ELEC 422: Tetris ASIC

Memory Mafia: Rice University, Spring 2026

SanDISK (Sandeep) Ramlochan
Electrical and Computer Engineering
Rice University

Kathryn Files
Electrical and Computer Engineering
Rice University

Atishay RAM Lalgudi
Electrical and Computer Engineering
Rice University

***Abstract*—This custom ASIC was part of a final project designed for Rice University's ELEC 422: VLSI System Design. This ASIC seeks to simulate a hardware-native implementation of Tetris. By using a synchronous digital architecture with two-phase clocking, the system updates the game state, managing the seven tetromino shapes across a 20×10 grid through direct manipulation of hardware registers. The chip outputs a row-by-row bit-mapped signal designed to drive external LED matrices or terminal-based renderers.**

I. Introduction

Tetris is one of the most enduring and recognizable video games ever created, and for the members of this team it has been a longtime personal favorite. When the opportunity arose to design a custom ASIC for ELEC 422, implementing Tetris in hardware felt like a natural and compelling choice: a chance to bring a game we genuinely enjoy to life at the transistor level. Beyond the personal motivation, Tetris presents a rich set of VLSI design challenges: managing a 20×10 grid of state in real time, encoding all seven tetromino shapes and their rotations in a compact combinational ROM, detecting collisions and line completions entirely in hardware, and producing a row-scanned display output fast enough for a human player. These requirements collectively demand careful partitioning between a finite state machine controller and a structured datapath, making the design an ideal exercise in the FSMD methodology taught in this course.

The primary design goals were: (1) full functional correctness of all game mechanics including piece movement, rotation, gravity, line clearing, and game-over detection; (2) complete synthesizability through Design Compiler without runtime array indexing.

This system accepts user inputs for movement, rotation, and piece placement and updates the game state on a fixed clock schedule. Internally, the system maintains a 20×10 grid of occupied and empty cells, continuously evaluates piece interactions, including boundary constraints and line completion events, and streams the current board state as a row-wise bit-mapped output suitable for driving an external LED matrix.

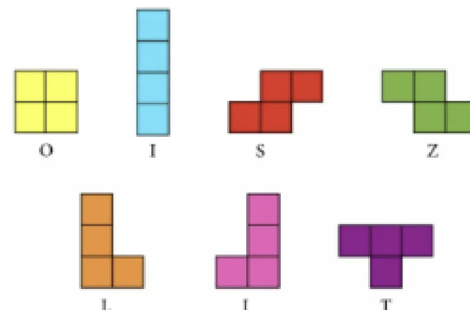

Fig. 1: Seven Tetromino Pieces

II. Architecture and Design Partitioning

The Tetris ASIC is architected using the **Finite State Machine with Datapath (FSMD)** methodology. The system operates as a closed-loop control system where the controller and datapath are in constant communication: the output of one determines the next action of the other. This separation ensures that high-level game logic is decoupled from the hardware-intensive grid manipulations. The system is divided into two primary hierarchical modules: the controller (`tetris_fsm`) and the datapath (`tetris_datapath`).

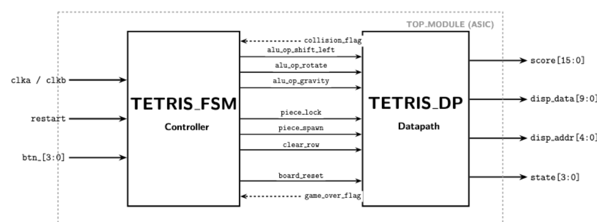

Fig. 2: FSM Datapath Architecture

A. *Two-Phase Clocking*

Two non-overlapping clock signals, `clka` and `clkb`, govern all sequential logic in the design.

It is important to note that the Tetris ASIC receives `clka` and `clkb` as external inputs via dedicated input pads (`in_clka` and `in_clkb` in the top module).

Within the FSM, the `FSM_SEQ` always block triggers on `negedge clka` and latches `next_state` and `move_cmd`. The `OUTPUT_LOGIC` block triggers on `negedge clkb` and uses the now-stable `next_state` to drive all control outputs to the datapath. In the datapath, gravity counting and the free-running LFSR seed counter update on `clka`, while board registers, piece position, score, and LFSR shift on `clkb`. This phase separation guarantees that by the time `clkb` fires, all FSM control signals from the `clka` phase are fully resolved and stable.

B. The Control Domain (`tetris_fsm`)

The controller is implemented as a Moore-style state machine that evaluates user inputs from the external pads (`btn_left`, `btn_right`, `btn_rotate`, `btn_drop`) alongside internal status flags. Rather than performing coordinate geometry, the FSM issues discrete command pulses and ALU operation codes to the datapath:

- **Movement Commands:** `alu_op_shift_left`, `alu_op_shift_right`, `alu_op_rotate`, and `alu_op_gravity`.
- **State Management:** `piece_lock`, `piece_spawn`, and `board_reset`.
- **Line Resolution:** Triggering the `clear_row` sequence and asserting `score_inc`.

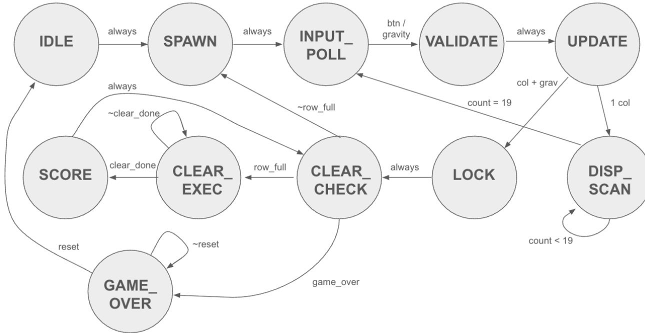

Fig. 3: State transition diagram for the `tetris_fsm` controller

C. The Structural Datapath (`tetris_datapath`)

The datapath maintains the 20×10-bit register array representing the playing field. To meet DC synthesis requirements, all internal loops are fully unrolled and no runtime array indexing is utilized. Key functional units include:

- **Active Piece Logic:** Registers storing the current tetromino type and its Cartesian coordinates.
- **Collision Detection:** Combinatorial logic that evaluates the next state against the existing board to assert `collision_flag`.
- **Display Scanning:** A multiplexer tree that outputs board state via `disp_row_data` for external matrix driving.

D. System Integration

At the top-level (`top_module`), the FSM drives the datapath's operations while the datapath returns critical status signals: `collision_flag`, `row_full_flag`, and `game_over_flag`: which dictate the FSM's next state transitions. This modularity allows for high-frequency operation and provides a clear diagnostic path for post-route simulation.

III. Special Circuit Blocks

A. Pseudo-Random Number Generation (LFSR)

A 7-bit Linear Feedback Shift Register (LFSR) generates pseudo-random tetromino types during gameplay. The LFSR implements the polynomial $x^7 + x^6 + 1$, advancing each cycle on the falling edge of `clkb`:

```
lfsr <= {lfsr[5:0], lfsr[6] ^ lfsr[5]};
```

This tap configuration produces a maximal-length sequence of $2^7 - 1 = 127$ states before repeating. To avoid the all-zeros state, the LFSR is seeded at reset using a free-running 7-bit counter (`free_cnt`) that increments on every falling edge of `clka`. If the counter is zero at reset, a hardcoded seed (`7'b1001101`) is substituted. The lower three bits select the tetromino type, with `3'd7` remapped to `3'd0`:

```
piece_type <= (lfsr[2:0]==3'd7)
? 3'd0 : lfsr[2:0];
```

B. Piece Mask ROM (Combinational)

All seven tetromino shapes across all four rotation states are encoded as a combinational lookup table. The select key {`piece_type, rot_state`} indexes 28 entries, each returning a 16-bit mask representing piece occupancy in a 4×4 bounding box. A separate `rot_mask` table encodes the *next* rotation, allowing the collision detector to pre-check a rotation before committing to it.

C. Board Register Array

The 20×10 board is implemented as 20 individually declared 10-bit registers (`board0`–`board19`) rather than a Verilog array. This was required for DC synthesis compatibility: runtime array indexing with a variable index is not synthesizable in the AMI 0.5 µm standard cell flow. All board access is implemented using fully unrolled `case` statements and explicit per-register logic.

IV. Verilog Simulations (Pre-Synthesis)

Before synthesis, each module was verified independently in Questa to confirm correct functional behavior. The testbenches exercise the full range of game mechanics including spawn, movement, gravity, line clear, and game-over conditions.

A. FSM Controller

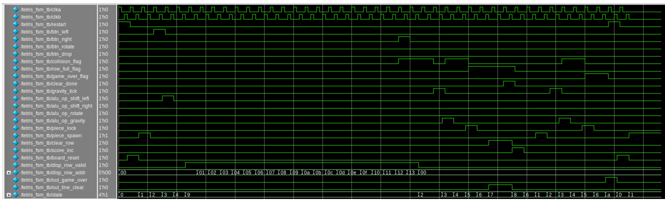

Fig. 4: Pre-synthesis Questa simulation of `tetris_fsm`.

- Restart signal goes high, asserting `board_reset` and returning the FSM to IDLE (statebits = `0000`). All control outputs are deasserted.
- FSM transitions from IDLE to SPAWN (statebits = `0001`), asserting `piece_spawn` to load the first tetromino from the LFSR.
- FSM enters INPUT_POLL (statebits = `0010`) and waits for a button event. `btn_drop` is asserted, triggering a transition to VALIDATE then UPDATE (statebits = `0100`), where `alu_op_gravity` is issued.
- After several gravity ticks the FSM cycles through DISP_SCAN (statebits = `1001`), scanning all 20 display rows with `disp_row_valid` asserted and `disp_row_addr` incrementing 0–19.
- A gravity collision is detected (`collision_flag` asserted with `move_cmd = 000`). FSM transitions to LOCK then CLEAR_CHECK (statebits = `0110`), asserting `piece_lock`. No full rows detected so FSM returns to SPAWN.

B. Datapath

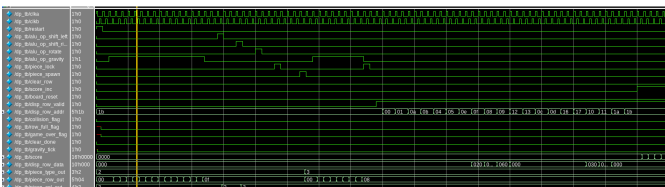

Fig. 5: Pre-synthesis Questa simulation of `tetris_datapath`.

- The gravity operation functions correctly as demonstrated by the incrementing value of `piece_row_out`.
- Left and right shift operations correctly alter `piece_col_out` (observed: $3 \rightarrow 2 \rightarrow 3$).
- After locking the piece and raising `disp_row_valid`, the row data matches the input row address from `disp_row_addr`.
- Piece spawn triggers output of a new tetromino type via `piece_type_out` using the LFSR seed.

C. Top Module

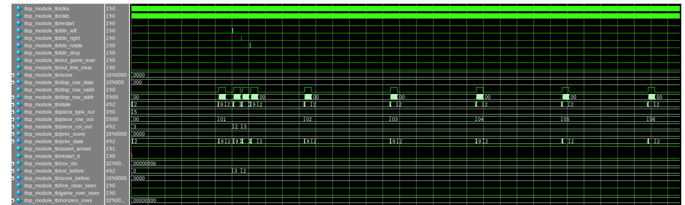

Fig. 6: Pre-synthesis Questa simulation of `top_module`.

- `piece_col_out` updates correctly ($3 \rightarrow 2 \rightarrow 3$) after `btn_left` then `btn_right`.
- Automatic gravity (every 256 clock cycles) correctly increments `piece_row_out`.
- State is primarily in state 2 (`INPUT_POLL`) except during button events or gravity ticks.
- Score does not update because no full row was cleared in this simulation segment.

V. Synthesis

The Verilog description was synthesized using Design Compiler targeting the AMI 0.5 µm (OSU05) standard cell library.

- **Target clock frequency:** *[20 MHz]*
- **Total cell count:** *[6056 cells]*
- **Estimated power consumption:** *[28.7718 mW]*

VI. Verilog Simulations (Post-Synthesis)

After synthesis by Design Compiler, each module was re-simulated in Questa using the identical testbenches from Section IV to confirm that synthesis introduced no functional changes.

A. FSM Controller

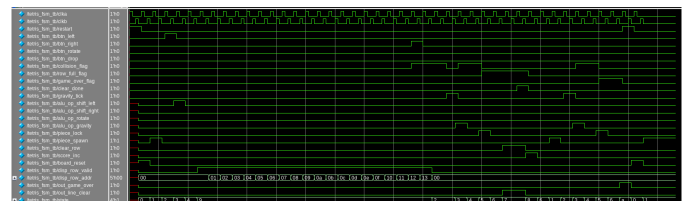

Fig. 7: Post-synthesis Questa simulation of `tetris_fsm`.

The post-synthesis FSM simulation results were identical to the pre-synthesis results, confirming that Design Compiler correctly mapped the behavioral Verilog to gate-level logic without introducing functional errors. All state transitions, control signal assertions, and display scan behavior matched the pre-synthesis waveforms exactly.

B. Datapath

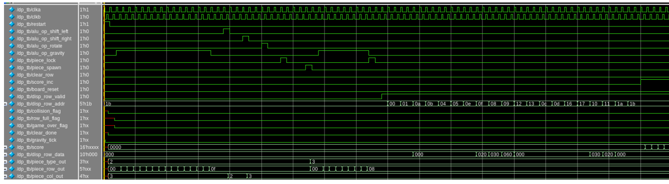

Fig. 8: Post-synthesis Questa simulation of `tetris_datapath`.

Post-synthesis datapath simulation confirmed functional equivalence with the pre-synthesis results. Gravity increments, shift operations, piece locking, and display multiplexing all produced identical waveforms, verifying that the fully unrolled loop structure synthesized correctly.

C. Top Module

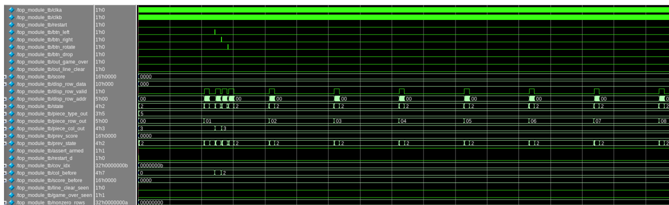

Fig. 9: Post-synthesis Questa simulation of `top_module`.

The integrated top-module post-synthesis simulation matched the pre-synthesis behavior across all observed signals, confirming that the closed-loop FSM-datapath interaction was preserved through synthesis.

VII. Chip Layout

Following post-synthesis verification, the design was placed and routed using Innovus and subsequently imported into Magic for DRC verification and padframe integration. The finalized core measures 7319×7140 lambda units. A 104-pin padframe was generated using Gavin Jing's padframe generator to accommodate these dimensions, as the standard 64-pin padframe was insufficient for the 61 required I/O pins.

A. Core Layout — Magic

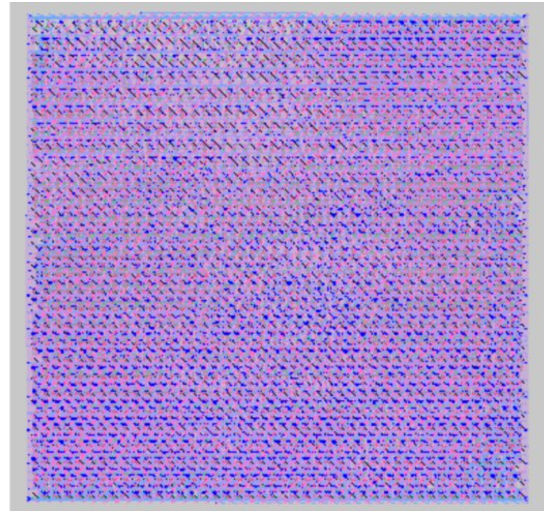

Fig. 10: Magic place-and-route view of the core.

The Innovus result shows a highly dense and uniform cell arrangement. This is a direct consequence of the 20 individually declared 10-bit board registers, the fully unrolled collision detection, and the 28-entry piece mask ROM, which collectively produce a very large standard cell count. The predominantly diagonal routing is characteristic of the AMI 0.5 µm metal layer usage.

B. Core Layout — Innovus

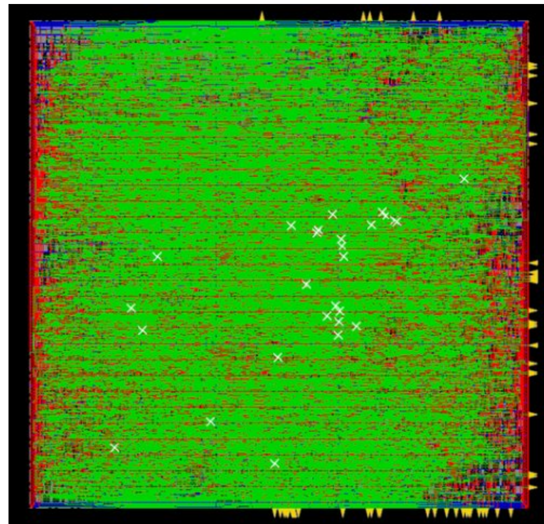

Fig. 11: Core layout in Innovus

After importing into Magic, DRC checking revealed violations visible as white $\times$ markers, concentrated in the central routing region. These are typical of high-density place-and-route at AMI 0.5 µm design rules.

C. Padframe Layout — Magic

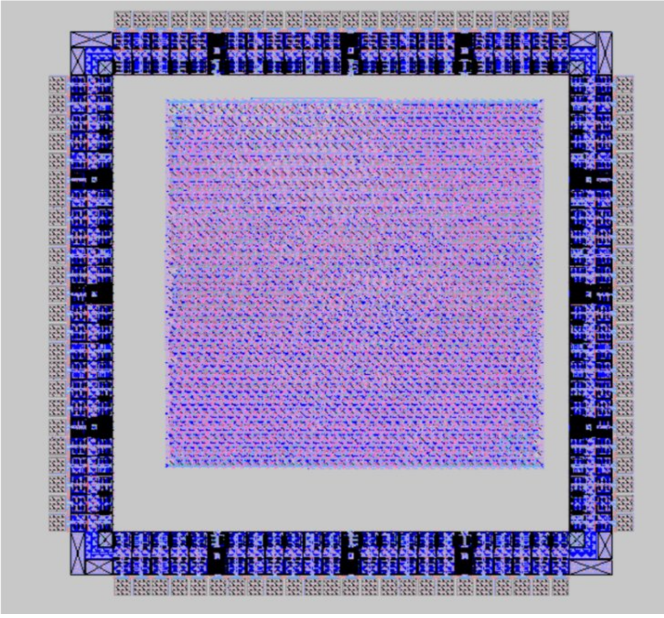

Fig. 12: Integrated core within the 104-pin padframe as viewed in Magic. Corner cells are visible at each corner of the padframe ring.

D. Padframe Layout — Magic

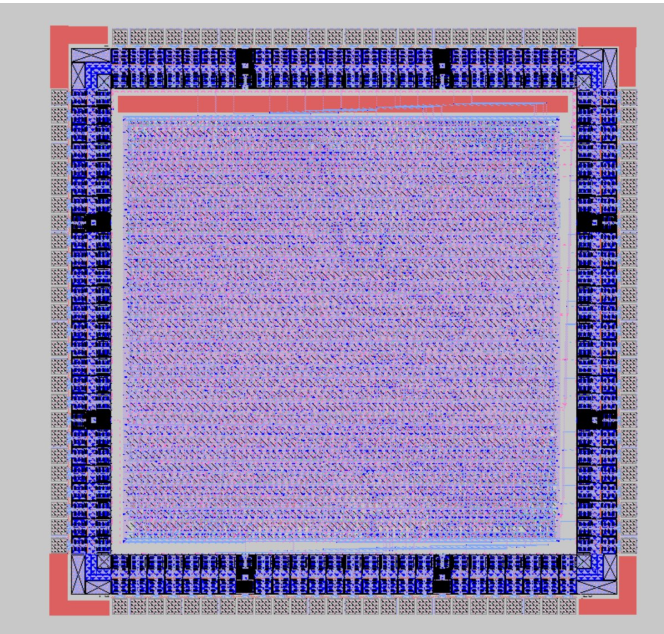

Fig. 13: Fully integrated core with 104-pin padframe in Magic. Salmon corner cells and the pink VDD supply ring are clearly visible.

The completed chip shows the core integrated within the 104-pin padframe. The salmon blocks at each corner are standard OSU corner cells. The thick pink band at the top of the core is the VDD supply rail; the GND rail runs along the bottom. Bidirectional pads from the OSU library are used for all signal connections: when `OEN` is high, the pad drives `DO` onto the physical pin; when `OEN` is low, the pad captures the external signal into `DI`. All unused pads are configured as inputs with `OEN` held low.

VIII. Core Testbench (IRSIM)

After chip layout, Magic generated a behavioral netlist for post-layout simulation. An IRSIM testbench was written to replicate the Questa test sequence, verifying that the placed-and-routed design maintains functional equivalence with the pre-layout simulation.

A. Core IRSIM

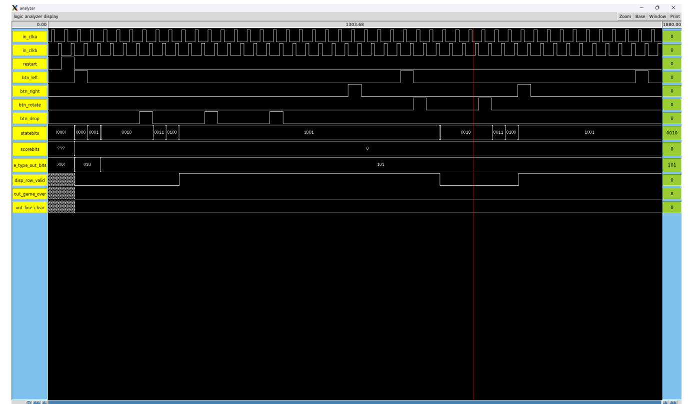

Fig. 14: IRSIM simulation of the core netlist after place and route.

Reading from the core IRSIM waveform (Fig. 14):

- `restart` goes high then low. `statebits` transitions from unknown (`XXX`) through `0000` (IDLE) to `0001` (SPAWN). `e_type_out_bits` resolves from `XXX` to `010`, indicating the first tetromino type is selected by the LFSR.
- `btn_left` is asserted. `statebits` transitions through `0010` (INPUT_POLL) → `0011` (VALIDATE) → `0100` (UPDATE) confirming the FSM correctly captures the button input and issues the shift command to the datapath.
- `btn_drop` is asserted multiple times. Each assertion causes the FSM to cycle through VALIDATE and UPDATE, advancing the piece downward. `disp_row_valid` pulses high during each DISP_SCAN phase (statebits = `1001`).
- After multiple drop operations, the piece type transitions to `101` as seen in `e_type_out_bits`, confirming a new piece spawned after the previous one locked. `scorebits` remains `0` throughout as no full row was cleared in this test sequence.
- `btn_right` and `btn_rotate` are exercised. The FSM correctly responds, cycling through VALIDATE and UPDATE states. `out_game_over` and `out_line_clear` remain low, confirming no erroneous end conditions are triggered.

B. Padframe IRSIM

Fig. 15: IRSIM simulation of the complete design including padframe I/O pads. Signal names are prefixed with `p_` reflecting the padframe output node naming.

The padframe IRSIM simulation in Fig. 15 replicates the same test sequence through the full padframe integration, exercising signals as they appear at the physical pad nodes. Key observations:

- The `p_` prefix on all signals confirms the simulation is operating at the padframe boundary, not internal core nodes.
- `p_restart`, `p_btn_left`, `p_btn_right`, `p_btn_rotate`, and `p_btn_drop` all toggle correctly and propagate through the bidirectional pads to the core.
- `statebits` and `e_type_out_bits` match the core IRSIM results, confirming the padframe introduces no functional errors in signal routing.
- `_disp_row_valid` (active-low at pad) pulses correctly during DISP_SCAN phases.
- `p_out_game_over` and `_out_line_clear` remain deasserted throughout, consistent with the test sequence not triggering end conditions.
- `scorebits` remains `0` as no row clears were performed in this simulation window.

The padframe simulation results are functionally consistent with the core IRSIM results, confirming successful integration of the core into the 104-pin padframe with no signal integrity issues introduced by the pad cells.

C. Unresolved Issues

The initial `statebits` value of `XXX` visible at the start of both IRSIM simulations is expected: the flip-flops have not yet been initialized before the first falling edge of `restart`. Once `restart` is asserted and released, all state registers resolve to known values and the simulation proceeds correctly. This is not a functional error but rather the expected behavior of uninitialized flip-flops in a post-layout netlist before the first reset event.

IX. Padframe Integration

The core has a size of 7319×7140 lambda units and requires 61 I/O pins. To accommodate these dimensions, Gavin Jing's padframe generator was used to construct a 104-pin padframe. The OSU bidirectional pad cell is used for all signal I/O. Table I describes the bidirectional pad behavior.

TABLE I: Bidirectional Pad Behavior (OSU Library)

OEN	PAD	DI	DO
0	1	1	X
0	0	0	X
1	1	X	1
1	0	X	0

When `OEN` is high, `DO` is written to the physical pad (output mode). When `OEN` is low, the pad value is read into `DI` (input mode). VDD and GND pads supply the core power rails and set all `OEN` flags accordingly.

X. Pinmap

Fig. 16: Labeled 104-pin padframe with signal names.

XI. Testing Strategy

To test a fabricated version of this ASIC, the chip must be connected to external hardware that exercises all inputs and observes all outputs. The following describes the required hardware setup and a verification procedure.

A. Required Parts

TABLE II: Required Parts for Chip Testing

Part	Quantity
Tetris ASIC	1
20×10 LED Matrix	1
Pushbuttons	5
Current-limiting resistors	200
3.3 V Power Supply	1
Dual-phase clock source (oscillator or FPGA)	1
LEDs or 7-segment display (score output)	16

B. Wiring and Verification Procedure

The dual-phase clock signals `clka` and `clkb` must be generated externally and fed into the `in_clka` and `in_clkb` input pads. These must be non-overlapping as described in Section II-A, and can be generated using an FPGA, a divide-by-2 circuit, or an oscillator. The absolute frequency should be chosen so that the gravity tick (every 256 clock cycles) produces a visible fall rate on the LED display.

The five pushbuttons connect directly to the `restart`, `btn_left`, `btn_right`, `btn_rotate`, and `btn_drop` input pads with pull-down resistors to prevent floating inputs. The 20×10 LED matrix is driven using the `disp_row_data[9:0]` column signals and `disp_row_addr[4:0]` row address, with `disp_row_valid` gating the output.
The display scans row by row through all 20 rows each DISP_SCAN phase. Current-limiting resistors (approximately $100\,\Omega$ per LED for 3.3 V supply) should be placed in series with each column line.

The score output (`score[15:0]`) can be monitored using 16 LEDs or a pair of 7-segment displays to observe line clears incrementing the count. The `out_game_over` and `out_line_clear` outputs can each drive a single indicator LED.

The following test sequence verifies correct chip operation after fabrication:

1) Assert `restart` — observe all LEDs clear and `state` return to IDLE then SPAWN.
2) Release `restart` — first tetromino should appear at the top of the LED matrix.
3) Press `btn_left`/`btn_right` — observe piece shifting horizontally on display.
4) Press `btn_rotate` — observe piece rotation on display.
5) Allow gravity to run — observe piece falling one row every 256 clock cycles.
6) Press `btn_drop` — observe immediate gravity step.
7) Stack pieces to fill a complete row — observe `out_line_clear` pulsing and `score` incrementing by 1.
8) Stack pieces to the top row — observe `out_game_over` asserting and FSM locking in GAME_OVER state.
9) Assert `restart` to reset and repeat.