_START && vcount <= V_VIS_END)
58 ;

```

```

59 // Current X and Y (relative to visible area)
60 always_comb begin
61     if (active_area) begin
62         curr_x = hcount - H_VIS_START;
63         curr_y = vcount - V_VIS_START;
64     end else begin
65         curr_x = 0;
66         curr_y = 0;
67     end
68 end
69
70 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.*

```

1 // draw_tetris: VGA renderer for the Tetris UI. Pipelines coordinates
2 // through
3 // region detection, text prerendering, block/ghost drawing, and final
4 // color
5 // mapping for the grid, next/hold previews, score, level, and
6 // heartbeat.
7 'include "../GLOBAL.sv"
8
9
10 module draw_tetris(
11     input wire logic clk,
12     input wire logic rst,
13
14     input wire logic [10:0] curr_x,
15     input wire logic [9:0] curr_y,
16     input wire logic active_area,
17     input wire logic hsync_in,
18     input wire logic vsync_in,
19
20     input field_t display,
21     input logic [31:0] score,
22     input logic game_over,
23     input tetromino_ctrl t_next, // Next piece
24     input tetromino_ctrl t_hold, // Hold piece
25     input logic hold_used, // Whether hold was used this
26     piece
27     input logic [3:0] current_level, // Game level
28     input logic [7:0] total_lines_cleared, // NEW: For level bar
29     input logic signed ['FIELD_VERTICAL_WIDTH : 0] ghost_y,
30     input tetromino_ctrl t_curr, // Current piece for ghost rendering
31
32     output logic [3:0] sprite_addr_x,
33     output logic [3:0] sprite_addr_y,
34     input logic [11:0] sprite_pixel,
35
36     // VGA Output
37     output logic [3:0] vga_r,
38     output logic [3:0] vga_g,
39     output logic [3:0] vga_b,
40
41 );
42
43 endmodule

```

```

35     output logic          hsync_out ,
36     output logic          vsync_out
37 );
38
39 // Constants
40 localparam BLOCK_SIZE = 32;
41 localparam GRID_W = `FIELD_HORIZONTAL * BLOCK_SIZE; // 320
42 localparam GRID_H = `FIELD_VERTICAL_DISPLAY * BLOCK_SIZE; // 640
43
44 // Centered Grid
45 localparam GRID_X_START = (1280 - GRID_W) / 2; // 480
46 localparam GRID_Y_START = (800 - GRID_H) / 2; // 80
47
48 // Right Sidebar (Next Piece & Score)
49 localparam SIDE_X_START = GRID_X_START + GRID_W + 50;
50 localparam NEXT_Y_START = GRID_Y_START;
51 localparam SCORE_Y_START = NEXT_Y_START + 200;
52 localparam MESSAGE_Y_START = SCORE_Y_START + 100; // Below score
53 localparam LEVEL_Y_START = MESSAGE_Y_START + 50;
54
55 // Left Sidebar (Hold Piece)
56 localparam HOLD_X_START = GRID_X_START - 200;
57 localparam HOLD_Y_START = GRID_Y_START;
58
59 // Colors - Vibrant scheme for black background
60 logic [11:0] color_map [0:7];
61 initial begin
62     color_map[0] = 12'h000; // Empty (Black)
63     color_map[1] = 12'h0FF; // I - Cyan
64     color_map[2] = 12'h00F; // J - Blue
65     color_map[3] = 12'hF80; // L - Orange
66     color_map[4] = 12'hFF0; // O - Yellow
67     color_map[5] = 12'h0F0; // S - Green
68     color_map[6] = 12'hF0F; // T - Magenta
69     color_map[7] = 12'hF00; // Z - Red
70 end
71
72 //
=====

73 // PIPELINE STAGE 1: Coordinate Calculation & Region Detection
74 //
=====

75
76 // Stage 1 Registers
77 logic s1_active_area;
78 logic s1_hsync, s1_vsync;
79 logic [10:0] s1_curr_x;
80 logic [9:0] s1_curr_y;
81
82 // Regions
83 logic s1_is_grid;
84 logic s1_is_border;
85 logic s1_is_next;
86 logic s1_is_hold;
87 logic s1_is_score;
88 logic s1_is_level;

```

```

89     logic s1_is_heartbeat;
90
91     // Grid Coordinates
92     logic signed [11:0] s1_rel_x, s1_rel_y;
93     logic signed [11:0] s1_grid_col, s1_grid_row;
94     logic [4:0] s1_block_pixel_x, s1_block_pixel_y;
95
96     // Sidebar relative coords
97     logic [10:0] s1_next_rel_x, s1_hold_rel_x;
98     logic [9:0] s1_next_rel_y, s1_hold_rel_y;
99
100    always_ff @(posedge clk) begin
101        // Pass-through control signals
102        s1_active_area <= active_area;
103        s1_hsync <= hsync_in;
104        s1_vsync <= vsync_in;
105        s1_curr_x <= curr_x;
106        s1_curr_y <= curr_y;
107
108        // Calculate Relative Coordinates
109        s1_rel_x <= curr_x - GRID_X_START;
110        s1_rel_y <= curr_y - GRID_Y_START;
111
112        // Calculate Grid Indices
113        s1_grid_col <= (curr_x - GRID_X_START) / BLOCK_SIZE;
114        s1_grid_row <= (curr_y - GRID_Y_START) / BLOCK_SIZE;
115        s1_block_pixel_x <= (curr_x - GRID_X_START) % BLOCK_SIZE;
116        s1_block_pixel_y <= (curr_y - GRID_Y_START) % BLOCK_SIZE;
117
118        // Region Detection
119        s1_is_grid <= (curr_x >= GRID_X_START && curr_x < GRID_X_START
+ GRID_W &&
120                                curr_y >= GRID_Y_START && curr_y < GRID_Y_START
+ GRID_H);
121
122        s1_is_border <= (curr_x >= GRID_X_START - 4 && curr_x <
GRID_X_START + GRID_W + 4 &&
123                                curr_y >= GRID_Y_START - 4 && curr_y <
GRID_Y_START + GRID_H + 4) &&
124                                !(curr_x >= GRID_X_START && curr_x <
GRID_X_START + GRID_W &&
125                                curr_y >= GRID_Y_START && curr_y <
GRID_Y_START + GRID_H);
126
127        s1_is_next <= (curr_x >= SIDE_X_START && curr_x < SIDE_X_START
+ 150 &&
128                                curr_y >= NEXT_Y_START && curr_y < NEXT_Y_START
+ 150);
129
130        s1_is_hold <= (curr_x >= HOLD_X_START && curr_x < HOLD_X_START
+ 150 &&
131                                curr_y >= HOLD_Y_START && curr_y < HOLD_Y_START
+ 150);
132
133        s1_is_score <= (curr_x >= SIDE_X_START && curr_x < SIDE_X_START
+ 200 &&
134                                curr_y >= SCORE_Y_START && curr_y <
SCORE_Y_START + 100);

```

```

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

```

```

184     logic render_level_pixel;
185
186     // Text module instantiations (for rendering - NOT for display)
187     draw_number score_renderer (
188         .curr_x(render_curr_x),
189         .curr_y(render_curr_y),
190         .pos_x(11'd0), // Relative to bitmap origin
191         .pos_y(10'd0),
192         .number(score),
193         .pixel_on(render_score_pixel)
194     );
195
196     logic [7:0] render_level_chars [0:15];
197     logic [3:0] render_level_len;
198
199     always_comb begin
200         render_level_chars[0] = 8'h4C; // L
201         render_level_chars[1] = 8'h45; // E
202         render_level_chars[2] = 8'h56; // V
203         render_level_chars[3] = 8'h45; // E
204         render_level_chars[4] = 8'h4C; // L
205         render_level_chars[5] = 8'h3A; // :
206         render_level_chars[6] = 8'h20; // Space
207         render_level_chars[7] = 8'h20; // Space (default)
208         render_level_chars[8] = 8'h20; // Space (default)
209         render_level_chars[9] = 8'h20;
210         render_level_chars[10] = 8'h20;
211         render_level_chars[11] = 8'h20;
212         render_level_chars[12] = 8'h20;
213         render_level_chars[13] = 8'h20;
214         render_level_chars[14] = 8'h20;
215         render_level_chars[15] = 8'h20;
216
217         if (current_level < 10) begin
218             render_level_chars[7] = 8'h30 + current_level;
219             render_level_len = 8;
220         end else begin
221             render_level_chars[7] = 8'h31;
222             render_level_chars[8] = 8'h30 + (current_level - 10);
223             render_level_len = 9;
224         end
225
226     end
227
228     draw_string_line level_renderer (
229         .curr_x(render_curr_x),
230         .curr_y(render_curr_y),
231         .pos_x(11'd0),
232         .pos_y(10'd0),
233         .str_chars(render_level_chars),
234         .str_len(render_level_len),
235         .scale(2'd2),
236         .pixel_on(render_level_pixel)
237     );
238
239     // State machine to render text to bitmaps once per frame (
240     PIPELINED)
241     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
262             // NEXT cycle
263             // (Text modules process combinationally, result
264             // available next clock)
265
266             // Store the result from PREVIOUS cycle
267             if (tr_x_counter > 0 || tr_y_counter > 0) begin
268                 score_text_bitmap[tr_y_counter][tr_x_counter] <=
269                 render_score_pixel;
270             end
271
272             // Advance to next pixel and set coordinates for NEXT
273             // cycle
274             if (tr_x_counter < 199) begin
275                 tr_x_counter <= tr_x_counter + 1;
276                 render_curr_x <= tr_x_counter + 1;
277                 render_curr_y <= tr_y_counter;
278             end else begin
279                 tr_x_counter <= 0;
280                 render_curr_x <= 0;
281                 if (tr_y_counter < 59) begin
282                     tr_y_counter <= tr_y_counter + 1;
283                     render_curr_y <= tr_y_counter + 1;
284                 end else begin
285                     // Done with score, move to level
286                     tr_state <= TR_LEVEL;
287                     tr_y_counter <= 0;
288                     tr_x_counter <= 0;
289                     render_curr_x <= 0;
290                     render_curr_y <= 0;
291                 end
292             end
293         end
294
295         TR_LEVEL: begin
296             // Store result from previous cycle
297             if (tr_x_counter > 0 || tr_y_counter > 0) begin

```

```

294         level_text_bitmap[tr_y_counter][tr_x_counter] <=
295     render_level_pixel;
296     end
297
298     // Advance and set coordinates for next cycle
299     if (tr_x_counter < 199) begin
300         tr_x_counter <= tr_x_counter + 1;
301         render_curr_x <= tr_x_counter + 1;
302         render_curr_y <= tr_y_counter;
303     end else begin
304         tr_x_counter <= 0;
305         render_curr_x <= 0;
306         if (tr_y_counter < 19) begin
307             tr_y_counter <= tr_y_counter + 1;
308             render_curr_y <= tr_y_counter + 1;
309         end else begin
310             tr_state <= TR_DONE;
311         end
312     end
313
314     TR_DONE: begin
315         tr_state <= TR_IDLE; // Wait for next frame
316     end
317     endcase
318 end
319 end
320
321 //
322 =====
323 // FAST TEXT LOOKUP (replaces Stage 1.5, 1.75 entirely)
324 //
325 =====
326
327 // Fast bitmap lookup (just 1 cycle - no 32-level logic!)
328 logic s1b_score_pixel_on;
329 logic s1b_level_text_pixel_on;
330
331 always_ff @(posedge clk) begin
332     s1b_score_pixel_on <= 0;
333     s1b_level_text_pixel_on <= 0;
334
335     // Score lookup
336     if (s1_is_score && s1_curr_y >= SCORE_Y_START + 40 && s1_curr_y
337     < SCORE_Y_START + 100) begin
338         automatic int bmp_y = s1_curr_y - (SCORE_Y_START + 40);
339         automatic int bmp_x = s1_curr_x - SIDE_X_START;
340         if (bmp_x >= 0 && bmp_x < 200 && bmp_y >= 0 && bmp_y < 60)
341     begin
342         s1b_score_pixel_on <= score_text_bitmap[bmp_y][bmp_x];
343     end
344     end
345
346     // Level lookup

```

```

344         if (s1_is_level && s1_curr_y >= LEVEL_Y_START && s1_curr_y <
345             LEVEL_Y_START + 20) begin
346             automatic int bmp_y = s1_curr_y - LEVEL_Y_START;
347             automatic int bmp_x = s1_curr_x - SIDE_X_START;
348             if (bmp_x >= 0 && bmp_x < 200 && bmp_y >= 0 && bmp_y < 20)
349             begin
350                 s1b_level_text_pixel_on <= level_text_bitmap[bmp_y][
351                 bmp_x];
352             end
353         end
354     // -----
355     // PIPELINE STAGE 2: Data Access & Logic
356     // -----
357
358     // Stage 2 Registers
359     logic s2_active_area;
360     logic s2_hsync, s2_vsync;
361
362     // Region Flags
363     logic s2_is_border;
364     logic s2_is_grid;
365     logic s2_is_next;
366     logic s2_is_hold;
367     logic s2_is_score;
368     logic s2_is_level;
369     logic s2_is_heartbeat;
370     logic s2_is_grid_line;
371     logic s2_is_ghost;
372
373     // Visual Data
374     logic [3:0] s2_cell_color_idx;
375     logic [3:0] s2_sprite_addr_x;
376     logic [3:0] s2_sprite_addr_y;
377
378     // Text/UI Pixels
379     logic s2_score_pixel;
380     logic s2_level_text_pixel;
381     logic s2_level_bar_pixel;
382     logic [3:0] s2_level_bar_color_idx;
383     logic s2_heartbeat_pixel;
384     logic s2_level_bar_border;
385
386     // Headers
387     logic s2_next_header;
388     logic s2_hold_header;
389     logic s2_score_header;
390     logic s2_level_header;
391     logic s2_hold_used_header;
392
393     // Heartbeat Counter
394     logic [25:0] heartbeat_cnt;
395     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 <
453         'FIELD_VERTICAL_DISPLAY) begin
454
455         // Main Block
456         if (display.data[s1_grid_row + 2][s1_grid_col].data != 'TETROMINO_EMPTY) begin
457             s2_cell_color_idx <= display.data[s1_grid_row + 2][
458                 s1_grid_col].data + 1;
459             s2_sprite_addr_x <= s1_block_pixel_x[4:1];
460             s2_sprite_addr_y <= s1_block_pixel_y[4:1];
461         end
462         // Ghost Piece
463         else if (!game_over) begin
464             if ((s1_grid_row + 2) >= ghost_y && (s1_grid_row +
465                 2) < ghost_y + 4 &&
466                 s1_grid_col >= t_curr.coordinate.x &&
467                 s1_grid_col < t_curr.coordinate.x + 4) begin
468
469                 if (t_curr.tetromino.data[t_curr.rotation][
470                     s1_grid_row + 2 - ghost_y][s1_grid_col - t_curr.coordinate.x]) begin
471                     s2_is_ghost <= 1;
472                     s2_sprite_addr_x <= s1_block_pixel_x[4:1];
473                     s2_sprite_addr_y <= s1_block_pixel_y[4:1];
474                 end
475             end
476         end
477     end
478
479     // 2. Next Piece Logic
480     else if (s1_is_next) begin
481         if (s1_next_rel_y < 20) begin
482             s2_next_header <= 1;
483         end else begin
484             // Draw Piece
485             automatic logic [10:0] nx = s1_next_rel_x - 20;
486             automatic logic [9:0] ny = s1_next_rel_y - 40;
487             automatic logic [2:0] nr = ny / BLOCK_SIZE;
488             automatic logic [2:0] nc = nx / BLOCK_SIZE;
489
490             if (nx < 128 && ny < 128 && nr < 4 && nc < 4) begin
491                 if (t_next.tetromino.data[0][nr][nc]) begin
492                     s2_cell_color_idx <= t_next.idx.data + 1;
493                     s2_sprite_addr_x <= (nx % BLOCK_SIZE) >> 1;
494                     s2_sprite_addr_y <= (ny % BLOCK_SIZE) >> 1;
495                 end
496             end
497         end
498     end
499
500     // 3. Hold Piece Logic
501     else if (s1_is_hold) begin
502         if (s1_hold_rel_y < 20) begin
503             s2_hold_header <= 1;
504             s2_hold_used_header <= hold_used;

```

```

501         end else begin
502             if (t_hold.idx.data != 'TETROMINO_EMPTY) begin
503                 automatic logic [10:0] hx = s1_hold_rel_x - 20;
504                 automatic logic [9:0] hy = s1_hold_rel_y - 40;
505                 automatic logic [2:0] hr = hy / BLOCK_SIZE;
506                 automatic logic [2:0] hc = hx / BLOCK_SIZE;
507
508                 if (hx < 128 && hy < 128 && hr < 4 && hc < 4) begin
509                     if (t_hold.tetromino.data[0][hr][hc]) begin
510                         s2_cell_color_idx <= t_hold.idx.data + 1;
511                         s2_sprite_addr_x <= (hx % BLOCK_SIZE) >> 1;
512                         s2_sprite_addr_y <= (hy % BLOCK_SIZE) >> 1;
513                     end
514                 end
515             end
516         end
517     end
518
519     // 4. Score Logic
520     else if (s1_is_score) begin
521         if (s1_curr_y < SCORE_Y_START + 20) begin
522             s2_score_header <= 1;
523         end
524         s2_score_pixel <= s1b_score_pixel_on; // NEW: From
525 buffered register
526     end
527
528     // 5. Level Logic
529     else if (s1_is_level) begin
530         if (s1_curr_y < LEVEL_Y_START + 20) begin
531             s2_level_header <= 1;
532         end
533         s2_level_text_pixel <= s1b_level_text_pixel_on; // NEW
534 : From buffered registeron;
535
536         // // Level Bar Border (2px wide white border)
537         // if (s1_curr_x >= SIDE_X_START - 2 && s1_curr_x <
538 SIDE_X_START + 202 &&
539             // s1_curr_y >= LEVEL_Y_START + 38 && s1_curr_y <
540 LEVEL_Y_START + 62) begin
541
542             // if (s1_curr_x < SIDE_X_START || s1_curr_x >=
543 SIDE_X_START + 200 ||
544                 // s1_curr_y < LEVEL_Y_START + 40 || s1_curr_y >=
545 LEVEL_Y_START + 60) begin
546                 // s2_level_bar_border <= 1;
547                 // end
548             // end
549
550             // Level Bar
551             if (s1_curr_x >= SIDE_X_START && s1_curr_x < SIDE_X_START +
552 ((total_lines_cleared % 10) * 20) &&
553                 s1_curr_y >= LEVEL_Y_START + 40 && s1_curr_y <
554 LEVEL_Y_START + 60) begin
555                 s2_level_bar_pixel <= 1;
556                 if (current_level < 5) s2_level_bar_color_idx <= 0;

```

```

550         else if (current_level < 10) s2_level_bar_color_idx <=
551             1;
552             else s2_level_bar_color_idx <= 2;
553         end
554     end
555
556     // 6. Heartbeat
557     if (s1_is_heartbeat && heartbeat_cnt[25] && game_over) begin
558         s2_heartbeat_pixel <= 1;
559     end
560 end
561
562 // =====
563 // PIPELINE STAGE 3: Sprite Lookup Alignment
564 // =====
565
566 // Stage 3 intermediate for sprite lookup
567 logic [3:0] s3_sprite_addr_x;
568 logic [3:0] s3_sprite_addr_y;
569
570 // Output sprite address for ROM lookup (1 cycle before color
571 // mapping)
572 always_ff @(posedge clk) begin
573     sprite_addr_x <= s2_sprite_addr_x;
574     sprite_addr_y <= s2_sprite_addr_y;
575
576     // Also pass these to next stage for alignment
577     s3_sprite_addr_x <= s2_sprite_addr_x;
578     s3_sprite_addr_y <= s2_sprite_addr_y;
579 end
580
581 // Stage 3 Registers (aligned with sprite_pixel output)
582 logic s3_active_area;
583 logic s3_hsync, s3_vsync;
584 logic s3_is_border;
585 logic s3_is_grid;
586 logic s3_is_grid_line;
587 logic s3_is_ghost;
588 logic s3_is_next;
589 logic s3_is_hold;
590 logic [3:0] s3_cell_color_idx;
591 logic s3_next_header;
592 logic s3_hold_header;
593 logic s3_hold_used_header;
594 logic s3_score_header;
595 logic s3_score_pixel;
596 logic s3_level_header;
597 logic s3_level_text_pixel;
598 logic s3_level_bar_pixel;
599 logic [3:0] s3_level_bar_color_idx;
600 logic s3_heartbeat_pixel;
601 logic s3_level_bar_border;

602 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
625
626    //
627    =====
628
629
630    // PIPELINE STAGE 4: Color Mapping & Final Output
631    //
632    =====
633
634
635    always_ff @(posedge clk) begin
636        hsync_out <= s3_hsync;
637        vsync_out <= s3_vsync;
638
639        // Default: BLACK background
640        vga_r <= 4'h0;
641        vga_g <= 4'h0;
642        vga_b <= 4'h0;
643
644        if (s3_active_area) begin
645            // 1. Grid Border - White
646            if (s3_is_border) begin
647                vga_r <= 4'hC; vga_g <= 4'hC; vga_b <= 4'hC;
648            end
649
650            // 2. Grid Lines - Dark Grey
651            else if (s3_is_grid_line) begin
652                vga_r <= 4'h3; vga_g <= 4'h3; vga_b <= 4'h3;
653            end
654
655            // 3. Grid Content (Blocks and Ghost)
656            else if (s3_is_grid && (s3_cell_color_idx != 0 ||
657            s3_is_ghost)) begin
658                automatic logic [3:0] r, g, b;
659                automatic logic [3:0] inten = sprite_pixel[7:4];

```

```

655     if (s3_is_ghost) begin
656         // Ghost piece - Semi-transparent white
657         r = 4'h5; g = 4'h5; b = 4'h5;
658     end else begin
659         // Normal block rendering
660         if (game_over) begin
661             // Grey out blocks when game over
662             r = 4'h4; g = 4'h4; b = 4'h4;
663         end else begin
664             r = color_map[s3_cell_color_idx][11:8];
665             g = color_map[s3_cell_color_idx][7:4];
666             b = color_map[s3_cell_color_idx][3:0];
667         end
668     end
669
670     // Apply sprite shading
671     if (inten != 4'hF) begin
672         vga_r <= r >> 1;
673         vga_g <= g >> 1;
674         vga_b <= b >> 1;
675     end else begin
676         vga_r <= r;
677         vga_g <= g;
678         vga_b <= b;
679     end
680 end
681
682     // 4. Next Piece Header - Bright White
683     else if (s3_next_header) begin
684         vga_r <= 4'hF; vga_g <= 4'hF; vga_b <= 4'hF;
685     end
686     // Next Piece Blocks
687     else if (s3_is_next && s3_cell_color_idx != 0) begin
688         automatic logic [3:0] r, g, b;
689         automatic logic [3:0] inten = sprite_pixel[7:4];
690
691         r = color_map[s3_cell_color_idx][11:8];
692         g = color_map[s3_cell_color_idx][7:4];
693         b = color_map[s3_cell_color_idx][3:0];
694
695         if (inten != 4'hF) begin
696             vga_r <= r >> 1; vga_g <= g >> 1; vga_b <= b >> 1;
697         end else begin
698             vga_r <= r; vga_g <= g; vga_b <= b;
699         end
700     end
701
702     // 5. Hold Piece Header
703     else if (s3_hold_header) begin
704         if (s3_hold_used_header) begin
705             // Grayed out when used
706             vga_r <= 4'h5; vga_g <= 4'h5; vga_b <= 4'h5;
707         end else begin
708             // Bright cyan when available
709             vga_r <= 4'h0; vga_g <= 4'hE; vga_b <= 4'hE;
710         end
711     end
712     // 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'
754 h0; end // Bright Green (easy)
755                 1: begin vga_r <= 4'hF; vga_g <= 4'hC; vga_b <= 4'
756 h0; end // Orange (medium)
757                 2: begin vga_r <= 4'hF; vga_g <= 4'h0; vga_b <= 4'
758 h0; end // Red (hard)
759                 default: begin vga_r <= 4'h0; vga_g <= 4'hE; vga_b
760 <= 4'h0; end
761             endcase
762         end
763
764         // 8. Heartbeat - Blinking white when game over
765         else if (s3_heartbeat_pixel) begin
766             vga_r <= 4'hF; vga_g <= 4'hF; vga_b <= 4'hF;

```

```

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,
3 // returning
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
2 // origin using
3 // a compact 4x5 bitmap font scaled for display overlays.
4 'include "../GLOBAL.sv"
5 module draw_number (
6     input logic [10:0] curr_x,
7     input logic [9:0] curr_y,
8     input logic [10:0] pos_x,
9     input logic [9:0] pos_y,
10    input logic [31:0] number,
11    output logic      pixel_on
```

```

11 );
12     logic [19:0] current_glyph;
13
14     always_comb begin
15         case (digit)
16             0: current_glyph = 20'b0110_1001_1001_1001_0110;
17             1: current_glyph = 20'b0010_0010_0010_0010_0010;
18             2: current_glyph = 20'b0110_0001_0010_0100_1111;
19             3: current_glyph = 20'b0110_0001_0010_0001_0110;
20             4: current_glyph = 20'b1001_1001_1111_0001_0001;
21             5: current_glyph = 20'b1111_1000_1110_0001_1110;
22             6: current_glyph = 20'b0110_1000_1110_1001_0110;
23             7: current_glyph = 20'b1111_0001_0010_0100_0100;
24             8: current_glyph = 20'b0110_1001_0110_1001_0110;
25             9: current_glyph = 20'b0110_1001_0111_0001_0110;
26             default: current_glyph = '0;
27         endcase
28     end
29
30     logic [3:0] digit;
31     logic [2:0] dx, dy; // 0-4
32     logic [3:0] digit_idx; // 0-7
33
34     localparam SCALE = 4;
35     localparam DIGIT_W = 4 * SCALE;
36     localparam DIGIT_H = 5 * SCALE;
37     localparam SPACING = 1 * SCALE;
38     localparam TOTAL_W = DIGIT_W + SPACING;
39
40     logic [10:0] rel_x;
41     logic [9:0] rel_y;
42
43     assign rel_x = curr_x - pos_x;
44     assign rel_y = curr_y - pos_y;
45
46     always_comb begin
47         pixel_on = 0;
48         if (curr_x >= pos_x && curr_x < pos_x + (8 * TOTAL_W) &&
49             curr_y >= pos_y && curr_y < pos_y + DIGIT_H) begin
50
51             digit_idx = rel_x / TOTAL_W;
52             dx = (rel_x % TOTAL_W) / SCALE;
53             dy = rel_y / SCALE;
54
55             if (dx < 4 && dy < 5) begin
56                 case (7 - digit_idx)
57                     0: digit = number[3:0];
58                     1: digit = number[7:4];
59                     2: digit = number[11:8];
60                     3: digit = number[15:12];
61                     4: digit = number[19:16];
62                     5: digit = number[23:20];
63                     6: digit = number[27:24];
64                     7: digit = number[31:28];
65                 endcase
66
67                 if (digit < 10) begin
68                     if (current_glyph[19 - (dy*4 + dx)]) pixel_on = 1;

```

```

69           end
70       end
71   end
72 end
73
74 endmodule

```

## draw\_string.sv

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

```

1 // draw_string: renders a single ASCII character with an 8x8 bitmap
2     font and
3 // optional scaling; font covers A-Z, 0-9, space, hyphen, and colon.
4 `include "../GLOBAL.sv"
5 module draw_string (
6     input  logic [10:0] curr_x,
7     input  logic [9:0]  curr_y,
8     input  logic [10:0] pos_x,
9     input  logic [9:0]  pos_y,
10    input  logic [7:0]  char_code, // ASCII character code
11    input  logic [1:0]  scale,    // Scale factor (1, 2, 3...)
12    output logic          pixel_on
13);
14
15
16    logic [63:0] font_data;
17
18    always_comb begin
19        case (char_code)
20            // Space
21            8'h20: font_data = 64'h0000000000000000;
22            // Numbers
23            8'h30: font_data = 64'h3C666E7666663C00; // 0
24            8'h31: font_data = 64'h18181818181800; // 1
25            8'h32: font_data = 64'h3C060C1830607E00; // 2
26            8'h33: font_data = 64'h3C06061C06063C00; // 3
27            8'h34: font_data = 64'h060E1E667F060600; // 4
28            8'h35: font_data = 64'h7E607C0606663C00; // 5
29            8'h36: font_data = 64'h3C60607C66663C00; // 6
30            8'h37: font_data = 64'h7E060C1818181800; // 7
31            8'h38: font_data = 64'h3C66663C66663C00; // 8
32            8'h39: font_data = 64'h3C66663E06063C00; // 9
33            // Letters
34            8'h41: font_data = 64'h3C66667E66666600; // A
35            8'h42: font_data = 64'h7C66667C66667C00; // B
36            8'h43: font_data = 64'h3C66606060663C00; // C
37            8'h44: font_data = 64'h786C6666666C7800; // D
38            8'h45: font_data = 64'h7E60607860607E00; // E
39            8'h46: font_data = 64'h7E60607860606000; // F
40            8'h47: font_data = 64'h3C66606E66663C00; // G
41            8'h48: font_data = 64'h6666667E66666600; // H
42            8'h49: font_data = 64'h3C18181818183C00; // I
43            8'h4A: font_data = 64'h1E0C0C0C6C6C3800; // J
44            8'h4B: font_data = 64'h666C7870786C6600; // K
45            8'h4C: font_data = 64'h6060606060607E00; // L
46            8'h4D: font_data = 64'h63777F6B63636300; // M

```

```

45     8'h4E: font_data = 64'h66767E7E6E666600; // N
46     8'h4F: font_data = 64'h3C66666666663C00; // O
47     8'h50: font_data = 64'h7C66667C60606000; // P
48     8'h51: font_data = 64'h3C66666E6C663A00; // Q
49     8'h52: font_data = 64'h7C66667C786C6600; // R
50     8'h53: font_data = 64'h3C60603C06063C00; // S
51     8'h54: font_data = 64'h7E18181818181800; // T
52     8'h55: font_data = 64'h66666666666663C00; // U
53     8'h56: font_data = 64'h666666666663C1800; // V
54     8'h57: font_data = 64'h63636B7F77636300; // W
55     8'h58: font_data = 64'h66663C183C666600; // X
56     8'h59: font_data = 64'h6666663C18181800; // Y
57     8'h5A: font_data = 64'h7E060C1830607E00; // Z
58 // Hyphen
59     8'h2D: font_data = 64'h0000007E00000000; // -
60 // Colon
61     8'h3A: font_data = 64'h0000180000180000; // :
62     default: font_data = 64'h0000000000000000;
63   endcase
64 end
65
66 localparam CHAR_W = 8;
67 localparam CHAR_H = 8;
68 localparam CHAR_SPACING = 1;
69
70 logic [10:0] rel_x, rel_y;
71 logic [2:0] pixel_x, pixel_y;
72
73 assign rel_x = curr_x - pos_x;
74 assign rel_y = curr_y - pos_y;
75
76 logic [10:0] scaled_rel_x, scaled_rel_y;
77 assign scaled_rel_x = rel_x / scale;
78 assign scaled_rel_y = rel_y / scale;
79
80 assign pixel_x = scaled_rel_x[2:0];
81 assign pixel_y = scaled_rel_y[2:0];
82
83 always_comb begin
84   pixel_on = 0;
85   if (rel_x >= 0 && rel_x < (CHAR_W * scale) &&
86       rel_y >= 0 && rel_y < (CHAR_H * scale)) begin
87     if (font_data[63 - (pixel_y * 8 + pixel_x)]) begin
88       pixel_on = 1;
89     end
90   end
91 end
92
93 endmodule
94
95 // draw_string_line: renders a short string (up to 16 chars) by tiling
96 // draw_string instances with configurable scale and spacing.
97 module draw_string_line (
98   input logic [10:0] curr_x,
99   input logic [9:0] curr_y,
100  input logic [10:0] pos_x,
101  input logic [9:0] pos_y,
102  input logic [7:0] str_chars [0:15], // Up to 16 characters

```

```

103     input  logic [3:0]  str_len,           // Actual string length
104     input  logic [1:0]  scale,            // Scale factor
105     output logic       pixel_on
106 );
107
108 localparam CHAR_W = 8;
109 localparam CHAR_SPACING = 1;
110 // localparam CHAR_TOTAL_W = CHAR_W + CHAR_SPACING;
111
112 logic [10:0] char_total_w;
113 assign char_total_w = (CHAR_W + CHAR_SPACING) * scale;
114
115 logic [10:0] rel_x, rel_y;
116 logic [3:0] char_idx;
117 logic [7:0] current_char;
118 logic char_pixel_on;
119
120 assign rel_x = curr_x - pos_x;
121 assign rel_y = curr_y - pos_y;
122
123 // Determine which character we're in
124 assign char_idx = rel_x / char_total_w;
125
126 // Get the character code for this position
127 always_comb begin
128     if (char_idx < str_len) begin
129         current_char = str_chars[char_idx];
130     end else begin
131         current_char = 8'h20; // Space
132     end
133 end
134
135 // Calculate position within character
136 logic [10:0] char_pos_x;
137 assign char_pos_x = pos_x + (char_idx * char_total_w);
138
139 draw_string char_draw (
140     .curr_x(curr_x),
141     .curr_y(curr_y),
142     .pos_x(char_pos_x),
143     .pos_y(pos_y),
144     .char_code(current_char),
145     .scale(scale),
146     .pixel_on(char_pixel_on)
147 );
148
149 assign pixel_on = char_pixel_on;
150
151 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.*

```

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

```

```

3 // key (with priority) and its scan code across four digits.
4 module seg7_key_display(
5     input logic          clk,           // System clock (100MHz)
6     input logic          rst,
7
8     // Current key state
9     input logic [7:0]   scan_code,    // Current scan code
10    input logic         key_valid,    // Key is valid (pressed)
11
12    // Active key indicators (directly from game)
13    input logic         key_left,
14    input logic         key_right,
15    input logic         key_down,
16    input logic         key_rotate_cw,
17    input logic         key_rotate_ccw,
18    input logic         key_drop,
19    input logic         key_hold,
20
21    // 7-Segment outputs
22    output logic [6:0]  SEG,          // Segment pattern (active low)
23    output logic [7:0]  AN,           // Anode select (active low)
24    output logic        DP            // Decimal point (active low)
25 );
26
27    // Clock divider for multiplexing (need ~1kHz refresh rate)
28    // 100MHz / 2^17      763 Hz
29    logic [19:0] clk_div;
30    logic [2:0] digit_sel;
31
32    always_ff @(posedge clk) begin
33        if (rst)
34            clk_div <= 0;
35        else
36            clk_div <= clk_div + 1;
37    end
38
39    assign digit_sel = clk_div[19:17];
40
41    // Determine what to display based on active key
42    // Priority: Drop > Hold > Rotate > Left > Right > Down
43    logic [3:0] display_char [0:7]; // Character for each digit
44
45    // Character encoding for 7-segment (active low)
46    // Segments: gfedcba
47    function logic [6:0] char_to_seg(input [3:0] c);
48        case (c)
49            // Numbers
50            4'h0: return 7'b1000000; // 0
51            4'h1: return 7'b1111001; // 1
52            4'h2: return 7'b0100100; // 2
53            4'h3: return 7'b0110000; // 3
54            4'h4: return 7'b0011001; // 4
55            4'h5: return 7'b0010010; // 5
56            4'h6: return 7'b0000010; // 6
57            4'h7: return 7'b1111000; // 7
58            4'h8: return 7'b0000000; // 8
59            4'h9: return 7'b0010000; // 9
60            4'hA: return 7'b0001000; // A

```

```

61         4'hB: return 7'b0000011; // b
62         4'hC: return 7'b1000110; // C
63         4'hD: return 7'b0100001; // d
64         4'hE: return 7'b00000110; // E
65         4'hF: return 7'b0001110; // F
66         default: return 7'b1111111; // blank
67     endcase
68 endfunction
69
70 // Special characters for key names
71 localparam [6:0] SEG_L = 7'b1000111; // L
72 localparam [6:0] SEG_r = 7'b0101111; // r (lowercase)
73 localparam [6:0] SEG_d = 7'b0100001; // d (lowercase)
74 localparam [6:0] SEG_U = 7'b1000001; // U
75 localparam [6:0] SEG_H = 7'b0001001; // H
76 localparam [6:0] SEG_S = 7'b0010010; // S (same as 5)
77 localparam [6:0] SEG_P = 7'b0001100; // P
78 localparam [6:0] SEG_o = 7'b0100011; // o (lowercase)
79 localparam [6:0] SEG_t = 7'b0000111; // t (lowercase)
80 localparam [6:0] SEG_n = 7'b0101011; // n (lowercase)
81 localparam [6:0] SEG_DASH = 7'b0111111; // -
82 localparam [6:0] SEG_BLANK = 7'b1111111; // blank
83
84 // What to display on each digit
85 logic [6:0] seg_data [0:7];
86
87 always_comb begin
88     // Default: show dashes (no key)
89     seg_data[7] = SEG_DASH; // Leftmost
90     seg_data[6] = SEG_DASH;
91     seg_data[5] = SEG_DASH;
92     seg_data[4] = SEG_DASH;
93     seg_data[3] = SEG_BLANK;
94     seg_data[2] = SEG_BLANK;
95     seg_data[1] = SEG_BLANK;
96     seg_data[0] = SEG_BLANK; // Rightmost
97
98     // Show key name on left 4 digits, scan code on right 2 digits
99     if (key_drop) begin
100         // "droP" for drop/space
101         seg_data[7] = SEG_d;
102         seg_data[6] = SEG_r;
103         seg_data[5] = SEG_o;
104         seg_data[4] = SEG_P;
105         // Scan code 0x29
106         seg_data[1] = 7'b0100100; // 2
107         seg_data[0] = 7'b0010000; // 9
108     end
109     else if (key_hold) begin
110         // "HoLd" for hold
111         seg_data[7] = SEG_H;
112         seg_data[6] = SEG_o;
113         seg_data[5] = SEG_L;
114         seg_data[4] = SEG_d;
115         // Scan code 0x12
116         seg_data[1] = 7'b1111001; // 1
117         seg_data[0] = 7'b0100100; // 2
118     end

```

```

119     else if (key_rotate_cw) begin
120         // "rot r" for rotate CW
121         seg_data[7] = SEG_r;
122         seg_data[6] = SEG_o;
123         seg_data[5] = SEG_t;
124         seg_data[4] = SEG_r;
125         // Scan code (generic or specific)
126         seg_data[1] = 7'b1111000; // 7
127         seg_data[0] = 7'b0010010; // 5
128     end
129     else if (key_rotate_ccw) begin
130         // "rot L" for rotate CCW
131         seg_data[7] = SEG_r;
132         seg_data[6] = SEG_o;
133         seg_data[5] = SEG_t;
134         seg_data[4] = SEG_L;
135         // Scan code
136         seg_data[1] = 7'b1111001; // 1
137         seg_data[0] = 7'b0001000; // A (1A for Z)
138     end
139     else if (key_left) begin
140         // "LEFT" for left
141         seg_data[7] = SEG_L;
142         seg_data[6] = 7'b0000110; // E
143         seg_data[5] = 7'b0001110; // F
144         seg_data[4] = SEG_t;
145         // Scan code 0x6B
146         seg_data[1] = 7'b0000010; // 6
147         seg_data[0] = 7'b0000011; // b
148     end
149     else if (key_right) begin
150         // "rGHT" for right
151         seg_data[7] = SEG_r;
152         seg_data[6] = 7'b1000010; // G (approximation)
153         seg_data[5] = SEG_H;
154         seg_data[4] = SEG_t;
155         // Scan code 0x74
156         seg_data[1] = 7'b1111000; // 7
157         seg_data[0] = 7'b0011001; // 4
158     end
159     else if (key_down) begin
160         // "doUn" for down
161         seg_data[7] = SEG_d;
162         seg_data[6] = SEG_o;
163         seg_data[5] = SEG_U;
164         seg_data[4] = SEG_n;
165         // Scan code 0x72
166         seg_data[1] = 7'b1111000; // 7
167         seg_data[0] = 7'b0100100; // 2
168     end
169 end
170
171 // Multiplex outputs
172 always_comb begin
173     // All anodes off by default
174     AN = 8'b11111111;
175     SEG = SEG_BLANK;
176     DP = 1'b1; // DP off

```

```

177     // Enable selected digit
178     AN[digit_sel] = 1'b0;
179     SEG = seg_data[digit_sel];
180
181     // Add decimal point between key name and scan code
182     if (digit_sel == 4)
183         DP = 1'b0;
184
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
3  * with kicks,
4  * manages holds, scoring/levels, line clears, ghost piece, and exposes
5  * display
6  * signals plus status outputs.
7  */
8  'include "../GLOBAL.sv"
9
10 module game_control (
11     input logic clk,
12     input logic rst,
13     input logic tick_game,
14
15     // Inputs (Single Pulse)
16     input logic key_left,
17     input logic key_right,
18     input logic key_down,
19     input logic key_rotate_cw,
20     input logic key_rotate_ccw,
21     input logic key_drop,
22     input logic key_hold,
23     input logic key_drop_held,
24
25     output field_t display,
26     output logic [31:0] score,
27     output logic game_over,
28     output tetromino_ctrl t_next_disp,
29     output tetromino_ctrl t_hold_disp,
30     output logic hold_used_out,
31     output logic [3:0] current_level_out,
32     output logic signed ['FIELD_VERTICAL_WIDTH : 0] ghost_y,
33     output tetromino_ctrl t_curr_out,
34
35     output logic [7:0] total_lines_cleared_out // Exposed for
36     level bar
37 );

```

```

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
91 // map to

```

```

91 // the drop delay expressed in game ticks. Keep this combinational to
92 // avoid
93 // hidden latches on the level or speed calculation.
94 always_comb begin
95
96     if (current_level != 15) current_level = total_lines_cleared / 10;
97
98     case (current_level)
99         0: drop_speed_frames = 40;
100        1: drop_speed_frames = 37;
101        2: drop_speed_frames = 34;
102        3: drop_speed_frames = 30;
103        4: drop_speed_frames = 26;
104        5: drop_speed_frames = 21;
105        6: drop_speed_frames = 15;
106        7: drop_speed_frames = 12;
107        8: drop_speed_frames = 8;
108        9: drop_speed_frames = 6;
109        10: drop_speed_frames = 5;
110        11: drop_speed_frames = 5;
111        12: drop_speed_frames = 5;
112        13: drop_speed_frames = 4;
113        14: drop_speed_frames = 3;
114        15: drop_speed_frames = 2;
115         default: drop_speed_frames = 2; // Impossible mode!
116     endcase
117 end
118
119 // SRS KICK TABLES (Corrected for Y-down coordinate system)
120 localparam kick_offset_t JLSTZ_KICKS_CW[4][5] = ^{
121     // 0 R
122     '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{-3'sd1, -3'sd1}, '{3'sd0, 3'sd2}, '{-3'sd1, 3'sd2}},
123     // R 2
124     '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{3'sd1, 3'sd1}, '{3'sd0, -3'sd2}, '{3'sd1, -3'sd2}},
125     // 2 L
126     '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{3'sd1, -3'sd1}, '{3'sd0, 3'sd2}, '{3'sd1, 3'sd2}},
127     // L 0
128     '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{-3'sd1, 3'sd1}, '{3'sd0, -3'sd2}, '{-3'sd1, -3'sd2}}
129 };
130
131 localparam kick_offset_t JLSTZ_KICKS_CCW[4][5] = ^{
132     // 0 L
133     '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{3'sd1, -3'sd1}, '{3'sd0, 3'sd2}, '{3'sd1, 3'sd2}},
134     // R 0
135     '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{3'sd1, 3'sd1}, '{3'sd0, -3'sd2}, '{3'sd1, -3'sd2}},
136     // 2 R
137     '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{-3'sd1, -3'sd1}, '{3'sd0, 3'sd2}, '{-3'sd1, 3'sd2}},
138     // L 2
139     '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{-3'sd1, 3'sd1}, '{3'sd0, -3'sd2}, '{-3'sd1, -3'sd2}}

```

```

140 };
141
142 localparam kick_offset_t I_KICKS_CW[4][5] = ^{
143     // 0 R
144     '{'{3'sd0, 3'sd0}, '{-3'sd2, 3'sd0}, '{3'sd1, 3'sd0}, '{-3'sd2, 3'sd1},
145     // R 2
146     '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{3'sd2, 3'sd0}, '{-3'sd1, -3'sd2},
147     // 2 L
148     '{'{3'sd0, 3'sd0}, '{3'sd2, 3'sd0}, '{-3'sd1, 3'sd0}, '{3'sd2, -3'sd1},
149     // L 0
150     '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{-3'sd2, 3'sd0}, '{3'sd1, 3'sd2},
151 };
152
153 localparam kick_offset_t I_KICKS_CCW[4][5] = ^{
154     // 0 L
155     '{'{3'sd0, 3'sd0}, '{-3'sd1, 3'sd0}, '{3'sd2, 3'sd0}, '{-3'sd1, -3'sd2},
156     // R 0
157     '{'{3'sd0, 3'sd0}, '{3'sd2, 3'sd0}, '{-3'sd1, 3'sd0}, '{3'sd2, -3'sd1},
158     // 2 R
159     '{'{3'sd0, 3'sd0}, '{3'sd1, 3'sd0}, '{-3'sd2, 3'sd0}, '{3'sd1, 3'sd2},
160     // L 2
161     '{'{3'sd0, 3'sd0}, '{-3'sd2, 3'sd0}, '{3'sd1, 3'sd0}, '{-3'sd2, 3'sd1},
162 };
163
164 kick_offset_t current_kick;
165 always_comb begin
166     current_kick = '{3'sd0, 3'sd0};
167
168     if (ps == ROTATE) begin
169         case (t_curr.idx.data)
170             'TETROMINO_I_IDX: begin
171                 if (rotate_direction)
172                     current_kick = I_KICKS_CW[rotation_from][
173                         kick_attempt];
174                 else
175                     current_kick = I_KICKS_CCW[rotation_from][
176                         kick_attempt];
177             end
178             'TETROMINO_O_IDX: begin
179                 current_kick = '{3'sd0, 3'sd0};
180             end
181             default: begin
182                 if (rotate_direction)
183                     current_kick = JLSTZ_KICKS_CW[rotation_from][
184                         kick_attempt];
185                 else
186                     current_kick = JLSTZ_KICKS_CCW[rotation_from][
187                         kick_attempt];
188             end
189         endcase

```

```

186     end
187 end
188
189 // =====
190 // SPIN DETECTION LOGIC
191 // =====
192 // Checks if piece is surrounded by blocks/walls (3-corner rule for T
193 // -spins)
194 // =====
195
196 spin_detector spin_detect (
197     .t_ctrl(t_curr),
198     .f(f_curr),
199     .last_move_was_rotation(last_move_was_rotation),
200     .kick_used(kick_used),
201     .is_t_spin(is_t_spin),
202     .is_t_spin_mini(is_t_spin_mini),
203     .is_s_spin(is_s_spin),
204     .is_z_spin(is_z_spin),
205     .is_j_spin(is_j_spin),
206     .is_l_spin(is_l_spin),
207     .is_i_spin(is_i_spin)
208 );
209
210 // Timers
211 integer drop_timer;
212 integer lock_timer; // NEW: Lock delay timer
213
214 localparam LOCK_DELAY_MAX = 30; // ~0.5s at 60Hz
215
216 // Submodules
217 generate_tetromino gen (
218     .clk(clk),
219     .rst(rst),
220     .enable(ps == GEN || ps == WARMUP),
221     .t_out(t_gen),
222     .t_next_out(t_gen_next)
223 );
224
225 tetromino_ctrl t_rotated;
226 rotate_tetromino rot (
227     .clk(clk),
228     .enable(ps == ROTATE && kick_attempt == 0),
229     .clockwise(rotate_direction),
230     .t_in(t_curr),
231     .t_out(t_rotated),
232     .success(rotate_success),
233     .done(rotate_done)
234 );
235
236 check_valid validator (
237     .t_ctrl(t_check),
238     .f(f_curr),
239     .isValid(valid)
240 );
241
242 create_field merger (
243     .t_ctrl(t_curr),

```

```

243     .f(f_curr),
244     .f_out(f_disp)
245 );
246
247 clean_field cleaner (
248     .clk(clk),
249     .enable(ps == CLEAN),
250     .f_in(f_curr),
251     .f_out(f_cleaned),
252     .lines_cleared(lines_cleared),
253     .done(clean_done)
254 );
255
256 ghost_calc ghost (
257     .t_curr(t_curr),
258     .f(f_curr),
259     .ghost_y(ghost_y)
260 );
261
262     assign t_curr_out = t_curr;
263     assign display = f_disp;
264     assign t_nextDisp = t_gen_next;
265     assign t_holdDisp = t_hold;
266     assign holdUsed_out = holdUsed;
267     assign currentLevel_out = current_level;
268     assign totalLinesCleared_out = total_lines_cleared;
269
270 // Candidate Logic for Validation
271 always_comb begin
272     t_check = t_curr;
273     case (ps)
274         GEN_WAIT: t_check = t_gen;
275         MOVE_LEFT: t_check.coordinate.x = t_curr.coordinate.x - 1;
276         MOVE_RIGHT: t_check.coordinate.x = t_curr.coordinate.x + 1;
277         DOWN: t_check.coordinate.y = t_curr.coordinate.y + 1;
278         HARD_DROP: t_check.coordinate.y = t_curr.coordinate.y + 1;
279         IDLE: t_check.coordinate.y = t_curr.coordinate.y + 1;
280         ROTATE: begin
281             t_check = t_rotated;
282             t_check.coordinate.x = t_rotated.coordinate.x +
283             current_kick.x;
284             t_check.coordinate.y = t_rotated.coordinate.y +
285             current_kick.y;
286             end
287         HOLD_SWAP: begin
288             t_check = t_hold_old;
289             t_check.coordinate.x = 3;
290             t_check.coordinate.y = 0;
291             t_check.rotation = 0;
292             end
293         default: t_check = t_curr;
294     endcase
295 end
296
297 // FSM Next State Logic
298 always_comb begin
299     ns = ps;
300     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 =
313             DOWN;
314             // Lock Delay Logic: If on ground (!valid) and timer
315             expired -> Lock
316             else if (!valid && lock_timer >= LOCK_DELAY_MAX) ns = CLEAN
317             ;
318         end
319
320     MOVE_LEFT: ns = IDLE;
321     MOVE_RIGHT: ns = IDLE;
322
323     ROTATE: begin
324         if (kick_attempt == 0 && !rotate_done) begin
325             ns = ROTATE;
326         end else if (valid) begin
327             ns = IDLE;
328         end else if (kick_attempt < 4) begin
329             ns = ROTATE;
330         end else begin
331             ns = IDLE;
332         end
333     end
334
335     DOWN: begin
336         if (valid) begin
337             ns = IDLE;
338         end else begin
339             ns = IDLE;
340         end
341     end
342
343     HARD_DROP: begin
344         if (valid) ns = HARD_DROP;
345         else ns = CLEAN;
346     end
347
348     HOLD: begin
349         if (hold_empty) ns = GEN;
350         else ns = HOLD_SWAP;
351     end
352
353     HOLD_SWAP: begin
354         if (valid) ns = IDLE;
355         else ns = GAME_OVER_STATE;
356     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 // =====
407 // SPIN BONUSES (Modern Tetris Guidelines)
408 // =====
409     spin_bonus = 16'd0;

```

```

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

```

```

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

```

```

517         lock_timer <= 0;
518     end
519 end
520
521 if (ps == DOWN || ps == GEN) drop_timer <= 0;
522
523 // Data Path Updates
524 case (ps)
525     IDLE: begin
526         kick_attempt <= 0;
527         if (key_rotate_cw) begin
528             rotate_direction <= 1;
529             rotation_from <= t_curr.rotation;
530         end else if (key_rotate_ccw) begin
531             rotate_direction <= 0;
532             rotation_from <= t_curr.rotation;
533         end
534
535         // Clear last move flag if moving left/right/down
536         if (key_left || key_right || key_down) begin
537             last_move_was_rotation <= 0;
538         end
539
540         // Lock piece when timer expires (right before
541         transitioning to CLEAN)
542         if (ns == CLEAN) begin
543             f_curr <= f_disp;
544         end
545     end
546
547
548     GEN_WAIT: begin
549         t_curr <= t_gen;
550         lock_timer <= 0;
551         hold_used <= 0;          // new piece allows hold again
552     end
553
554     DROP_LOCKOUT: begin
555         hold_used <= 0;
556         // Flags are now cleared by msg_timer
557     end
558
559     MOVE_LEFT: begin
560         if (valid) begin
561             t_curr.coordinate.x <= t_curr.coordinate.x - 1;
562             last_move_was_rotation <= 0; // Movement cancels
563             rotation_flag
564             lock_timer <= 0; // Reset lock timer on successful
565             move
566         end
567     end
568
569     MOVE_RIGHT: begin
570         if (valid) begin
571             t_curr.coordinate.x <= t_curr.coordinate.x + 1;
572             last_move_was_rotation <= 0; // Movement cancels
573             rotation_flag

```

```

571         lock_timer <= 0; // Reset lock timer on successful
572     move
573         end
574     end
575 
576     ROTATE: begin
577         if (kick_attempt == 0) begin
578             if (rotate_done) begin
579                 if (valid) begin
580                     t_curr <= t_check;
581                     kick_attempt <= 0;
582                     last_move_was_rotation <= 1; // Mark as
583                     rotation move
584                     kick_used <= 0; // No kick needed
585                     lock_timer <= 0; // Reset lock timer
586                     end else begin
587                         kick_attempt <= 1;
588                         end
589                     end else begin
590                         if (valid) begin
591                             t_curr <= t_check;
592                             kick_attempt <= 0;
593                             last_move_was_rotation <= 1; // Mark as
594                             rotation move
595                             kick_used <= kick_attempt; // Record which
596                         kick worked
597                         lock_timer <= 0; // Reset lock timer
598                         end else if (kick_attempt < 4) begin
599                             kick_attempt <= kick_attempt + 1;
600                         end else begin
601                             kick_attempt <= 0;
602                             last_move_was_rotation <= 0; // Rotation
603                         failed
604                         end
605                     end
606                 end
607             end
608             DOWN: begin
609                 if (valid) begin
610                     t_curr.coordinate.y <= t_curr.coordinate.y + 1;
611                     last_move_was_rotation <= 0; // Down movement
612                 cancels rotation
613                 lock_timer <= 0; // Reset lock timer on successful
614             move
615                 end
616             end
617             HARD_DROP: begin
618                 if (valid) begin
619                     t_curr.coordinate.y <= t_curr.coordinate.y + 1;
620                 end else begin
621                     f_curr <= f_disp;
622                     // Hard drop cancels rotation flag (debatable - you
623                     can change this)
624                     last_move_was_rotation <= 0;
625                 end
626             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 +
652 lines_cleared;
653         end
654
655         if (lines_cleared != 0) begin
656             if (consecutive_clears < 15)
657                 consecutive_clears <= consecutive_clears +
658 1;
659             end else begin
660                 consecutive_clears <= 0;
661             end
662
663             // BCD Addition
664             if (lines_cleared != 0) begin
665                 automatic logic [3:0] new_ones, new_tens,
666 newHundreds, newThousands, newTenths;
667                 automatic logic c0, c1, c2, c3, c4;
668
669                 new_ones = score[3:0] + bcd_ones;
670                 if (new_ones >= 10) begin
671                     score[3:0] <= new_ones - 10;
672                     c0 = 1;
673                 end else begin
674                     score[3:0] <= new_ones;
675                     c0 = 0;
676                 end
677             end
678         end
679     end
680 end

```

```

676         new_tens = score[7:4] + bcd_tens + c0;
677         if (new_tens >= 10) begin
678             score[7:4] <= new_tens - 10;
679             c1 = 1;
680         end else begin
681             score[7:4] <= new_tens;
682             c1 = 0;
683         end
684
685         newHundreds = score[11:8] + bcdHundreds + c1;
686         if (newHundreds >= 10) begin
687             score[11:8] <= newHundreds - 10;
688             c2 = 1;
689         end else begin
690             score[11:8] <= newHundreds;
691             c2 = 0;
692         end
693
694         newThousands = score[15:12] + bcdThousands +
695         c2;
696         if (newThousands >= 10) begin
697             score[15:12] <= newThousands - 10;
698             c3 = 1;
699         end else begin
700             score[15:12] <= newThousands;
701             c3 = 0;
702         end
703
704         newTenthsousands = score[19:16] +
705         bcdTenthsousands + c3;
706         if (newTenthsousands >= 10) begin
707             score[19:16] <= newTenthsousands - 10;
708             c4 = 1;
709         end else begin
710             score[19:16] <= newTenthsousands;
711             c4 = 0;
712         end
713
714         if (c4) begin
715             if (score[23:20] == 9) begin
716                 score[23:20] <= 0;
717                 if (score[27:24] == 9) begin
718                     score[27:24] <= 0;
719                     score[31:28] <= score[31:28] + 1;
720                 end else begin
721                     score[27:24] <= score[27:24] + 1;
722                 end
723             end else begin
724                 score[23:20] <= score[23:20] + 1;
725             end
726         end
727     end
728
729     GAME_OVER_STATE: begin
730         game_over <= 1;
731     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
3   * tetromino
4   * controls and refreshes the bag when emptied.
5   */
6  'include "../GLOBAL.sv"
7
8  module generate_tetromino (
9      input    logic          clk,
10     input    logic          rst,
11     input    logic          enable,
12     output   tetromino_ctrl t_out,
13     output   tetromino_ctrl t_next_out // Optional: Next piece preview
14 );
15
16     logic [15:0] lfsr;
17     logic [2:0]  rand_idx;
18
19     always_ff @(posedge clk) begin
20         if (rst) begin
21             lfsr <= 16'hACE1;
22         end else begin
23             lfsr <= {lfsr[14:0], lfsr[15] ^ lfsr[13] ^ lfsr[12] ^ lfsr
24             [10]};
25         end
26     end
27
28     assign rand_idx = lfsr[2:0] % 'NUMBER_OF_TETROMINO;
29
30     tetromino_t current_shape;

```

```

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) %
39            7;
40        else if (bag[(rand_idx + 2) % 7]) selected_idx = (rand_idx + 2) %
41            7;
42        else if (bag[(rand_idx + 3) % 7]) selected_idx = (rand_idx + 3) %
43            7;
44        else if (bag[(rand_idx + 4) % 7]) selected_idx = (rand_idx + 4) %
45            7;
46        else if (bag[(rand_idx + 5) % 7]) selected_idx = (rand_idx + 5) %
47            7;
48        else if (bag[(rand_idx + 6) % 7]) selected_idx = (rand_idx + 6) %
49            7;
50
51        next_bag = bag & ~(1 << selected_idx);
52
53        if (next_bag == 0) next_bag = 7'b1111111;
54    end
55
56    always_comb begin
57        case (selected_idx)
58            'TETROMINO_I_IDX: begin
59                current_shape.data[0] = {4'b0000, 4'b1111, 4'b0000, 4'b0000
60            };
61                current_shape.data[1] = {4'b0010, 4'b0010, 4'b0010, 4'b0010
62            };
63                current_shape.data[2] = {4'b0000, 4'b1111, 4'b0000, 4'b0000
64            };
65                current_shape.data[3] = {4'b0010, 4'b0010, 4'b0010, 4'b0010
66            };
67            end
68            'TETROMINO_J_IDX: begin
69                current_shape.data[0] = {4'b1000, 4'b1110, 4'b0000, 4'b0000
70            };
71                current_shape.data[1] = {4'b0110, 4'b0100, 4'b0100, 4'b0000
72            };
73                current_shape.data[2] = {4'b0000, 4'b1110, 4'b0010, 4'b0000
74            };
75                current_shape.data[3] = {4'b0010, 4'b0010, 4'b0110, 4'b0000
76            };
77            end
78            'TETROMINO_L_IDX: begin
79                current_shape.data[0] = {4'b0010, 4'b1110, 4'b0000, 4'b0000
80            };
81                current_shape.data[1] = {4'b0100, 4'b0100, 4'b0110, 4'b0000
82            };
83                current_shape.data[2] = {4'b0000, 4'b1110, 4'b1000, 4'b0000
84            };
85                current_shape.data[3] = {4'b1100, 4'b0100, 4'b0100, 4'b0000
86            };
87            end
88        end
89    end

```

```

70     'TETROMINO_O_IDX: begin
71         current_shape.data[0] = {4'b0110, 4'b0110, 4'b0000, 4'b0000
72     };
73         current_shape.data[1] = {4'b0110, 4'b0110, 4'b0000, 4'b0000
74     };
75         current_shape.data[2] = {4'b0110, 4'b0110, 4'b0000, 4'b0000
76     };
77         current_shape.data[3] = {4'b0110, 4'b0110, 4'b0000, 4'b0000
78     };
79     end
80     'TETROMINO_S_IDX: begin
81         current_shape.data[0] = {4'b0110, 4'b1100, 4'b0000, 4'b0000
82     };
83         current_shape.data[1] = {4'b0100, 4'b0110, 4'b0010, 4'b0000
84     };
85         current_shape.data[2] = {4'b0000, 4'b0110, 4'b1100, 4'b0000
86     };
87         current_shape.data[3] = {4'b1000, 4'b1100, 4'b0100, 4'b0000
88     };
89     end
90     'TETROMINO_T_IDX: begin
91         current_shape.data[0] = {4'b0100, 4'b1110, 4'b0000, 4'b0000
92     };
93         current_shape.data[1] = {4'b0100, 4'b0110, 4'b0100, 4'b0000
94     };
95         current_shape.data[2] = {4'b0000, 4'b1110, 4'b0100, 4'b0000
96     };
97         current_shape.data[3] = {4'b0100, 4'b1100, 4'b0100, 4'b0000
98     };
99     end
100    default: current_shape = '0;
101    endcase
102 end
103
104 always_ff @(posedge clk) begin
105     if (rst) begin
106         bag <= 7'b1111111;
107
108         t_out.idx.data <= 'TETROMINO_EMPTY;
109         t_out.rotation <= 0;
110         t_out.coordinate.x <= 3;
111         t_out.coordinate.y <= 0;
112         t_out.tetromino.data <= '0;
113
114         t_next_out.idx.data <= 0; // Default I
115         t_next_out.tetromino <= '0;
116         t_next_out.rotation <= 0;
117         t_next_out.coordinate.x <= 3;

```

```

112     t_next_out.coordinate.y <= 0;
113
114 end else if (enable) begin
115     t_out <= t_next_out;
116
117     t_next_out.idx.data <= selected_idx;
118     t_next_out.tetromino <= current_shape;
119     t_next_out.rotation <= 0;
120     t_next_out.coordinate.x <= 3;
121     t_next_out.coordinate.y <= 0;
122
123     bag <= next_bag;
124 end
125 end
126
127 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.*

```

1  /* check_valid
2   * Combinational gatekeeper: returns 1 only when the candidate
3   * tetromino
4   * stays inside the field and does not overlap placed blocks.
5   */
5  'include "../GLOBAL.sv"
6
7 module check_valid (
8   input   tetromino_ctrl  t_ctrl,
9   input   field_t          f,
10  output  logic            isValid
11 );
12
13 logic [1:0] currRotation;
14 assign currRotation = t_ctrl.rotation;
15
16 integer signed i, j;
17 integer signed tx, ty;
18
19 always_comb begin
20   isValid = 1'b1;
21   for (i = 0; i < 4; ++i) begin
22     for (j = 0; j < 4; ++j) begin
23       if (t_ctrl.tetromino.data[currRotation][i][j] == 1) begin
24         tx = t_ctrl.coordinate.x + j;
25         ty = t_ctrl.coordinate.y + i;
26
27         if (tx < 0 || tx >= 'FIELD_HORIZONTAL ||
28             ty < 0 || ty >= 'FIELD_VERTICAL) begin
29           isValid = 1'b0;
30         end
31       end
32     end
33     if (f.data[ty][tx].data != 'TETROMINO_EMPTY) begin
34       isValid = 1'b0;
35     end

```

```

35         end
36     end
37   end
38 end
39 end
40 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.*

```

1  /* clean_field
2   * Sequential row clearer: scans for full rows, shifts above content
3   * downward,
4   * counts cleared lines, and exposes the updated field when done.
5   */
6   'include "../GLOBAL.sv"
7
8 module clean_field (
9   input   logic      clk,
10  input   logic      enable,
11  input   field_t    f_in,
12  output  field_t    f_out,
13  output  logic [2:0] lines_cleared,
14  output  logic      done
15 );
16
17   typedef enum logic [1:0] {IDLE, CHECK, SHIFT, FINISH} state_t;
18   state_t state = IDLE;
19
20   integer row, col, k;
21   logic row_full;
22
23   field_t f_temp;
24
25   always_ff @(posedge clk) begin
26     if (!enable) begin
27       state <= IDLE;
28       done <= 0;
29       lines_cleared <= 0;
30     end else begin
31       case (state)
32         IDLE: begin
33           f_temp <= f_in;
34           row <= 'FIELD_VERTICAL - 1;
35           lines_cleared <= 0;
36           done <= 0;
37           state <= CHECK;
38         end
39
40         CHECK: begin
41           if (row < 0) begin
42             state <= FINISH;
43           end else begin
44             row_full = 1;
45             for (col = 0; col < 'FIELD_HORIZONTAL; col++) begin

```

```

45      if (f_temp.data[row][col].data == 
46          'TETROMINO_EMPTY) begin
47          row_full = 0;
48          end
49          end
50
51          if (row_full) begin
52              lines_cleared <= lines_cleared + 1;
53              state <= SHIFT;
54          end else begin
55              row <= row - 1;
56          end
57      end
58
59 SHIFT: begin
60     for (k = 'FIELD_VERTICAL - 1; k > 0; k--) begin
61         if (k <= row) begin
62             f_temp.data[k] <= f_temp.data[k-1];
63         end
64     end
65     for (col = 0; col < 'FIELD_HORIZONTAL; col++) begin
66         f_temp.data[0][col].data <= 'TETROMINO_EMPTY;
67     end
68
69     state <= CHECK;
70 end
71
72 FINISH: begin
73     f_out <= f_temp;
74     done <= 1;
75 end
76 endcase
77 end
78 end
79
80 endmodule

```

## create\_field.sv

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

```

1  /* create_field
2   * Combines the settled field with the active tetromino to yield a
3   * display
4   * snapshot without mutating the stored field.
5   */
6   include "../GLOBAL.sv"
7
8 module create_field (
9   input  tetromino_ctrl  t_ctrl ,
10  input   field_t        f ,
11  output  field_t        f_out
12 );
13 logic [1:0] currRotation;

```

```

14 assign currRotation = t_ctrl.rotation;
15
16 integer signed i, j;
17 integer signed tx, ty;
18
19 always_comb begin
20     f_out = f;
21
22     for (i = 0; i < 4; ++i) begin
23         for (j = 0; j < 4; ++j) begin
24             if (t_ctrl.tetromino.data[currRotation][i][j] == 1) begin
25                 tx = t_ctrl.coordinate.x + j;
26                 ty = t_ctrl.coordinate.y + i;
27
28                 if (tx >= 0 && tx < 'FIELD_HORIZONTAL &&
29                     ty >= 0 && ty < 'FIELD_VERTICAL) begin
30                     f_out.data[ty][tx] = t_ctrl.idx;
31                 end
32             end
33         end
34     end
35 end
36 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.*

```

1 /* rotate_tetromino
2  * Produces a rotated tetromino candidate (CW or CCW) and flags
3  * completion;
4  * rotation wrap-around relies on 2-bit arithmetic.
5 */
5 include "../GLOBAL.sv"
6
7 module rotate_tetromino (
8     input    logic          clk,
9     input    logic          enable,
10    input    logic          clockwise,
11    input    tetromino_ctrl t_in,
12    output   tetromino_ctrl t_out,
13    output   logic          success,
14    output   logic          done
15 );
16
17 always_ff @(posedge clk) begin
18     if (!enable) begin
19         done <= 0;
20         success <= 0;
21     end else begin
22         t_out <= t_in;
23         if (clockwise) begin
24             t_out.rotation <= t_in.rotation + 1;
25         end else begin
26             t_out.rotation <= t_in.rotation - 1;
27         end

```

```

28         success <= 1;
29         done <= 1;
30     end
31 end
32
33 endmodule

```

### rotate\_clockwise.sv

*Basic rotation matrix. Performs a simple 90-degree clockwise rotation of a  $4 \times 4$  tetromino shape matrix.*

```

1  /* rotate_clockwise
2   * Reference helper that advances rotation clockwise and flags
3   * completion;
4   * caller is responsible for validity and kicks.
5   */
6
7  'include "../GLOBAL.sv"
8
9  module rotate_clockwise (
10    input  logic      clk,
11    input  logic      enable,
12    input  tetromino_ctrl t_in,
13    output tetromino_ctrl t_out,
14    output logic      success,
15    output logic      done
16  );
17
18  always_ff @(posedge clk) begin
19    if (!enable) begin
20      done <= 0;
21      success <= 0;
22    end else begin
23      t_out <= t_in;
24      t_out.rotation <= t_in.rotation + 1;
25      success <= 1;
26      done <= 1;
27    end
28  end
29
30 endmodule

```

### ghost\_calc.sv

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

```

1  /* ghost_calc
2   * Computes the ghost landing row by scanning downward until collision.
3   */
4
5  'include "../GLOBAL.sv"
6
7  module ghost_calc (
8    input  tetromino_ctrl  t_curr,
9    input  field_t          f,
10   output logic signed ['FIELD_VERTICAL_WIDTH : 0] ghost_y
11  );
12
13 endmodule

```

```

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 >=
39 'FIELD_VERTICAL) begin
40                             collision = 1;
41                         end
42                         // Check Field Collision
43                         else if (ty >= 0 && f.data[ty][tx].data !=
44 'TETROMINO_EMPTY) begin
45                             collision = 1;
46                         end
47                     end
48                 end
49             if (collision) begin
50                 // If collision at 'offset', then 'offset-1' was the last
51             valid position
52                 // But we are looking for the position *before* collision.
53                 // So ghost_y = t_curr.y + (offset - 1)
54                 ghost_y = t_curr.coordinate.y + (offset - 1);
55                 break;
56             end else begin
57                 // If no collision, this position is valid. Keep searching.
58                 ghost_y = t_curr.coordinate.y + offset;
59             end
60         end
61     end
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.*

```
1  /* spin_detector
2   * Detects spin events (T, S, Z, J, L, I) using the last rotation flag,
3   * applied kick, and surrounding field occupancy.
4   */
5  'include "../GLOBAL.sv"
6  module spin_detector (
7    input  tetromino_ctrl  t_ctrl,
8    input  field_t          f,
9    input  logic             last_move_was_rotation,
10   input  logic [2:0]       kick_used,
11
12   output logic            is_t_spin,
13   output logic            is_t_spin_mini,
14   output logic            is_s_spin,
15   output logic            is_z_spin,
16   output logic            is_j_spin,
17   output logic            is_l_spin,
18   output logic            is_i_spin
19 );
20  logic signed ['FIELD_HORIZONTAL_WIDTH:0] cx;
21  logic signed ['FIELD_VERTICAL_WIDTH:0] cy;
22
23  assign cx = t_ctrl.coordinate.x + 1;
24  assign cy = t_ctrl.coordinate.y + 1;
25
26  function automatic logic is_blocked(
27    input logic signed ['FIELD_HORIZONTAL_WIDTH:0] x,
28    input logic signed ['FIELD_VERTICAL_WIDTH:0] y
29  );
30    if (x < 0 || x >= 'FIELD_HORIZONTAL || y < 0 || y >=
'FIELD_VERTICAL)
31      return 1'b1;
32    else
33      return (f.data[y][x] != 'TETROMINO_EMPTY);
34  endfunction
35
36  always_comb begin
37    is_t_spin = 0;
38    is_t_spin_mini = 0;
39    is_s_spin = 0;
40    is_z_spin = 0;
41    is_j_spin = 0;
42    is_l_spin = 0;
43    is_i_spin = 0;
44
45    if (last_move_was_rotation) begin
46      case (t_ctrl.idx.data)
47        'TETROMINO_T_IDX: begin
48          logic [3:0] corners_filled;
49          logic [1:0] front_corners_filled;
50
51          corners_filled[0] = is_blocked(cx - 1, cy - 1);
52          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
58             [1:0];
59                 2'b01: front_corners_filled = {corners_filled
60             [1], corners_filled[3]};
61                 2'b10: front_corners_filled = corners_filled
62             [3:2];
63                 2'b11: front_corners_filled = {corners_filled
64             [0], corners_filled[2]};
65             endcase
66
67             if ((corners_filled[0] + corners_filled[1] +
68 corners_filled[2] + corners_filled[3]) >= 3) begin
69                 if (front_corners_filled == 2'b11) begin
70                     is_t_spin = 1;
71                 end else begin
72                     is_t_spin_mini = 1;
73                 end
74             end
75         end
76
77     'TETROMINO_S_IDX: begin
78         logic [3:0] corners_filled;
79         corners_filled[0] = is_blocked(cx - 1, cy - 1);
80         corners_filled[1] = is_blocked(cx + 1, cy - 1);
81         corners_filled[2] = is_blocked(cx - 1, cy + 1);
82         corners_filled[3] = is_blocked(cx + 1, cy + 1);
83
84         if ((corners_filled[0] + corners_filled[1] +
85 corners_filled[2] + corners_filled[3]) >= 3) begin
86             is_s_spin = 1;
87         end
88     end
89
90     'TETROMINO_Z_IDX: begin
91         logic [3:0] corners_filled;
92         corners_filled[0] = is_blocked(cx - 1, cy - 1);
93         corners_filled[1] = is_blocked(cx + 1, cy - 1);
94         corners_filled[2] = is_blocked(cx - 1, cy + 1);
95         corners_filled[3] = is_blocked(cx + 1, cy + 1);
96
97         if ((corners_filled[0] + corners_filled[1] +
98 corners_filled[2] + corners_filled[3]) >= 3) begin
99             is_z_spin = 1;
100        end
101    end
102
103     'TETROMINO_J_IDX: begin
104         logic [3:0] corners_filled;
105         corners_filled[0] = is_blocked(cx - 1, cy - 1);
106         corners_filled[1] = is_blocked(cx + 1, cy - 1);
107         corners_filled[2] = is_blocked(cx - 1, cy + 1);
108         corners_filled[3] = is_blocked(cx + 1, cy + 1);

```

```

103             if ((corners_filled[0] + corners_filled[1] +
corners_filled[2] + corners_filled[3]) >= 3) begin
104                 is_j_spin = 1;
105             end
106         end
107
108     'TETROMINO_L_IDX: begin
109         logic [3:0] corners_filled;
110         corners_filled[0] = is_blocked(cx - 1, cy - 1);
111         corners_filled[1] = is_blocked(cx + 1, cy - 1);
112         corners_filled[2] = is_blocked(cx - 1, cy + 1);
113         corners_filled[3] = is_blocked(cx + 1, cy + 1);
114
115         if ((corners_filled[0] + corners_filled[1] +
corners_filled[2] + corners_filled[3]) >= 3) begin
116             is_l_spin = 1;
117         end
118     end
119
120     'TETROMINO_I_IDX: begin
121         if (kick_used > 0) begin
122             is_i_spin = 1;
123         end
124     end
125 endcase
126 end
127 end
128 endmodule

```

## bin\_to\_bcd.sv

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

```

1  /* bin_to_bcd
2  * Converts a binary input to packed BCD digits via the double-dabble
3  * method.
4  */
5
6 module bin_to_bcd #(
7     parameter BINARY_WIDTH = 16,
8     parameter BCD_DIGITS = 5
9 )(
10     input  logic [BINARY_WIDTH-1:0] binary,
11     output logic [3:0]               bcd [BCD_DIGITS-1:0]
12 );
13     integer i, j;
14     logic [BINARY_WIDTH + BCD_DIGITS*4 - 1:0] shift_reg;
15
16     always_comb begin
17         shift_reg = {{(BCD_DIGITS*4){1'b0}}, binary};
18
19         for (i = 0; i < BINARY_WIDTH; i = i + 1) begin
20             for (j = 0; j < BCD_DIGITS; j = j + 1) begin
21                 if (shift_reg[BINARY_WIDTH + j*4 +: 4] >= 5)
22                     shift_reg[BINARY_WIDTH + j*4 +: 4] = shift_reg[
BINARY_WIDTH + j*4 +: 4] + 3;

```

```

23         end
24         shift_reg = shift_reg << 1;
25     end
26
27     for (j = 0; j < BCD_DIGITS; j = j + 1) begin
28         bcd[j] = shift_reg[BINARY_WIDTH + j*4 +: 4];
29     end
30 end
31 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.*

```

1  `timescale 1ns / 1ps
2  //
3  // PS2 Keyboard Controller
4  // Processes raw PS2 scan codes and provides make/break status for each
   // key event.
5  // Handles both normal and extended (E0-prefixed) scan codes.
6  //
7
8 module ps2_keyboard(
9     input wire logic clk,
10    input wire logic rst,
11    input wire logic ps2_clk,
12    input wire logic ps2_data,
13    output logic [7:0] current_scan_code,
14    output logic      current_make_break, // 1 = Make (Press), 0 =
Break (Release)
15    output logic      key_event_valid      // Pulse high for one cycle
when a valid key event occurs
16 );
17
18 logic [31:0] keycode;
19 logic [31:0] prev_keycode;
20
21 PS2Receiver ps2_rx (
22     .clk(clk),
23     .kclk(ps2_clk),
24     .kdata(ps2_data),
25     .keycodeout(keycode)
26 );
27
28 // Extract bytes from keycode buffer
29 logic [7:0] new_byte;
30 logic [7:0] prev_byte;
31 logic [7:0] prev_prev_byte;
32
33 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)
76             begin
77                 current_make_break <= 1'b0; // Release
78                 current_scan_code <= new_byte;
79                 event_detected <= 1;
80                 event_pulse_counter <= 0;
81             end
82             // Normal key release: F0 XX
83             else if (prev_byte == 8'hF0) begin
84                 current_make_break <= 1'b0; // Release
85                 current_scan_code <= new_byte;
86                 event_detected <= 1;
87                 event_pulse_counter <= 0;
88             end
89             // Extended key press: E0 XX (but not E0 F0)
90             else if (prev_byte == 8'hE0) begin
91                 current_make_break <= 1'b1; // Press

```

```

91         current_scan_code <= new_byte;
92         event_detected <= 1;
93         event_pulse_counter <= 0;
94     end
95     // Normal key press: just XX (not E0, not F0, and prev
96     wasn't F0)
97     else begin
98         current_make_break <= 1'b1; // Press
99         current_scan_code <= new_byte;
100        event_detected <= 1;
101        event_pulse_counter <= 0;
102    end
103 end
104 end
105
106 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.*

```

1  `timescale 1ns / 1ps
2  //
3  ///////////////////////////////////////////////////////////////////
4  // Company: Digilent Inc.
5  // Engineer: Thomas Kappenman
6  // Create Date: 03/03/2015 09:33:36 PM
7  // Design Name:
8  // Module Name: PS2Receiver
9  // Project Name: Nexys4DDR Keyboard Demo
10 // Target Devices: Nexys4DDR
11 // Tool Versions:
12 // Description: PS2 Receiver module used to shift in keycodes from a
13 // keyboard plugged into the PS2 port
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22
23 module PS2Receiver(
24     input clk,
25     input kclk,
26     input kdata,
27     output [31:0] keycodeout
28 );
29

```

```

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.*

```
1  `include "../GLOBAL.sv"
2
3  module input_manager (
4      input  logic clk,
5      input  logic rst,
6      input  logic tick_game, // 60Hz tick
7
8      // Raw Inputs (Level)
9      input  logic raw_left,
10     input  logic raw_right,
11     input  logic raw_down,
12     input  logic raw_rotate_cw,
13     input  logic raw_rotate_ccw,
14     input  logic raw_drop,
15     input  logic raw_hold,
16
17     // Processed Outputs
18     output logic cmd_left,    // Pulse (DAS)
19     output logic cmd_right,   // Pulse (DAS)
20     output logic cmd_down,    // Pulse (DAS or continuous?)
21     output logic cmd_rotate_cw, // Pulse (One-shot)
22     output logic cmd_rotate_ccw, // Pulse (One-shot)
23     output logic cmd_drop,    // Pulse (One-shot)
24     output logic cmd_hold     // Pulse (One-shot)
25 );
26
27     // Parameters for DAS (Delayed Auto Shift) - modern Tetris feel
28     localparam DAS_DELAY = 6; // Frames before auto-repeat
29     localparam DAS_SPEED = 2; // Frames between repeats
30     localparam SOFT_DROP_SPEED = 0; // 0 = every frame (instant)
31
32     // Timers and edge detectors
33     logic [5:0] timer_left, timer_right, timer_down;
34     logic prev_left, prev_right, prev_down, prev_rotate_cw,
35         prev_rotate_ccw, prev_drop, prev_hold;
36
37     always_ff @(posedge clk) begin
38         if (rst) begin
39             cmd_left <= 0; cmd_right <= 0; cmd_down <= 0;
40             cmd_rotate_cw <= 0; cmd_rotate_ccw <= 0; cmd_drop <= 0;
41             cmd_hold <= 0;
42             timer_left <= 0; timer_right <= 0; timer_down <= 0;
43             prev_left <= 0; prev_right <= 0; prev_down <= 0;
44             prev_rotate_cw <= 0; prev_rotate_ccw <= 0; prev_drop <= 0;
45             prev_hold <= 0;
46         end else begin
47             // Default low
48             cmd_left <= 0;
49             cmd_right <= 0;
50             cmd_down <= 0;
51             cmd_rotate_cw <= 0;
52             cmd_rotate_ccw <= 0;
53             cmd_drop <= 0;
```

```

51     cmd_hold <= 0;

52
53 // --- LEFT ---
54 if (raw_left) begin
55     if (!prev_left) begin
56         // Initial Press
57         cmd_left <= 1;
58         timer_left <= 0;
59     end else if (tick_game) begin
60         // Holding
61         if (timer_left < DAS_DELAY) begin
62             timer_left <= timer_left + 1;
63         end else begin
64             // Auto Repeat
65             if (timer_left >= DAS_DELAY + DAS_SPEED) begin
66                 cmd_left <= 1;
67                 timer_left <= DAS_DELAY; // Reset to delay base
68             end else begin
69                 timer_left <= timer_left + 1;
70             end
71         end
72     end
73 end else begin
74     timer_left <= 0;
75 end
76 prev_left <= raw_left;
77
78 // --- RIGHT ---
79 if (raw_right) begin
80     if (!prev_right) begin
81         cmd_right <= 1;
82         timer_right <= 0;
83     end else if (tick_game) begin
84         if (timer_right < DAS_DELAY) begin
85             timer_right <= timer_right + 1;
86         end else begin
87             if (timer_right >= DAS_DELAY + DAS_SPEED) begin
88                 cmd_right <= 1;
89                 timer_right <= DAS_DELAY;
90             end else begin
91                 timer_right <= timer_right + 1;
92             end
93         end
94     end
95 end else begin
96     timer_right <= 0;
97 end
98 prev_right <= raw_right;
99
100 if (raw_down) begin
101    if (!prev_down) begin
102        cmd_down <= 1;
103        timer_down <= 0;
104    end else if (tick_game) begin
105        if (timer_down >= SOFT_DROP_SPEED) begin
106            cmd_down <= 1;
107            timer_down <= 0;
108        end else begin

```

```

109         timer_down <= timer_down + 1;
110     end
111 end
112 end else begin
113     timer_down <= 0;
114 end
115 prev_down <= raw_down;
116
117 // --- ROTATE CW (One Shot) ---
118 if (raw_rotate_cw && !prev_rotate_cw) begin
119     cmd_rotate_cw <= 1;
120 end
121 prev_rotate_cw <= raw_rotate_cw;
122
123 // --- ROTATE CCW (One Shot) ---
124 if (raw_rotate_ccw && !prev_rotate_ccw) begin
125     cmd_rotate_ccw <= 1;
126 end
127 prev_rotate_ccw <= raw_rotate_ccw;
128
129 // --- DROP (One Shot) ---
130 if (raw_drop && !prev_drop) begin
131     cmd_drop <= 1;
132 end
133 prev_drop <= raw_drop;
134
135 // --- HOLD (One Shot) ---
136 if (raw_hold && !prev_hold) begin
137     cmd_hold <= 1;
138 end
139 prev_hold <= raw_hold;
140 end
141 end
142
143 endmodule

```

## debouncer.sv

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

```

1 'timescale 1ns / 1ps
2 //
3 ///////////////////////////////////////////////////////////////////
4 // PS2 Signal Debouncer
5 // Based on Digilent Nexys-A7-100T-Keyboard reference implementation
6 // Uses fast debouncing (~20 cycles) suitable for PS2 protocol timing
7 ///////////////////////////////////////////////////////////////////
8 module debouncer(
9     input clk,
10    input I0,
11    input I1,
12    output reg OO,

```

```

13     output reg 01
14 );
15
16 // Use 5-bit counters for fast PS2 debouncing (~20 cycles)
17 // At 50MHz: 20 cycles = 400ns, well within PS2 timing requirements
18 reg [4:0] cnt0 = 0, cnt1 = 0;
19 reg Iv0 = 0, Iv1 = 0;
20
21 // Initialize outputs to match expected idle state (PS2 lines idle
22 // high)
23 initial begin
24     00 = 1'b1;
25     01 = 1'b1;
26 end
27
28 localparam CNT_MAX = 19; // Fast debounce for PS2 signals
29
30 always @ (posedge clk) begin
31     // Debounce IO
32     if (IO == Iv0) begin
33         if (cnt0 == CNT_MAX)
34             00 <= IO;
35         else
36             cnt0 <= cnt0 + 1;
37     end else begin
38         cnt0 <= 5'b00000;
39         Iv0 <= IO;
40     end
41
42     // Debounce I1
43     if (I1 == Iv1) begin
44         if (cnt1 == CNT_MAX)
45             01 <= I1;
46         else
47             cnt1 <= cnt1 + 1;
48     end else begin
49         cnt1 <= 5'b00000;
50         Iv1 <= I1;
51     end
52 end
53 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.*

```

1 `timescale 1ns / 1ps
2 `include "GLOBAL.sv"
3
4 /* tb_game_control
5  * Integration testbench covering spawn, movement, rotation, hold, drop
6  * , and ghost.
7 */

```

```

7 module tb_game_control;
8
9 // Inputs
10 logic clk;
11 logic rst;
12 logic tick_game;
13 logic key_left;
14 logic key_right;
15 logic key_down;
16 logic key_rotate_cw;
17 logic key_rotate_ccw;
18 logic key_drop;
19 logic key_hold;
20 logic key_drop_held;
21
22 field_t display;
23 logic [31:0] score;
24 logic game_over;
25 tetromino_ctrl t_next_disp;
26 tetromino_ctrl t_hold_disp;
27 logic hold_used_out;
28 logic [3:0] current_level_out;
29 logic signed ['FIELD_VERTICAL_WIDTH : 0] ghost_y;
30 tetromino_ctrl t_curr_out;
31 logic [7:0] total_lines_cleared_out;
32
33 int pass_count = 0;
34 int fail_count = 0;
35
36 // Instantiate the Unit Under Test (UUT)
37 game_control uut (
38     .clk(clk),
39     .rst(rst),
40     .tick_game(tick_game),
41     .key_left(key_left),
42     .key_right(key_right),
43     .key_down(key_down),
44     .key_rotate_cw(key_rotate_cw),
45     .key_rotate_ccw(key_rotate_ccw),
46     .key_drop(key_drop),
47     .key_hold(key_hold),
48     .key_drop_held(key_drop_held),
49     .display(display),
50     .score(score),
51     .game_over(game_over),
52     .t_next_disp(t_next_disp),
53     .t_hold_disp(t_hold_disp),
54     .hold_used_out(hold_used_out),
55     .current_level_out(current_level_out),
56     .ghost_y(ghost_y),
57     .t_curr_out(t_curr_out),
58     .total_lines_cleared_out(total_lines_cleared_out)
59 );
60
61 // Clock Generation
62 always #5 clk = ~clk; // 100MHz clock (10ns period)
63
64 // 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);

```

```

123         key_drop_hold = 0;
124         @(posedge clk);
125     end
126 endtask
127
128 task press_hold;
129 begin
130     key_hold = 1;
131     @(posedge clk);
132     key_hold = 0;
133     @(posedge clk);
134 end
135 endtask
136
137 task press_down;
138 begin
139     key_down = 1;
140     @(posedge clk);
141     key_down = 0;
142     @(posedge clk);
143 end
144 endtask
145
146 initial begin
147     // Initialize Inputs
148     clk = 0;
149     rst = 1;
150     tick_game = 0;
151     key_left = 0;
152     key_right = 0;
153     key_down = 0;
154     key_rotate_cw = 0;
155     key_rotate_ccw = 0;
156     key_drop = 0;
157     key_hold = 0;
158     key_drop_hold = 0;
159
160     $display("== Game Control Testbench ==");
161     $display("Testing game mechanics including HOLD feature\n");
162
163     // Wait 100 ns for global reset to finish
164     #100;
165     rst = 0;
166
167     // Wait for Generator to spawn first piece
168     repeat(20) @(posedge clk);
169
170     //
171 =====
172     // Test 1: Initialization
173     //
174 =====
175     $display("Test 1: Initialization");
176     if (game_over == 0) begin
177         $display("  PASS: Game not over at start");
178         pass_count++;
179     end else begin
180         $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
184 .idx.data);
185         pass_count++;
186     end else begin
187         $display(" FAIL: No piece spawned");
188         fail_count++;
189     end
190
191     //
192 =====
193     // Test 2: Movement
194     //
195 =====
196     $display("\nTest 2: Movement");
197     begin
198         logic signed ['FIELD_HORIZONTAL_WIDTH:0] start_x;
199         start_x = t_curr_out.coordinate.x;
200
201         // Move Left
202         press_left();
203         repeat(5) @ (posedge clk);
204
205         if (t_curr_out.coordinate.x == start_x - 1) begin
206             $display(" PASS: Moved left (x: %d -> %d)", start_x,
207 t_curr_out.coordinate.x);
208             pass_count++;
209         end else begin
210             $display(" FAIL: Left move failed (x: %d -> %d)", start_x,
211 t_curr_out.coordinate.x);
212             fail_count++;
213         end
214
215         // Move Right twice
216         press_right();
217         repeat(3) @ (posedge clk);
218         press_right();
219         repeat(3) @ (posedge clk);
220
221         if (t_curr_out.coordinate.x == start_x + 1) begin
222             $display(" PASS: Moved right twice (x: %d)", t_curr_out.
223 coordinate.x);
224             pass_count++;
225         end else begin
226             $display(" INFO: Right move result (x: %d)", t_curr_out.
227 coordinate.x);
228         end
229     end
230
231     //
232 =====
233     // Test 3: Rotation
234     //
235 =====
236     $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         // 0-piece doesn't rotate visibly, but rotation state still
236         // changes
237         $display(" INFO: Rotation state: %d -> %d", start_rot,
238             t_curr_out.rotation);
239         pass_count++; // Rotation test is informational
240     end
241
242     //
243     =====
244     // Test 4: Hold Feature (First Hold - Empty)
245     //
246     =====
247     $display("\nTest 4: Hold Feature - First Hold");
248     begin
249
250         tetromino_idx_t first_piece_idx;
251         first_piece_idx = t_curr_out.idx;
252
253         if (t_hold_disp.idx.data == 'TETROMINO_EMPTY) begin
254             $display(" INFO: Hold slot is empty before hold");
255         end
256
257         press_hold();
258         repeat(20) @ (posedge clk);
259
260         if (t_hold_disp.idx.data == first_piece_idx.data) begin
261             $display(" PASS: First piece stored in hold (idx=%d)", t_hold_disp.idx.data);
262             pass_count++;
263         end else begin
264             $display(" FAIL: Hold did not store piece correctly");
265             fail_count++;
266         end
267     end
268     //
269     =====
270     // Test 5: Hold Available After New Piece
271     //
272     =====
273     $display("\nTest 5: Hold Available After New Piece");
274     begin
275
276         tetromino_idx_t piece_before_second_hold;
277         piece_before_second_hold = t_curr_out.idx;
278         begin
279             tetromino_idx_t hold_before_second_hold;
280             hold_before_second_hold = t_hold_disp.idx;
281
282             press_hold();
283             repeat(20) @ (posedge clk);

```

```

279         // After second hold, pieces should swap: current becomes
280         // previous hold, hold becomes previous current
281         if (t_curr_out.idx.data == hold_before_second_hold.data &&
282             t_hold_disp.idx.data == piece_before_second_hold.data)
283     begin
284         $display("  PASS: Second hold swaps current and hold
285 pieces");
286         pass_count++;
287     end else begin
288         $display("  FAIL: Second hold did not swap as expected
289 (curr=%d hold=%d)", t_curr_out.idx.data, t_hold_disp.idx.data);
290         fail_count++;
291     end
292     end
293     //
294     =====
295     // Test 6: Hard Drop & Lock
296     //
297     =====
298     $display("\nTest 6: Hard Drop & Lock");
299     press_drop();
300     repeat(30) @(posedge clk); // Wait for drop and lockout
301
302     if (game_over == 0) begin
303         $display("  PASS: Game continues after drop");
304         pass_count++;
305     end else begin
306         $display("  FAIL: Game ended unexpectedly");
307         fail_count++;
308     end
309
310     //
311     =====
312     // Test 7: Ghost Y Position
313     //
314     =====
315     $display("\nTest 7: Ghost Position");
316
317     if (ghost_y >= t_curr_out.coordinate.y) begin
318         $display("  PASS: Ghost Y (%d) >= Current Y (%d)", ghost_y,
319         t_curr_out.coordinate.y);
320         pass_count++;
321     end else begin
322         $display("  FAIL: Ghost Y invalid");
323         fail_count++;
324     end
325
326     //
327     =====
328     // Summary
329     //
330     =====
331     $display("\n==== Test Summary ===");
332     $display("Passed: %0d", pass_count);
333     $display("Failed: %0d", fail_count);
334     $display("Simulation Finished");

```

```

326     $finish;
327   end
328
329 endmodule

```

### Simulation Log:

```

1 source tb_game_control.tcl
2 # set curr_wave [current_wave_config]
3 # if { [string length $curr_wave] == 0 } {
4 #   if { [llength [get_objects]] > 0} {
5 #     add_wave /
6 #     set_property needs_save false [current_wave_config]
7 #   } else {
8 #     send_msg_id Add_Wave-1 WARNING "No top level signals found.
9 #     Simulator will start without a wave window. If you want to open a
10#      wave window go to 'File->New Waveform Configuration' or type '
11#      'create_wave_config' in the TCL console."
12#   }
13# }
14# run 1000ns
15==== Game Control Testbench ====
16Testing game mechanics including HOLD feature
17
18Test 1: Initialization
19  PASS: Game not over at start
20  PASS: First piece spawned (idx=1)
21
22Test 2: Movement
23  PASS: Moved left (x: 3 -> 2)
24  PASS: Moved right twice (x: 4)
25
26Test 3: Rotation
27  INFO: Rotation state: 0 -> 1
28
29Test 4: Hold Feature - First Hold
30  INFO: Hold slot is empty before hold
31  PASS: First piece stored in hold (idx=1)
32
33Test 5: Hold Available After New Piece
34  INFO: [USF-XSim-96] XSim completed. Design snapshot ,
35    'tb_game_control_behav' loaded.
36  INFO: [USF-XSim-97] XSim simulation ran for 1000ns
37  launch_simulation: Time (s): cpu = 00:00:12 ; elapsed = 00:00:09 .
38    Memory (MB): peak = 8644.832 ; gain = 15.008 ; free physical =
39    1012901 ; free virtual = 1028180
40 run all
41  PASS: Second hold swaps current and hold pieces
42
43Test 6: Hard Drop & Lock
44  PASS: Game continues after drop
45
46Test 7: Ghost Position
47  PASS: Ghost Y ( 20 ) >= Current Y ( 20 )
48
49==== Test Summary ====
50 Passed: 9
51 Failed: 0
52 Simulation Finished

```

```

47 $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.*

```

1  `timescale 1ns / 1ps
2  `include "GLOBAL.sv"
3
4  /* tb_input_manager
5   * Verifies one-shot and DAS behavior for controller inputs.
6   */
7  module tb_input_manager;
8
9    logic clk;
10   logic rst;
11   logic tick_game;
12   logic raw_left, raw_right, raw_down, raw_rotate_cw, raw_rotate_ccw,
13   raw_drop, raw_hold;
14   logic cmd_left, cmd_right, cmd_down, cmd_rotate_cw, cmd_rotate_ccw,
15   cmd_drop, cmd_hold;
16
17   int pass_count = 0;
18   int fail_count = 0;
19
20   input_manager uut (
21     .clk(clk),
22     .rst(rst),
23     .tick_game(tick_game),
24     .raw_left(raw_left),
25     .raw_right(raw_right),
26     .raw_down(raw_down),
27     .raw_rotate_cw(raw_rotate_cw),
28     .raw_rotate_ccw(raw_rotate_ccw),
29     .raw_drop(raw_drop),
30     .raw_hold(raw_hold),
31     .cmd_left(cmd_left),
32     .cmd_right(cmd_right),
33     .cmd_down(cmd_down),
34     .cmd_rotate_cw(cmd_rotate_cw),
35     .cmd_rotate_ccw(cmd_rotate_ccw),
36     .cmd_drop(cmd_drop),
37     .cmd_hold(cmd_hold)
38   );
39
40   initial begin
41     clk = 0; rst = 1; tick_game = 0;
42     raw_left = 0; raw_right = 0; raw_down = 0;
43     raw_rotate_cw = 0; raw_rotate_ccw = 0; raw_drop = 0; raw_hold =
44     0;
45
46     $display("==> Input Manager Testbench ==>");
47     $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
112 holding");
113         pass_count++;
114     end else begin
115         $display(" FAIL: Rotate CCW re-triggered while holding");
116         fail_count++;
117     end
118
119     @(negedge clk);
120     raw_rotate_ccw = 0;
121     @(posedge clk);
122
123     $display("\nTest 2: Drop One-Shot");
124     @(negedge clk); // Setup time before clock edge
125     raw_drop = 1;
126     @(posedge clk);
127     #1;
128     if (cmd_drop) begin
129         $display(" PASS: Drop Triggered on press");
130         pass_count++;
131     end else begin
132         $display(" FAIL: Drop did not trigger");
133         fail_count++;
134     end
135
136     @(posedge clk);
137     #1;
138     if (!cmd_drop) begin
139         $display(" PASS: Drop Pulse Ended");
140         pass_count++;
141     end else begin
142         $display(" FAIL: Drop Pulse lasted too long");
143         fail_count++;
144     end
145
146     @(negedge clk);
147     raw_drop = 0;
148     @(posedge clk);
149
150     $display("\nTest 3: Hold One-Shot");
151     @(negedge clk); // Setup time before clock edge
152     raw_hold = 1;
153     @(posedge clk);
154     #1;
155     if (cmd_hold) begin
156         $display(" PASS: Hold Triggered on press");
157         pass_count++;
158     end else begin
159         $display(" FAIL: Hold did not trigger");
160         fail_count++;
161     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",
210             cmd_left);
211         fail_count++;
212     end
213
214     repeat(15) begin
215         tick_game = 1; @(posedge clk);
216         tick_game = 0; @(posedge clk);
217     end

```

```

218     #1;
219     if (!cmd_left) begin
220         $display("  PASS: No trigger during DAS delay (15 frames)")
221     ;
222         pass_count++;
223     end else begin
224         $display("  FAIL: Premature trigger during DAS delay");
225         fail_count++;
226     end
227
228     tick_game = 1; @(posedge clk);
229     tick_game = 0; @(posedge clk);
230
231     repeat(5) begin
232         tick_game = 1; @(posedge clk);
233         tick_game = 0; @(posedge clk);
234     end
235
236     tick_game = 1;
237     @(posedge clk);
238     @(negedge clk);
239
240     if (cmd_left) begin
241         $display("  PASS: Left DAS Auto-Repeat Triggered");
242         pass_count++;
243     end else begin
244         $display("  FAIL: Left DAS missing after delay+speed");
245         fail_count++;
246     end
247
248     tick_game = 0;
249     raw_left = 0;
250     @(posedge clk);
251
252     $display("\nTest 5: Down Fast Repeat (Soft Drop)");
253     @(negedge clk); // Setup time
254     raw_down = 1;
255     @(posedge clk);
256     #1;
257     if (cmd_down) begin
258         $display("  PASS: Down Initial trigger");
259         pass_count++;
260     end else begin
261         $display("  FAIL: Down Initial missing");
262         fail_count++;
263     end
264
265     tick_game = 1; @(posedge clk); tick_game = 0; @(posedge clk);
266     tick_game = 1; @(posedge clk); tick_game = 0; @(posedge clk);
267     tick_game = 1; @(posedge clk);
268     #1;
269
270     if (cmd_down) begin
271         $display("  PASS: Down Fast Repeat working (every ~2 frames
272         )");
273         pass_count++;
274     end else begin
275         $display("  FAIL: Down Fast Repeat missing");

```

```

274         fail_count++;
275     end
276
277     tick_game = 0;
278     raw_down = 0;
279     @(posedge clk);
280
281     $display("\n==== Test Summary ====");
282     $display("Passed: %0d", pass_count);
283     $display("Failed: %0d", fail_count);
284     $display("Simulation Finished");
285     $finish;
286   end
287 endmodule

```

### Simulation Log:

```

1 source tb_input_manager.tcl
2 # set curr_wave [current_wave_config]
3 # if { [string length $curr_wave] == 0 } {
4 #   if { [llength [get_objects]] > 0} {
5 #     add_wave /
6 #     set_property needs_save false [current_wave_config]
7 #   } else {
8 #     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."
9 #   }
10 # }
11 # run 1000ns
12 === Input Manager Testbench ===
13 Testing DAS and One-Shot behavior
14
15 Test 1: Rotate CW One-Shot
16 PASS: Rotate CW Triggered on press
17 PASS: Rotate CW Pulse Ended after one cycle
18 PASS: Rotate CW did not re-trigger while holding
19
20 Test 1b: Rotate CCW One-Shot
21 PASS: Rotate CCW Triggered on press
22 PASS: Rotate CCW Pulse Ended after one cycle
23 PASS: Rotate CCW did not re-trigger while holding
24
25 Test 2: Drop One-Shot
26 PASS: Drop Triggered on press
27 PASS: Drop Pulse Ended
28
29 Test 3: Hold One-Shot
30 PASS: Hold Triggered on press
31 PASS: Hold Pulse Ended after one cycle
32 PASS: Hold did not re-trigger while holding
33 PASS: Hold Triggered again after release
34
35 Test 4: Left DAS (Delayed Auto Shift)
36 PASS: Left Initial Move on press
37 PASS: No trigger during DAS delay (15 frames)
38 PASS: Left DAS Auto-Repeat Triggered
39

```

```

40 Test 5: Down Fast Repeat (Soft Drop)
41 PASS: Down Initial trigger
42 PASS: Down Fast Repeat working (every ~2 frames)
43
44 === Test Summary ===
45 Passed: 17
46 Failed: 0
47 Simulation Finished
48 $finish called at time : 975 ns : File "/home/alkhalmr/gtprj/ag-tetris/
  test/tb_input_manager.sv" Line 313
49 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.*

```

1  `timescale 1ns / 1ps
2  `include "GLOBAL.sv"
3
4  /* tb_rotate_tetromino
5   * Exercises rotate_tetromino CW/CCW wrap-around behavior.
6   */
7  module tb_rotate_tetromino;
8
9    logic clk;
10   logic enable;
11   logic clockwise;
12   logic success;
13   logic done;
14   tetromino_ctrl t_in;
15   tetromino_ctrl t_out;
16
17   rotate_tetromino uut (
18     .clk(clk),
19     .enable(enable),
20     .clockwise(clockwise),
21     .t_in(t_in),
22     .t_out(t_out),
23     .success(success),
24     .done(done)
25   );
26
27   always #5 clk = ~clk;
28
29   initial begin
30     clk = 0;
31     enable = 0;
32     clockwise = 1'b1;
33     t_in.idx.data = 'TETROMINO_T_IDX;
34     t_in.rotation = 0;
35     t_in.coordinate.x = 3;
36     t_in.coordinate.y = 0;
37     t_in.tetromino.data = '{default: '{default: 4'b0000}};
38
39     #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

```

```

98         $display("FAIL: CCW expected 0, got %0d", t_out.rotation);
99
100        $display("Simulation finished");
101        $finish;
102    end
103
104 endmodule

```

### Simulation Log:

```

1 source tb_rotate_tetromino.tcl
2 # set curr_wave [current_wave_config]
3 # if { [string length $curr_wave] == 0 } {
4 #     if { [llength [get_objects]] > 0} {
5 #         add_wave /
6 #         set_property needs_save false [current_wave_config]
7 #     } else {
8 #         send_msg_id Add_Wave-1 WARNING "No top level signals found.
9 #             Simulator will start without a wave window. If you want to open a
10#             wave window go to 'File->New Waveform Configuration' or type '
11#             'create_wave_config' in the TCL console."
12#     }
13# }
14# run 1000ns
15==> Testing rotate_tetromino CW/CCW ===
16Test 1: CW 0 -> 1
17PASS: CW rotation 0 -> 1
18Test 2: CW wrap 3 -> 0
19PASS: CW rotation 3 -> 0
20Test 3: CCW 0 -> 3
21PASS: CCW rotation 0 -> 3
22Test 4: CCW 1 -> 0
23PASS: CCW rotation 1 -> 0
24Simulation 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.*

```

1 `timescale 1ns / 1ps
2 `include "GLOBAL.sv"
3
4 /* tb_hold_feature
5  * End-to-end checks of hold behavior: initial empty, store, swap,
6  * lockout, reset.
7 */
8 module tb_hold_feature;
9
10 // Clock and Reset
11 logic clk;
12 logic rst;
13 logic tick_game;

```

```

13
14 // Inputs
15 logic key_left, key_right, key_down, key_rotate_cw, key_rotate_ccw,
16 key_drop, key_hold;
17 logic key_drop_held;
18
19 // Outputs
20 field_t display;
21 logic [31:0] score;
22 logic game_over;
23 tetromino_ctrl t_next_disp;
24 tetromino_ctrl t_hold_disp;
25 logic hold_used_out;
26 logic [3:0] current_level_out;
27 logic signed ['FIELD_VERTICAL_WIDTH : 0] ghost_y;
28 tetromino_ctrl t_curr_out;
29 logic [7:0] total_lines_cleared_out;
30
31 int pass_count = 0;
32 int fail_count = 0;
33
34 // UUT
35 game_control uut (
36     .clk(clk),
37     .rst(rst),
38     .tick_game(tick_game),
39     .key_left(key_left),
40     .key_right(key_right),
41     .key_down(key_down),
42     .key_rotate_cw(key_rotate_cw),
43     .key_rotate_ccw(key_rotate_ccw),
44     .key_drop(key_drop),
45     .key_hold(key_hold),
46     .key_drop_held(key_drop_held),
47     .display(display),
48     .score(score),
49     .game_over(game_over),
50     .t_next_disp(t_next_disp),
51     .t_hold_disp(t_hold_disp),
52     .hold_used_out(hold_used_out),
53     .current_level_out(current_level_out),
54     .ghost_y(ghost_y),
55     .t_curr_out(t_curr_out),
56     .total_lines_cleared_out(total_lines_cleared_out)
57 );
58
59 always #5 clk = ~clk;
60
61 // Piece name helper
62 function string piece_name(input [2:0] idx);
63     case(idx)
64         3'b000: return "I";
65         3'b001: return "J";
66         3'b010: return "L";
67         3'b011: return "O";
68         3'b100: return "S";
69         3'b101: return "T";
70         3'b110: return "Z";

```

```

70         3'b111: return "EMPTY";
71         default: return "?";
72     endcase
73 endfunction
74
75 // Wait for state machine to settle
76 task wait_settle();
77     repeat(20) @(posedge clk);
78 endtask
79
80 // Press hold key
81 task do_hold();
82     key_hold = 1;
83     @(posedge clk);
84     key_hold = 0;
85     wait_settle();
86 endtask
87
88 // Press drop and wait for next piece
89 task do_drop();
90     key_drop = 1;
91     key_drop_held = 1;
92     @(posedge clk);
93     key_drop = 0;
94     // Wait longer for HARD_DROP (~20) + CLEAN (~10) + transitions
95     repeat(50) @(posedge clk);
96     key_drop_held = 0;
97     // Wait for DROP_LOCKOUT -> GEN -> GEN_WAIT -> IDLE
98     repeat(30) @(posedge clk);
99 endtask
100
101 initial begin
102     // Initialize
103     clk = 0;
104     rst = 1;
105     tick_game = 0;
106     key_left = 0; key_right = 0; key_down = 0;
107     key_rotate_cw = 0; key_rotate_ccw = 0; key_drop = 0; key_hold =
0;
108     key_drop_held = 0;
109
110     $display("=====");
111     $display("      HOLD FEATURE TESTBENCH");
112     $display("=====\\n");
113
114     // Release reset
115     #100;
116     rst = 0;
117     wait_settle();
118
119     //
=====//
120     // Test 1: Initial State (Hold Empty)
121     //
=====
122     $display("Test 1: Initial State");
123
124     if (t_hold_disp.idx.data == 'TETROMINO_EMPTY) begin

```

```

125         $display("  PASS: Hold slot is empty at start");
126         pass_count++;
127     end else begin
128         $display("  FAIL: Hold slot should be empty at start");
129         fail_count++;
130     end
131
132     if (hold_used_out == 0) begin
133         $display("  PASS: hold_used is false at start");
134         pass_count++;
135     end else begin
136         $display("  FAIL: hold_used should be false at start");
137         fail_count++;
138     end
139
140     //
141 =====
142 // Test 2: First Hold (Empty Slot)
143 //
144 =====
145 $display("\nTest 2: First Hold (Empty -> Store)");
146
147 // Capture first piece
148 logic [2:0] first_piece;
149 first_piece = t_curr_out.idx.data;
150 $display("  Current piece before hold: %s (%d)", piece_name(
151 first_piece), first_piece);
152
153 do_hold();
154
155 $display("  After hold: curr=%s, hold=%s, hold_used=%b",
156             piece_name(t_curr_out.idx.data),
157             piece_name(t_hold_disp.idx.data),
158             hold_used_out);
159
160 if (t_hold_disp.idx.data == first_piece) begin
161     $display("  PASS: First piece stored in hold");
162     pass_count++;
163 end else begin
164     $display("  FAIL: First piece not stored correctly");
165     fail_count++;
166 end
167
168 if (hold_used_out == 1) begin
169     $display("  PASS: hold_used set after first hold");
170     pass_count++;
171 end else begin
172     $display("  FAIL: hold_used not set after first hold");
173     fail_count++;
174 end
175
176 //
177 =====
178 // Test 3: Hold Lockout (Cannot Hold Twice)
179 //
180 =====
181 $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 =====
197 // Test 4: Drop and Check Hold Reset
198 // =====
199
200     $display("\nTest 4: Hold Reset After Piece Placement");
201
202     do_drop();
203
204     if (hold_used_out == 0) begin
205         $display("  PASS: hold_used reset after drop");
206         pass_count++;
207     end else begin
208         $display("  FAIL: hold_used not reset after drop");
209         fail_count++;
210     end
211
212     //
213 =====
214 // Test 5: Hold Swap
215 // =====
216
217     $display("\nTest 5: Hold Swap");
218
219     logic [2:0] curr_before_swap, hold_before_swap;
220     curr_before_swap = t_curr_out.idx.data;
221     hold_before_swap = t_hold_disp.idx.data;
222
223     $display("  Before swap: curr=%s, hold=%s",
224             piece_name(curr_before_swap), piece_name(
225             hold_before_swap));
226
227     do_hold();
228
229     $display("  After swap:  curr=%s, hold=%s",
230             piece_name(t_curr_out.idx.data), piece_name(
231             t_hold_disp.idx.data));
232
233     if (t_curr_out.idx.data == hold_before_swap &&
234         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     =====
238     // Test 6: Multiple Drops and Holds
239     //
240     =====
241     $display("\nTest 6: Multiple Game Cycles");
242
243     repeat(3) begin
244         // Drop current piece
245         do_drop();
246
247         // Verify hold is available
248         if (hold_used_out != 0) begin
249             $display("  FAIL: hold_used not reset after drop");
250             fail_count++;
251         end
252
253         // Swap with hold
254         do_hold();
255
256         // Verify lockout
257         if (hold_used_out != 1) begin
258             $display("  FAIL: hold_used not set after hold");
259             fail_count++;
260         end
261     end
262
263     $display("  PASS: Multiple game cycles completed");
264     pass_count++;
265
266     //
267     =====
268     // Test 7: Hold Piece Reset Position
269     //
270     =====
271     $display("\nTest 7: Swapped Piece Position Reset");
272
273     // After swap, piece should be at spawn position
274     if (t_curr_out.coordinate.x == 3 && t_curr_out.coordinate.y ==
275 0) begin
276         $display("  PASS: Swapped piece at spawn position (3, 0)");
277         pass_count++;
278     end else begin
279         $display("  INFO: Piece at (%d, %d) - may vary based on
state",
280                 t_curr_out.coordinate.x, t_curr_out.coordinate.y);
281     end
282
283     //
284     =====
285     // Summary

```

```

280      // =====
281      $display("\n===== TEST SUMMARY");
282      $display("===== Passed: %0d", pass_count);
283      $display("===== Failed: %0d", fail_count);
284      if (fail_count == 0)
285          $display("ALL TESTS PASSED!");
286      else
287          $display("SOME TESTS FAILED!");
288      $display("=====\\n");
289
290      $finish;
291
292 end
293
294
295 endmodule

```

### Simulation Log:

```

1 =====
2      HOLD FEATURE TESTBENCH
3 =====
4
5 Test 1: Initial State
6      PASS: Hold slot is empty at start
7      PASS: hold_used is false at start
8
9 Test 2: First Hold (Empty -> Store)
10     Current piece before hold: J (1)
11     After hold: curr=0, hold=J, hold_used=1
12     PASS: First piece stored in hold
13     PASS: hold_used set after first hold
14
15 Test 3: Hold Lockout (Cannot Hold Twice Per Piece)
16     Attempting second hold while hold_used=1...
17     PASS: Lockout prevented second hold
18
19 Test 4: Hold Reset After Piece Placement
20 INFO: [USF-XSim-96] XSim completed. Design snapshot ,
21     tb_hold_feature_behav' loaded.
22 INFO: [USF-XSim-97] XSim simulation ran for 1000ns
23 launch_simulation: Time (s): cpu = 00:00:24 ; elapsed = 00:00:12 .
24     Memory (MB): peak = 8409.602 ; gain = 757.141 ; free physical =
25     1013473 ; free virtual = 1028701
26 run 5 s
27     PASS: hold_used reset after drop
28
29 Test 5: Hold Swap
30     Before swap: curr=I, hold=J
31     After swap: curr=J, hold=I
32     PASS: Swap successful
33
34 Test 6: Multiple Game Cycles
35     PASS: Multiple game cycles completed
36
37 Test 7: Swapped Piece Position Reset
38     PASS: Swapped piece at spawn position (3, 0)

```

```

37 =====
38      TEST SUMMARY
39 =====
40 Passed: 9
41 Failed: 0
42 ALL TESTS PASSED!
43 =====
44
45 $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.

```

1  `timescale 1ns / 1ps
2  `include "GLOBAL.sv"
3
4  /* tb_generate_tetromino
5   * Sanity tests for 7-bag generation: valid indices and non-empty
6   * shapes.
7   */
8  module tb_generate_tetromino;
9
10    logic clk;
11    logic rst;
12    logic enable;
13    tetromino_ctrl t_curr_out;
14    tetromino_ctrl t_next_out;
15
16    generate_tetromino uut (
17      .clk(clk),
18      .rst(rst),
19      .enable(enable),
20      .t_out(t_curr_out),
21      .t_next_out(t_next_out)
22    );
23
24    always #5 clk = ~clk;
25
26    initial begin
27      clk = 0;
28      rst = 1;
29      enable = 0;
30
31      #20 rst = 0;
32
33      $display("== Test 1: Initial Generation ==");
34      enable = 1;
35      @(posedge clk);
36      repeat(2) @(posedge clk);
37      @(posedge clk);
38
39      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",
42 t_curr_out.idx.data);
43
44     if (t_next_out.idx.data >= 'TETROMINO_I_IDX && t_next_out.idx.
45 data <= 'TETROMINO_Z_IDX)
46         $display("PASS: Next piece index valid: %d", t_next_out.idx
47 .data);
48     else
49         $display("FAIL: Invalid next piece index: %d", t_next_out.
50 idx.data);
51
52     enable = 0;
53     @(posedge clk);
54
55     $display("\n==== Test 2: Sequential Generation ===");
56     logic [2:0] prev_idx;
57     prev_idx = t_curr_out.idx.data;
58
59     // Generate 10 pieces and verify they're all valid
60     for (int i = 0; i < 10; i++) begin
61         enable = 1;
62         @(posedge clk);
63         repeat(2) @(posedge clk);
64         @(posedge clk);
65         enable = 0;
66         @(posedge clk);
67
68         if (t_curr_out.idx.data >= 'TETROMINO_I_IDX && t_curr_out.
69 idx.data <= 'TETROMINO_Z_IDX)
70             $display("PASS: Piece %d - Valid index: %d", i,
71 t_curr_out.idx.data);
72         else
73             $display("FAIL: Piece %d - Invalid index: %d", i,
74 t_curr_out.idx.data);
75     end
76
77     $display("\n==== Test 3: Verify Tetromino Data ===");
78     // Check that generated piece has valid shape data
79     logic has_blocks;
80     has_blocks = 0;
81     for (int rot = 0; rot < 4; rot++) begin
82         for (int r = 0; r < 4; r++) begin
83             for (int c = 0; c < 4; c++) begin
84                 if (t_curr_out.tetromino.data[rot][r][c])
85                     has_blocks = 1;
86             end
87         end
88     end
89
90     if (has_blocks)
91         $display("PASS: Generated piece has valid shape data");
92     else
93         $display("FAIL: Generated piece has no blocks");
94
95     $display("\nSimulation Finished");
96     $finish;
97 end

```

```

91
92 endmodule

```

**Simulation Log:**

```

1 source tb_generate_tetromino.tcl
2 # set curr_wave [current_wave_config]
3 # if { [string length $curr_wave] == 0 } {
4 #     if { [llength [get_objects]] > 0} {
5 #         add_wave /
6 #         set_property needs_save false [current_wave_config]
7 #     } else {
8 #         send_msg_id Add_Wave-1 WARNING "No top level signals found.
9 #             Simulator will start without a wave window. If you want to open a
10 #             wave window go to 'File->New Waveform Configuration' or type '
11 #             create_wave_config' in the TCL console."
12 #
13 # run 1000ns
14 === Test 1: Initial Generation ===
15 PASS: Current piece index valid: 3
16 PASS: Next piece index valid: 0
17
18 === Test 2: Sequential Generation ===
19 PASS: Piece      0 - Valid index: 2
20 PASS: Piece      1 - Valid index: 2
21 PASS: Piece      2 - Valid index: 4
22 PASS: Piece      3 - Valid index: 4
23 PASS: Piece      4 - Valid index: 4
24 PASS: Piece      5 - Valid index: 3
25 PASS: Piece      6 - Valid index: 6
26 PASS: Piece      7 - Valid index: 2
27 PASS: Piece      8 - Valid index: 2
28 PASS: Piece      9 - Valid index: 3
29
30 === Test 3: Verify Tetromino Data ===
31 PASS: Generated piece has valid shape data
32
33 Simulation Finished
34 $finish called at time : 565 ns : File "/home/alkhalmr/gtprj/ag-tetris/
35     test/tb_generate_tetromino.sv" Line 90
36 INFO: [USF-XSim-96] XSim completed. Design snapshot ,
37     tb_generate_tetromino_behav' loaded.
38 INFO: [USF-XSim-97] XSim simulation ran for 1000ns
39 launch_simulation: Time (s): cpu = 00:00:12 ; elapsed = 00:00:09 .
40     Memory (MB): peak = 8409.602 ; gain = 0.000 ; free physical =
41     1013476 ; free virtual = 1028715

```

### tb\_vga\_out.sv

*VGA timing testbench. Runs simulation over multiple frames to verify hsync, vsync, and active\_area signals.*

```

1 `timescale 1ns / 1ps
2
3 /* tb_vga_out
4  * Drives vga_out through reset and a short run to sanity-check sync/
5   * active signals.

```

```

5  /*
6  module tb_vga_out;
7
8      logic clk;
9      logic rst;
10     logic [10:0] curr_x;
11     logic [9:0] curr_y;
12     logic hsync;
13     logic vsync;
14     logic active_area;
15
16     vga_out uut (
17         .clk(clk),
18         .rst(rst),
19         .curr_x(curr_x),
20         .curr_y(curr_y),
21         .hsync(hsync),
22         .vsync(vsync),
23         .active_area(active_area)
24     );
25
26     initial begin
27         clk = 0;
28         forever #6 clk = ~clk;
29     end
30
31     initial begin
32         rst = 1;
33         #100;
34         rst = 0;
35
36         // Run for a few frames
37         #20000000; // 20ms (approx 1 frame at 60Hz is 16ms)
38
39         $finish;
40     end
41
42 endmodule

```

### Simulation Log:

```

1 source tb_vga_out.tcl
2 # set curr_wave [current_wave_config]
3 # if { [string length $curr_wave] == 0 } {
4 #     if { [llength [get_objects]] > 0} {
5 #         add_wave /
6 #         set_property needs_save false [current_wave_config]
7 #     } else {
8 #         send_msg_id Add_Wave-1 WARNING "No top level signals found.
9 #             Simulator will start without a wave window. If you want to open a
10            wave window go to 'File->New Waveform Configuration' or type '
11            create_wave_config' in the TCL console."
12 #     }
13 # }
14 # run 1000ns
12 INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_vga_out_behav'
13 loaded.
13 INFO: [USF-XSim-97] XSim simulation ran for 1000ns
14 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
15 run all
16 $finish called at time : 20000100 ns : File "/home/alkhalmr/gtprj/ag-
    tetris/test/tb_vga_out.sv" Line 37
17 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.*

```

1  `timescale 1ns / 1ps
2  `include "GLOBAL.sv"
3
4  /* tb_ps2_keyboard
5   * Feeds ps2_keyboard with make/break sequences and checks decoded
6   * events.
7  */
8  module tb_ps2_keyboard;
9
10    logic clk;
11    logic rst;
12    logic ps2_clk;
13    logic ps2_data;
14    logic [7:0] current_scan_code;
15    logic current_make_break;
16    logic key_event_valid;
17
18    ps2_keyboard uut (
19      .clk(clk),
20      .rst(rst),
21      .ps2_clk(ps2_clk),
22      .ps2_data(ps2_data),
23      .current_scan_code(current_scan_code),
24      .current_make_break(current_make_break),
25      .key_event_valid(key_event_valid)
26    );
27
28    always #5 clk = ~clk;
29
30    int pass_count = 0;
31    int fail_count = 0;
32
33    task send_ps2_byte(input [7:0] data);
34      integer i;
35      logic parity;
36      begin
37        parity = ^data;
38
39        ps2_data = 0;
40        #20000 ps2_clk = 0;
41        #20000 ps2_clk = 1;
42        for (i = 0; i < 8; i = i + 1) begin

```

```

43         ps2_data = data[i];
44         #20000 ps2_clk = 0;
45         #20000 ps2_clk = 1;
46     end
47
48     ps2_data = parity;
49     #20000 ps2_clk = 0;
50     #20000 ps2_clk = 1;
51
52     ps2_data = 1;
53     #20000 ps2_clk = 0;
54     #20000 ps2_clk = 1;
55
56     #50000; // Wait between bytes
57 end
58 endtask
59
60 task wait_for_event;
61 begin
62     repeat(10000) begin
63         @(posedge clk);
64         if (key_event_valid) begin
65             repeat(10) @(posedge clk);
66             return;
67         end
68     end
69     $display("WARNING: Timeout waiting for key_event_valid");
70 end
71 endtask
72
73 initial begin
74     clk = 0;
75     rst = 1;
76     ps2_clk = 1;
77     ps2_data = 1;
78
79     #20 rst = 0;
80
81     $display("== ps2_keyboard Testbench ==");
82     $display("Testing PS2 keyboard interface with key_event_valid
pulse\n");
83
84     #100;
85
86     // Test 1: Make Code (Press 'A' - 0x1C)
87     $display("Test 1: Press 'A' (0x1C)");
88     send_ps2_byte(8'h1C);
89     wait_for_event();
90
91     if (current_scan_code == 8'h1C && current_make_break == 1)
begin
92         $display("  PASS: Key Press Detected (Code: %h, State: Make
)", current_scan_code);
93         pass_count++;
94     end else begin
95         $display("  FAIL: Expected 0x1C Make, got %h %b",
current_scan_code, current_make_break);
96         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)
108 begin
109     $display("  PASS: Key Release Detected (Code: %h, State:
Break)", current_scan_code);
110     pass_count++;
111 end else begin
112     $display("  FAIL: Expected 0x1C Break, got %h %b",
current_scan_code, current_make_break);
113     fail_count++;
114 end
115
116 // Test 3: Extended Key Press (Left Arrow - E0 6B)
117 $display("\nTest 3: Press Left Arrow (E0 6B)");
118 send_ps2_byte(8'hE0);
119 #5000; // E0 prefix should NOT generate event alone
120
121 send_ps2_byte(8'h6B);
122 wait_for_event();
123
124 if (current_scan_code == 8'h6B && current_make_break == 1)
125 begin
126     $display("  PASS: Extended Key Press (Left Arrow: %h, Make)
", current_scan_code);
127     pass_count++;
128 end else begin
129     $display("  FAIL: Expected 0x6B Make, got %h %b",
current_scan_code, current_make_break);
130     fail_count++;
131 end
132
133 // Test 4: Extended Key Release (Left Arrow - E0 F0 6B)
134 $display("\nTest 4: Release Left Arrow (E0 F0 6B)");
135 send_ps2_byte(8'hE0);
136 #5000;
137 send_ps2_byte(8'hF0);
138 #5000;
139 send_ps2_byte(8'h6B);
140 wait_for_event();
141
142 if (current_scan_code == 8'h6B && current_make_break == 0)
143 begin
144     $display("  PASS: Extended Key Release (Left Arrow: %h,
Break)", current_scan_code);
145     pass_count++;
146 end else begin
147     $display("  FAIL: Expected 0x6B Break, got %h %b",
current_scan_code, current_make_break);
148     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
    $display(" PASS: Left Shift Press (Code: %h, State: Make)",
154 , current_scan_code);
    pass_count++;
end else begin
    $display(" FAIL: Expected 0x12 Make, got %h %b",
157 current_scan_code, current_make_break);
    fail_count++;
end
159
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
    $display(" PASS: Space Press (Code: %h, State: Make)",
167 current_scan_code);
    pass_count++;
end else begin
    $display(" FAIL: Expected 0x29 Make, got %h %b",
170 current_scan_code, current_make_break);
    fail_count++;
end
172
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);
#5000;
178 send_ps2_byte(8'h75);
179 wait_for_event();
180
181 if (current_scan_code == 8'h75 && current_make_break == 1)
begin
    $display(" PASS: Up Arrow Press (Code: %h, State: Make)",
182 current_scan_code);
    pass_count++;
end else begin
    $display(" FAIL: Expected 0x75 Make, got %h %b",
185 current_scan_code, current_make_break);
    fail_count++;
end
187
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

```

```

195
196 endmodule

```

**Simulation Log:**

```

1 == ps2_keyboard Testbench ===
2 Testing PS2 keyboard interface with key_event_valid pulse
3
4 Test 1: Press 'A' (0x1C)
5 INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_ps2_keyboard_behav' loaded.
6 INFO: [USF-XSim-97] XSim simulation ran for 1000ns
7 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
8 run all
9 WARNING: Timeout waiting for key_event_valid
10    PASS: Key Press Detected (Code: 1c, State: Make)
11
12 Test 2: Release 'A' (F0 1C)
13 WARNING: Timeout waiting for key_event_valid
14    PASS: Key Release Detected (Code: 1c, State: Break)
15
16 Test 3: Press Left Arrow (E0 6B)
17 WARNING: Timeout waiting for key_event_valid
18    PASS: Extended Key Press (Left Arrow: 6b, Make)
19
20 Test 4: Release Left Arrow (E0 F0 6B)
21 WARNING: Timeout waiting for key_event_valid
22    PASS: Extended Key Release (Left Arrow: 6b, Break)
23
24 Test 5: Press Left Shift (0x12) - Hold Key
25 WARNING: Timeout waiting for key_event_valid
26    PASS: Left Shift Press (Code: 12, State: Make)
27
28 Test 6: Press Space (0x29) - Hard Drop
29 WARNING: Timeout waiting for key_event_valid
30    PASS: Space Press (Code: 29, State: Make)
31
32 Test 7: Press Up Arrow (E0 75) - Rotate
33 WARNING: Timeout waiting for key_event_valid
34    PASS: Up Arrow Press (Code: 75, State: Make)
35
36 === Test Summary ===
37 Passed: 7
38 Failed: 0
39 Simulation Finished
40 $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.*

```

1 `timescale 1ns / 1ps
2
3 /* tb_PS2Receiver
4  * Drives PS2Receiver with representative scan codes and checks decoded
   history.

```

```

5  /*
6  module tb_PS2Receiver;
7
8      logic clk;
9      logic kclk;
10     logic kdata;
11     logic [31:0] keycodeout;
12
13    int pass_count = 0;
14    int fail_count = 0;
15
16    PS2Receiver uut (
17        .clk(clk),
18        .kclk(kclk),
19        .kdata(kdata),
20        .keycodeout(keycodeout)
21    );
22
23    always #5 clk = ~clk;
24
25    task send_ps2_byte(input [7:0] data);
26        integer i;
27        logic parity;
28        begin
29            parity = ^data;
30
31            kdata = 0;
32            #20000 kclk = 0;
33            #20000 kclk = 1;
34
35            for (i = 0; i < 8; i = i + 1) begin
36                kdata = data[i];
37                #20000 kclk = 0;
38                #20000 kclk = 1;
39            end
40
41            kdata = parity;
42            #20000 kclk = 0;
43            #20000 kclk = 1;
44
45            kdata = 1;
46            #20000 kclk = 0;
47            #20000 kclk = 1;
48
49            #50000; // Wait between bytes
50        end
51    endtask
52
53    initial begin
54        clk = 0;
55        kclk = 1;
56        kdata = 1;
57
58        $display("== PS2Receiver Testbench ==");
59        $display("Testing low-level PS2 protocol reception\n");
60
61        #100;
62
```

```

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
71             [7:0]);
72         fail_count++;
73     end
74
75     $display("\nTest 2: Multiple Scancodes");
76     send_ps2_byte(8'h23); // 'D' key
77     #1000;
78     if (keycodeout[7:0] == 8'h23 && keycodeout[15:8] == 8'h1C)
79     begin
80         $display(" PASS: History buffer: prev=0x1C, curr=0x23");
81         pass_count++;
82     end else begin
83         $display(" FAIL: Expected prev=0x1C curr=0x23, got %h",
84             keycodeout[15:0]);
85         fail_count++;
86     end
87
88     send_ps2_byte(8'h2B); // 'F' key
89     #1000;
90     if (keycodeout[7:0] == 8'h2B) begin
91         $display(" PASS: Received scancode 0x2B");
92         pass_count++;
93     end else begin
94         $display(" FAIL: Expected 0x2B, got 0x%h", keycodeout
95             [7:0]);
96         fail_count++;
97     end
98
99     $display("\nTest 3: Break Code (F0 1C - Release A)");
100    send_ps2_byte(8'hF0);
101    #1000;
102    send_ps2_byte(8'h1C);
103    #1000;
104    if (keycodeout[15:8] == 8'hF0 && keycodeout[7:0] == 8'h1C)
105    begin
106        $display(" PASS: Break code received (F0 1C)");
107        pass_count++;
108    end else begin
109        $display(" FAIL: Break code error. Got: %h", keycodeout
110             [15:0]);
111        fail_count++;
112    end
113
114    $display("\nTest 4: Extended Key (E0 6B - Left Arrow)");
115    send_ps2_byte(8'hE0);
116    #1000;
117    send_ps2_byte(8'h6B);
118    #1000;
119    if (keycodeout[15:8] == 8'hE0 && keycodeout[7:0] == 8'h6B)
120    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
118 [15:0]);
119         fail_count++;
120     end
121
122     $display("\nTest 5: Extended Key Release (E0 F0 6B - Release
Left Arrow)");
123     send_ps2_byte(8'hE0);
124     #1000;
125     send_ps2_byte(8'hF0);
126     #1000;
127     send_ps2_byte(8'h6B);
128     #1000;
129     if (keycodeout[23:16] == 8'hE0 && keycodeout[15:8] == 8'hF0 &&
keycodeout[7:0] == 8'h6B) begin
130         $display("  PASS: Extended release received (E0 F0 6B)");
131         pass_count++;
132     end else begin
133         $display("  FAIL: Extended release error. Got: %h",
keycodeout[23:0]);
134         fail_count++;
135     end
136
137     $display("\nTest 6: Space Key (0x29 - Hard Drop)");
138     send_ps2_byte(8'h29);
139     #1000;
140     if (keycodeout[7:0] == 8'h29) begin
141         $display("  PASS: Space key received (0x29)");
142         pass_count++;
143     end else begin
144         $display("  FAIL: Expected 0x29, got 0x%h", keycodeout
145 [7:0]);
146         fail_count++;
147     end
148
149     $display("\nTest 7: Left Shift Key (0x12 - Hold)");
150     send_ps2_byte(8'h12);
151     #1000;
152     if (keycodeout[7:0] == 8'h12) begin
153         $display("  PASS: Left Shift key received (0x12)");
154         pass_count++;
155     end else begin
156         $display("  FAIL: Expected 0x12, got 0x%h", keycodeout
157 [7:0]);
158         fail_count++;
159     end
160
161     $display("\nTest 8: Up Arrow (E0 75 - Rotate)");
162     send_ps2_byte(8'hE0);
163     #1000;
164     send_ps2_byte(8'h75);
165     #1000;
166     if (keycodeout[15:8] == 8'hE0 && keycodeout[7:0] == 8'h75)
begin
167         $display("  PASS: Up Arrow received (E0 75)");

```

```

165         pass_count++;
166     end else begin
167         $display("  FAIL: Expected E0 75, got %h", keycodeout
168             [15:0]);
169         fail_count++;
170     end
171
172     $display("\n==== Test Summary ===");
173     $display("Passed: %0d", pass_count);
174     $display("Failed: %0d", fail_count);
175     $display("Simulation Finished");
176     $finish;
177 end
178 endmodule

```

### Simulation Log:

```

1 # run 1000ns
2 === PS2Receiver Testbench ===
3 Testing low-level PS2 protocol reception
4
5 Test 1: Single Scancode (A key - 0x1C)
6 INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_PS2Receiver_behav' loaded.
7 INFO: [USF-XSim-97] XSim simulation ran for 1000ns
8 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
9 run all
10    PASS: Received scancode 0x1C
11
12 Test 2: Multiple Scancodes
13    PASS: History buffer: prev=0x1C, curr=0x23
14    PASS: Received scancode 0x2B
15
16 Test 3: Break Code (F0 1C - Release A)
17    PASS: Break code received (F0 1C)
18
19 Test 4: Extended Key (E0 6B - Left Arrow)
20    PASS: Extended key received (E0 6B)
21
22 Test 5: Extended Key Release (E0 F0 6B - Release Left Arrow)
23    PASS: Extended release received (E0 F0 6B)
24
25 Test 6: Space Key (0x29 - Hard Drop)
26    PASS: Space key received (0x29)
27
28 Test 7: Left Shift Key (0x12 - Hold)
29    PASS: Left Shift key received (0x12)
30
31 Test 8: Up Arrow (E0 75 - Rotate)
32    PASS: Up Arrow received (E0 75)
33
34 === Test Summary ===
35 Passed: 9
36 Failed: 0
37 Simulation Finished
38 $finish called at time : 6874100 ns : File "/home/alkhalmr/gtprj/ag-

```

tetris/test/tb\_PS2Receiver.sv" Line 204