



The EE354 project involves building some sort of cool digital design using the provided A7 FPGAs. This design can use any of the available I/O, including but not limited to: Switches, LEDs, SSDs, buttons, VGA, USB (keyboard, mouse, etc.), Ethernet, pin headers, temperature sensor, accelerometer, microphone, and mono audio. Although there are many input output modes available, please note that most projects only use the switches, LEDs, SSDs, buttons, and/or VGA to interface with the outside world.

The design itself can basically be anything that you can think to use an FPGA for. Almost all projects are VGA output games (snake, tetris, pacman, 2048, space invaders, flappy bird, etc.), since that is what the skeleton code provides. It is not required to be this though, you could also implement some algorithms such as matrix multiplication, FFT, or HFT algorithms and then write a testbench to check its correctness and then measure its performance on the board (# clock cycles, resources utilized, energy, etc).

How would you make 2048 game

Build 2048 as a small hardware system on the Artix-7 board: a game-state core that updates the 4×4 grid on each button press, a VGA renderer that draws the board at 640×480 , a debounced input interface for Up/Down/Left/Right/Reset, and a simple PRNG to spawn new tiles with 90% 2s and 10% 4s. The core of the design is a “compress–merge–compress” datapath per row/column, wrapped in a finite-state machine that sequences inputs, updates, scoring, spawning, and game-over detection.

What you'll build

- A top-level that ties together: clocking, reset, input debouncers, game FSM, move/merge engine, RNG, score tracker, and VGA sync + pixel renderer.
- Controls via on-board buttons (Up, Down, Left, Right, Reset); output to VGA for the grid and score; optional LEDs/SSDs for status and score mirroring.

Core architecture

- Data model: store the board as 16 cells of 4–6 bits each holding the exponent e of the tile value 2^e ($0 = \text{empty}$, $1 = 2$, $2 = 4$, ...). This avoids large integers and eases comparisons and merges.
- Game FSM states: IDLE → READ_INPUT → APPLY_MOVE → IF_MOVED_SPAWN → CHECK_END → RENDER_READY, with a one-move-per-button-press policy to prevent repeats.
- Score register: add $2^{e_{\text{new}}}$ on each successful merge that produces exponent e_{new} .

Move/merge engine

- Algorithm per line (row or column) for a direction: compress empties out, merge adjacent equals once, compress again.
- Implement as a small hardware pipeline:
 - Stage 1 (Compress1): pack non-zero exponents to the “front” (e.g., left for a left move) and fill trailing with zeros.
 - Stage 2 (Merge): scan from the front; if $\text{cell}[i] == \text{cell}[i+1]$ and non-zero, output $\text{cell}[i]+1$, set $\text{cell}[i+1]=0$, and accumulate score $+ = 2^{\text{newExp}}$; skip the next to ensure a cell merges at most once.
 - Stage 3 (Compress2): pack again to remove zeros created by merges.
- Directions:
 - Left/Right process rows; for Right, reverse the row before Stage 1 and reverse back at the end.
 - Up/Down process columns; extract the column into a 4-wide vector, apply the same pipeline, then write back; for Down, reverse similarly.
- Change-detection: OR of (pre-line != post-line) across all 4 lines determines whether the board moved to conditionally spawn a tile.

Random tile spawn

- Use an n -bit LFSR as a PRNG; advance it once per spawn.
- Tile value: assign 2 if the top 4-bit slice < 14 ($\approx 87.5\%$) else 4; or implement exact 90% by a 10-bit threshold.
- Spawn location: collect indices of empty cells, use LFSR modulo the count to choose one uniformly; write exponent 1 for a 2 or 2 for a 4.

Score and end conditions

- Score increments by the value formed on each merge: add $2^{e_{\text{new}}}$ for each merged pair producing exponent e_{new} .
- Game over if: no empty cells and for every row and column, no adjacent equal exponents remain; game won flag when any cell exponent reaches 11 for 2048 (optionally continue

play).

VGA rendering

- Timing: $640 \times 480 @ 60$ Hz with a 25 MHz pixel enable (many monitors accept 25.0 MHz vs. 25.175 MHz). Generate hsync/vsync, x,y pixel, and a video_on gate.
- Layout: center a 4×4 grid; compute $\text{cell_w} = \min(480, 640) / 4$ minus margins; map pixel x,y to cell indices (cx, cy) and in-cell coordinates.
- Graphics: draw:
 - Board background and grid lines.
 - Tile rectangles with color LUT indexed by exponent e .
 - Numbers via an 8×8 or 8×16 bitmap font in a small ROM; render by testing font bit for each character pixel.
- Score/Status: render score and "Game Over"/"You Win" text with the same font; optionally mirror score on 7-seg.

Controls and debouncing

- For each button: synchronize to the system clock (2FF synchronizer), then debounce with a counter-based filter, then detect rising edge to generate a one-clock pulse per press.
- Lockout: after a valid move pulse, ignore further moves until the FSM finishes APPLY_MOVE → SPAWN → CHECK_END and returns to IDLE.

Clocks and resets

- Use the board's 100 MHz input clock for logic and derive a 25 MHz pixel enable with a counter (clk_div by 4) or MMCM if desired; keep a clean clock-domain crossing if you generate a separate pixel clock.
- Reset should synchronously clear FSM, board state, score, PRNG seed, and renderer state; optionally allow a soft reset via a switch.

Data paths and modules

- board_ram: simple 16×6 register file for exponents; 2 read/1 write ports if you stream lines, or purely registers for small size.
- line_unit: takes 4 exponents, outputs 4 exponents, merge score increment, and moved flag; implement with small combinational networks and a tiny controller for the scan.
- dir_mux: extracts rows/columns and reverses as needed; writes back updated lines.
- prng_lfsr: parameterizable width (e.g., 16-bit) with a primitive polynomial; seed on reset.
- spawn_unit: scans empties, indexes by PRNG, and writes a new exponent.
- endcheck_unit: empty count plus neighbor-equal checks across rows and columns.
- vga_sync: generates timing; pixel_x/pixel_y counters and video_on.

- `text_rom/font`: bitmap font ROM; decimal-to-ASCII conversion pipeline for score display.

Verification plan

- Unit testbench for `line_unit`: exhaustively test all 4-cell inputs (65,536 combos) or sample randomly to verify compress-merge-compress and one-merge-per-move.
- Property checks:
 - After a move, all non-zeros are packed against the motion side.
 - No two adjacent equals remain immediately after merge stage.
 - Score increase equals sum of $2^{e_{\text{new}}}$ merges in that move.
- Integration TB: drive sequences of moves, fix PRNG seed for reproducibility, check known positions and outcomes, and assert no change on illegal moves.
- Hardware bring-up: show grid and test pattern; then overlay tiles; finally hook inputs; add a “step” switch to single-step moves while debugging.

Timing and resources

- Target 100 MHz for logic and 25 MHz for pixel enable; the move/merge is tiny logic and easily meets timing on Artix-7.
- Storage: 16 cells \times 6 bits + a small ROM font (a few kbytes) fits in distributed RAM/BRAM trivially.
- Rendering: simple rectangles and font lookups keep utilization low; avoid per-pixel division by precomputing tile bounds.

Edge cases to handle

- Ensure a tile merges at most once per move: when merging `cell[i]` and `cell[i+1]`, skip `i+1` on this move’s scan.
- If no tiles move, do not spawn; also do not increment score.
- When three equal tiles appear in motion direction, only the pair closest to the motion side merges in that move.
- Avoid PRNG bias when selecting among k empty cells: use modulo reduction with retry if desired; for $k \leq 16$ a simple modulo is acceptable.

Minimal HDL sketch for a line

```
// inputs: a[0..3] exponents, 0==empty. direction assumed 'left'
function automatic [23:0] compress4(input [23:0] v); // 4x6-bit packed
    // pack nonzeros to the left; simple bubble of zeros rightward
    reg [5:0] x[0:3];
    integer i, pass;
    begin
        {x[1_3],x[1_2],x[1_1],x[1_0]} = v;
        for (pass=0; pass<3; pass=pass+1) begin
            for (i=0;i<3;i=i+1) if (x[i]==0) begin x[i]=x[i+1]; x[i+1]=0; end
    end
endfunction
```

```

    end
    compress4 = {x[^\_3],x[^\_2],x[^\_1],x[^\_0]};
end
endfunction

function automatic [47:0] merge4(input [23:0] v); // out_line[23:0], score[23:0]
reg [5:0] x[0:3];
reg [23:0] score;
integer i;
begin
{x[^\_3],x[^\_2],x[^\_1],x[^\_0]} = v;
score = 24'd0;
i = 0;
while (i < 3) begin
if (x[i] != 0 && x[i] == x[i+1]) begin
x[i] = x[i] + 6'd1;           // exponent++
x[i+1] = 6'd0;                // consumed
score = score + (24'd1 << x[i]); // add 2^(new exponent)
i = i + 2;                   // skip next; merged once
end else begin
i = i + 1;
end
end
end
merge4 = { {x[^\_3],x[^\_2],x[^\_1],x[^\_0]}, score };
end
endfunction

// one-line transform:
wire [23:0] c1 = compress4({a3,a2,a1,a0});
wire [47:0] m = merge4(c1);
wire [23:0] c2 = compress4(m[47:24]);
wire moved = c2 != {a3,a2,a1,a0};
wire [23:0] out_line = c2;
wire [23:0] add_score = m[23:0];

```

Pinout and I/O mapping

- Map VGA hsync/vsync and RGB (4-4-4 or 3-3-2 bits) to the board's VGA header; constrain clock to 100 MHz and specify I/O standards in the XDC.
- Buttons: Up/Down/Left/Right/Reset; switches: optional “fast spawn test” or “restart on win/lose.”
- LEDs/SSDs: display status (win/lose), highest tile, or score (scaled).

Nice extras

- Undo buffer: store last board/score to allow a single-step undo.
- Speed mode / auto-repeat: enable held-button repeat with a move cooldown timer.
- Animated slides: interpolate tile positions between frames for a smoother look (optional and cosmetic).
- Keyboard input: if your board has PS/2 or USB HID core available, map arrow keys, but buttons are sufficient.

Build milestones

- Week 1: VGA sync + background + grid, debounced button inputs, board rendering with constants.
- Week 2: Implement line_unit + direction mux; full move update; score accumulation and spawn logic.
- Week 3: End detection, on-screen score/status text, polish colors; unit/integration testbenches.
- Week 4: Optional features, performance measurements, and documentation.

If you want, a starter Vivado project outline and a small ROM font can be provided, plus an XDC template for common Artix-7 student boards; just share the exact board model so the pin constraints match.

**

**can i build a tetris 2048 hybrid on the fpga,
provide the detailed state machine design.
essentially it works like 2048 w/ a 4x4 grid except
i can choose which column i want to drop the
number in**

Yes—build a “Tetris-2048” hybrid on a 4×4 grid where the player selects a column and a single tile drops under gravity, then merging resolves exactly as in 2048 with one-merge-per-tile per drop and score equal to the sum of created tile values. Below is a detailed finite-state-machine and module design that fits on Artix-7 with standard 640×480 VGA. [\[18\]](#) [\[19\]](#) [\[20\]](#)

Rules recap and hybrid specifics

- Grid: 4×4, each cell stores exponent e so tile value is 2^e with $e = 0$ meaning empty; a falling tile is initialized as exponent 1 for “2”, with occasional exponent 2 for “4” to match 90/10 spawn ratio. [\[21\]](#) [\[20\]](#)
- Turn: player chooses a column; a new tile spawns at the top of that column, then falls until resting atop the highest filled cell or the bottom. [\[22\]](#) [\[23\]](#)
- Merge: after the tile lands, apply 2048 merge rules along the drop column only: equal adjacent exponents collide to become one tile with exponent+1; a tile can merge at most once in this drop; when three in a row are equal, the pair closest to the landing side merges. Score increases by the created tile’s value $2^{e_{\text{new}}}$. [\[20\]](#) [\[21\]](#)
- Loss: if the selected column is full before spawning, the move is invalid; treat as no-op or game over depending on mode. Win condition can be first tile exponent reaching 11 (value 2048). [\[21\]](#) [\[20\]](#)

Top modules

- Input: debounced buttons for Col0..Col3 selection and Drop/Reset pulses; optional keyboard mapping mirrors arrows/number keys. [\[18\]](#)
- PRNG: LFSR for spawn rarity; choose exponent 1 ($\approx 90\%$) or 2 ($\approx 10\%$). [\[20\]](#)
- Board: 16×6 -bit register file holding exponents; column extractor/writer. [\[20\]](#)
- Physics: gravity stepper to drop the new tile down the chosen column. [\[22\]](#)
- Merge engine: column compress–merge–compress with 2048 rules and once-per-drop merge. [\[21\]](#) [\[20\]](#)
- Score: 32-bit accumulator adds $2^{e_{\text{new}}}$ per merge. [\[20\]](#)
- VGA: $640 \times 480 @ 60$ Hz sync, pixel_x/y, video_on, tile rectangles and numbers, HUD text for score and status. [\[19\]](#) [\[24\]](#) [\[18\]](#)

VGA timing notes

- Use standard 640×480 timing with 25.175 MHz pixel clock; many boards use 25.0 MHz enable which is typically accepted by monitors; HSYNC/VSYNC per porch counts as reference. [\[24\]](#) [\[25\]](#) [\[19\]](#)
- Horizontal: 640 visible, 16 front porch, 96 sync, 48 back porch; Vertical: 480 visible, 10 front, 2 sync, 33 back. [\[19\]](#) [\[24\]](#)

Detailed FSM design

Represent the game control as a Moore FSM that sequences one complete drop-merger-render cycle per player action.

- S0 RESET: Clear board, score, RNG seed; draw idle frame; go to S1. [\[18\]](#) [\[20\]](#)
- S1 IDLE: Wait for a debounced column selection and Drop pulse; capture column index C and RNG sample for tile value; if column full, branch to S1 (beep/flash) or S9 GAME_OVER per chosen policy. [\[23\]](#) [\[20\]](#)
- S2 SPAWN_TOP: If board[C]==empty, write new exponent e_new in row 0, set falling_flag and row r=0; else invalid. [\[20\]](#)
- S3 GRAVITY_STEP: If $r < 3$ and board[r+1][C]==empty, move tile down one cell $r \leftarrow r+1$, continue S3; else go to S4 LANDED. This simulates gravity one or multiple cycles; optionally animate one row per video frame. [\[22\]](#)
- S4 LANDED: Latch landing row rL; proceed to S5 PREPARE_MERGE. [\[20\]](#)
- S5 PREPARE_MERGE: Extract the 4 exponents of column C into v[0..3] top – bottom. Create a working copy w[0..3]. [\[20\]](#)
- S6 MERGE_PHASE: Apply 2048 merge along the column toward the landing direction (downward): compress empties toward bottom, scan bottom-up for adjacent equals, merge pair into exponent+1, zero the source, and mark the merged index so it cannot merge again this drop; sum merge_score += $2^{e_{\text{new}}}$. Then compress again to settle. If any merge occurred and further merges are still possible due to chaining created by the first merge, repeat S6

once more only if the newly created tile was not already merged before; the “one-merge-per-tile-per-drop” rule implies that the tile produced in this drop does not merge again until the next player action. A simple implementation is a single merge scan with skip-next on merge.^[21]^[20]

- S7 WRITEBACK: Write w[0..3] back into column C, update total score += merge_score.^[20]
- S8 CHECK_END: If any cell exponent reaches 11, set WIN flag; If any top cell in all columns is occupied such that no future spawns are possible (all columns full), set GAME_OVER; else return to S1 IDLE.^[20]
- S9 GAME_OVER: Freeze inputs except Reset; render “Game Over” overlay.^[20]

Notes on S6 implementation details:

- Compress toward bottom: pack non-zero exponents downward and fill above with zeros.^[20]
- Merge rule ordering with triples: with three equal tiles stacked at bottom, only the bottom pair merges this drop since scanning from the resting side ensures closest pair merges; the upper tile will be considered in later drops.^[20]
- Score increments by the value of the created tile per merge, aligning with 2048 scoring.^[20]

Column merge unit (combinational)

- Inputs: v[0..3] exponents, direction=down.^[20]
- Steps:
 - CompressDown: produce c1[0..3] by moving non-zeros to bottom indices.^[20]
 - MergeDown: i from 3 downto 1: if $c1[i]==c1[i-1]!=0$ and not merged[i] and not merged[i-1], set $c1[i]=c1[i]+1$, $c1[i-1]=0$, merged[i]=1, score+= $2^{c1[i]}$, and skip i-1.^[20]
 - CompressDown again to c2; moved flag is OR of inequality with input.^[20]
- Outputs: c2[0..3], moved, add_score.^[20]

Handling invalid or blocked spawns

- If top cell of chosen column is occupied in S1, either ignore input and stay in IDLE or transition to GAME_OVER depending on desired difficulty. Many “Number Drop” variants end the game when any new spawn cannot be placed.^[23]

Input and debouncing

- Use synchronized, counter-based debounce per button; produce single-cycle pulses for column selection and Drop/Reset; lock inputs between S2 and S8 to prevent double-presses during a drop.^[18]

RNG and spawn distribution

- LFSR width 10–16; for ≈90% 2s vs 10% 4s, compare PRNG against threshold 102 (out of 1024) or use top 4 bits with $14/16=87.5\%$ as acceptable.[\[20\]](#)

VGA rendering

- Draw a centered 4×4 board; map pixel_x/y to cell indices and in-cell offsets; color tiles using a LUT keyed by exponent; render numbers via a small bitmap font ROM; overlay score and state text; sync at 640×480 timing.[\[24\]](#) [\[19\]](#) [\[18\]](#)

Verification strategy

- Unit TB for column merge unit: exhaustively test 4^4 occupancy with equalities to confirm compress–merge–compress and once-per-drop merge behavior; assert score math matches sum of created values.[\[21\]](#) [\[20\]](#)
- Integration TB: fix RNG seed, simulate sequences of column drops; check that invalid full-column selection is handled per policy and that landing/merging positions match rules.[\[20\]](#)

Edge cases to cover

- Selecting a full column (decide: reject or end).[\[23\]](#)
- Triple equal stack: only the closest pair to the landing side merges.[\[20\]](#)
- Newly created tile during a drop must not merge again in the same drop.[\[20\]](#)
- No-op drops (dropping onto already-occupied top cell when allowed) must not change score.[\[20\]](#)

Minimal column-down merge sketch

- Compress and merge toward bottom using a single pass with skip-on-merge; this enforces at-most-one merge per tile per drop and the correct priority of pairs nearest the landing side.[\[21\]](#) [\[20\]](#)

This architecture keeps the logic compact, leverages the well-known 640×480 VGA controller pattern for visuals, and cleanly adapts 2048's merge semantics to a column-drop control like number-drop games.[\[19\]](#) [\[18\]](#) [\[20\]](#)

*

write a verilog code for this

Below is a compact, synthesizable Verilog reference design for a 4×4 “Tetris–2048” hybrid on FPGA with column-select drop, single-pass 2048 merge on landing, LFSR-based 2/4 spawn, debounced inputs, and a simple VGA text/tiles renderer for 640×480@60 Hz. It follows compress–merge–compress semantics and one-merge-per-drop along the drop column, and uses a standard VGA sync generator pattern. Replace pin names and clocking with your board's XDC and clocking resources.

Top-level: ties inputs, game core, VGA

```
// Top-level skeleton for Artix-7 boards with 100 MHz clk_in
module top_2048_drop (
    input wire          clk_100,      // 100 MHz system clock
    input wire          rst_n,        // active-low reset
    input wire [3:0]    btn_col,      // column select buttons [^3_3]=col3 ... [^3_0]=col0
    input wire          btn_drop,     // drop button
    output wire         vga_hs,       // VGA hsync
    output wire         vga_vs,       // VGA vsync
    output wire [3:0]   vga_r,        // VGA R
    output wire [3:0]   vga_g,        // VGA G
    output wire [3:0]   vga_b,        // VGA B
);
// Clock enable for 25 MHz pixel enable from 100 MHz
reg [1:0] div;
always @ (posedge clk_100 or negedge rst_n) begin
    if (!rst_n) div <= 2'd0;
    else        div <= div + 2'd1;
end
wire pix_en = (div == 2'd0); // 25 MHz enable tick for VGA pixel advance [web:33][web:2

// Debounce/sync inputs
wire [3:0] col_pulse;
wire drop_pulse;
genvar i;
generate
    for (i=0;i<4;i=i+1) begin: DB_COL
        btn_debounce udb_col(.clk(clk_100), .rst_n(rst_n), .btn(btn_col[i]), .pulse(col_pu)
    end
endgenerate
btn_debounce udb_drop(.clk(clk_100), .rst_n(rst_n), .btn(btn_drop), .pulse(drop_pulse))

// Column selection: capture one-hot into index on pulse
reg [1:0] sel_col_idx;
always @ (posedge clk_100 or negedge rst_n) begin
    if (!rst_n) sel_col_idx <= 2'd0;
    else if (|col_pulse) begin
        casex (col_pulse)
            4'b0001: sel_col_idx <= 2'd0;
            4'b0010: sel_col_idx <= 2'd1;
            4'b0100: sel_col_idx <= 2'd2;
            4'b1000: sel_col_idx <= 2'd3;
            default: sel_col_idx <= sel_col_idx;
        endcase
    end
end [web:27]

// Game core
wire [5:0] board_e[0:3][0:3]; // exponents (0 empty, 1=2, 2=4, ...)
wire [31:0] score;
wire game_over, game_win;

game_core_2048_drop ucore (
    .clk(clk_100),
    .rst_n(rst_n),
```

```

.col_sel(sel_col_idx),
.drop_pulse(drop_pulse),
.board_e00(board_e[~3_0][~3_0]), .board_e01(board_e[~3_0][~3_1]), .board_e02(board_e|
.board_e10(board_e[~3_1][~3_0]), .board_e11(board_e[~3_1][~3_1]), .board_e12(board_e|
.board_e20(board_e[~3_2][~3_0]), .board_e21(board_e[~3_2][~3_1]), .board_e22(board_e|
.board_e30(board_e[~3_3][~3_0]), .board_e31(board_e[~3_3][~3_1]), .board_e32(board_e|
.score(score),
.game_over(game_over),
.game_win(game_win)
); [web:12] [web:36] [web:3]

// VGA sync and renderer
wire [9:0] px, py;
wire video_on;
vga_640x480_timing uvga (.clk(clk_100), .rst_n(rst_n), .pix_en(pix_en),
                           .hs(vga_hs), .vs(vga_vs), .video_on(video_on), .px(px), .py(py)

wire [11:0] rgb;
renderer_2048_drop urend (
  .clk(clk_100),
  .video_on(video_on),
  .px(px), .py(py),
  .board_e00(board_e[~3_0][~3_0]), .board_e01(board_e[~3_0][~3_1]), .board_e02(board_e|
  .board_e10(board_e[~3_1][~3_0]), .board_e11(board_e[~3_1][~3_1]), .board_e12(board_e|
  .board_e20(board_e[~3_2][~3_0]), .board_e21(board_e[~3_2][~3_1]), .board_e22(board_e|
  .board_e30(board_e[~3_3][~3_0]), .board_e31(board_e[~3_3][~3_1]), .board_e32(board_e|
.score(score),
.game_over(game_over),
.game_win(game_win),
.rgb(rgb)
); [web:27] [web:33]

assign vga_r = rgb[11:8];
assign vga_g = rgb[7:4];
assign vga_b = rgb[3:0];
endmodule

```

Debounce and edge-pulse

```

module btn_debounce(
  input wire clk, input wire rst_n, input wire btn, output reg pulse
);
  reg [15:0] cnt; reg sync0, sync1, stable;
  always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin sync0<=0; sync1<=0; end
    else begin sync0<=btn; sync1<=sync0; end
  end
  wire btn_s = sync1;
  always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin cnt<=0; stable<=0; pulse<=0; end
    else begin
      if (btn_s==stable) begin cnt<=0; pulse<=0; end
      else begin
        cnt <= cnt + 16'd1;
        if (cnt==16'hffff) begin stable<=btn_s; pulse <= btn_s; end
      end
    end
  end
endmodule

```

```

        else pulse<=0;
    end
end
end
endmodule

```

Game core with FSM

```

module game_core_2048_drop(
    input wire          clk,
    input wire          rst_n,
    input wire [1:0]    col_sel,
    input wire          drop_pulse,
    output reg [5:0]   board_e00, output reg [5:0] board_e01, output reg [5:0] board_e02, c
    output reg [5:0]   board_e10, output reg [5:0] board_e11, output reg [5:0] board_e12, c
    output reg [5:0]   board_e20, output reg [5:0] board_e21, output reg [5:0] board_e22, c
    output reg [5:0]   board_e30, output reg [5:0] board_e31, output reg [5:0] board_e32, c
    output reg [31:0]  score,
    output reg         game_over,
    output reg         game_win
);
// LFSR for spawn distribution (16-bit maximal tap: x^16 + x^14 + x^13 + x^11 + 1)
reg [15:0] lfsr;
wire lfsr_fb = lfsr[~3_15] ^ lfsr[~3_13] ^ lfsr[~3_12] ^ lfsr[~3_10];
always @ (posedge clk or negedge rst_n) begin
    if (!rst_n) lfsr <= 16'h1ACE;
    else         lfsr <= {lfsr[14:0], lfsr_fb};
end [web:42][web:49]

// Convenience accessors
function [5:0] get_cell(input [1:0] r, input [1:0] c);
begin
    case ({r,c})
        4'h0: get_cell=board_e00; 4'h1: get_cell=board_e01; 4'h2: get_cell=board_e02; 4'h
        4'h4: get_cell=board_e10; 4'h5: get_cell=board_e11; 4'h6: get_cell=board_e12; 4'h
        4'h8: get_cell=board_e20; 4'h9: get_cell=board_e21; 4'hA: get_cell=board_e22; 4'h
        4'hC: get_cell=board_e30; 4'hD: get_cell=board_e31; 4'hE: get_cell=board_e32; 4'h
    endcase
end
endfunction
task set_cell(input [1:0] r, input [1:0] c, input [5:0] v);
begin
    case ({r,c})
        4'h0: board_e00=v; 4'h1: board_e01=v; 4'h2: board_e02=v; 4'h3: board_e03=v;
        4'h4: board_e10=v; 4'h5: board_e11=v; 4'h6: board_e12=v; 4'h7: board_e13=v;
        4'h8: board_e20=v; 4'h9: board_e21=v; 4'hA: board_e22=v; 4'hB: board_e23=v;
        4'hC: board_e30=v; 4'hD: board_e31=v; 4'hE: board_e32=v; 4'hF: board_e33=v;
    endcase
end
endtask [web:12]

// FSM
typedef enum logic [3:0] {S_RST, S_IDLE, S_CHECK_FULL, S_SPAWN, S_FALL, S_LANDED, S_MEF
state_t st;
reg [1:0] cur_col;

```

```

reg [1:0] r_pos;           // current falling row
reg      merging;
reg [31:0] add_score;

// Reset and clear
integer rr, cc;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        st<=S_RST; score<=0; game_over<=0; game_win<=0;
        board_e00<=0; board_e01<=0; board_e02<=0; board_e03<=0;
        board_e10<=0; board_e11<=0; board_e12<=0; board_e13<=0;
        board_e20<=0; board_e21<=0; board_e22<=0; board_e23<=0;
        board_e30<=0; board_e31<=0; board_e32<=0; board_e33<=0;
    end else begin
        case (st)
            S_RST: begin
                score<=0; game_over<=0; game_win<=0;
                board_e00<=0; board_e01<=0; board_e02<=0; board_e03<=0;
                board_e10<=0; board_e11<=0; board_e12<=0; board_e13<=0;
                board_e20<=0; board_e21<=0; board_e22<=0; board_e23<=0;
                board_e30<=0; board_e31<=0; board_e32<=0; board_e33<=0;
                st<=S_IDLE;
            end
            S_IDLE: begin
                if (drop_pulse) begin
                    cur_col <= col_sel;
                    st <= S_CHECK_FULL;
                end
            end
            S_CHECK_FULL: begin
                if (get_cell(2'd0, cur_col)!=0) begin
                    // policy: immediate game over if cannot spawn
                    game_over <= 1'b1;
                    st <= S_GAMEOVER;
                end else st <= S_SPAWN;
            end
            S_SPAWN: begin
                // 90/10 roughly using 4 MSBs: <14 -> 2, else 4
                // exact 90/10 could use 10-bit threshold
                set_cell(2'd0, cur_col, (lfsr[15:12] < 4'd14) ? 6'd1 : 6'd2);
                r_pos <= 2'd0;
                st <= S_FALL;
            end
            S_FALL: begin
                if (r_pos < 2'd3 && get_cell(r_pos+1, cur_col)==0) begin
                    set_cell(r_pos+1, cur_col, get_cell(r_pos, cur_col));
                    set_cell(r_pos, cur_col, 6'd0);
                    r_pos <= r_pos + 2'd1;
                    st <= S_FALL;
                end else begin
                    st <= S_LANDED;
                end
            end
            S_LANDED: begin
                merging <= 1'b0;
                add_score <= 32'd0;
            end
        endcase
    end
end

```

```

    st <= S_MERGE;
end
S_MERGE: begin
    // Extract column top->bottom into v[0..3]
    reg [5:0] v0,v1,v2,v3;
    v0 = get_cell(2'd0,cur_col);
    v1 = get_cell(2'd1,cur_col);
    v2 = get_cell(2'd2,cur_col);
    v3 = get_cell(2'd3,cur_col);

    // compress toward bottom
    automatic reg [5:0] t[0:3];
    integer k;
    t[^3_0]=v0; t[^3_1]=v1; t[^3_2]=v2; t[^3_3]=v3;
    // bubble zeros upward 3 passes
    integer pass, i;
    for (pass=0; pass<3; pass=pass+1) begin
        for (i=3; i>0; i=i-1) begin
            if (t[i]==0) begin t[i]=t[i-1]; t[i-1]=0; end
        end
    end

    // merge bottom-up, single pass, skip-on-merge
    reg merged_any;
    merged_any = 1'b0;
    i = 3;
    while (i>0) begin
        if (t[i]!=0 && t[i]==t[i-1]) begin
            t[i] = t[i] + 6'd1;           // exponent++
            t[i-1] = 6'd0;
            // score += 2^(new exponent)
            add_score <= add_score + (32'd1 << t[i]);
            merged_any = 1'b1;
            i = i - 2;                  // skip next (already consumed)
        end else begin
            i = i - 1;
        end
    end
end

// compress again toward bottom
for (pass=0; pass<3; pass=pass+1) begin
    for (i=3; i>0; i=i-1) begin
        if (t[i]==0) begin t[i]=t[i-1]; t[i-1]=0; end
    end
end

// write-back staged into temp regs; advance to WRITE
v0 = t[^3_0]; v1 = t[^3_1]; v2 = t[^3_2]; v3 = t[^3_3];
set_cell(2'd0,cur_col, v0);
set_cell(2'd1,cur_col, v1);
set_cell(2'd2,cur_col, v2);
set_cell(2'd3,cur_col, v3);
if (merged_any) merging <= 1'b1;
st <= S_WRITE;
end
S_WRITE: begin

```

```

        score <= score + add_score;
        st <= S_CHECK_END;
    end
    S_CHECK_END: begin
        // Win if any cell exponent >= 11 (value 2048)
        game_win <= (board_e00>=11) || (board_e01>=11) || (board_e02>=11) || (board_e03>=11) ||
                    (board_e10>=11) || (board_e11>=11) || (board_e12>=11) || (board_e13>=11) ||
                    (board_e20>=11) || (board_e21>=11) || (board_e22>=11) || (board_e23>=11) ||
                    (board_e30>=11) || (board_e31>=11) || (board_e32>=11) || (board_e33>=11);
        // Game over if all top cells non-zero (all columns full)
        if (board_e00!=0 && board_e01!=0 && board_e02!=0 && board_e03!=0) begin
            game_over <= 1'b1;
            st <= S_GAMEOVER;
        end else begin
            st <= S_IDLE;
        end
    end
    S_GAMEOVER: begin
        // wait for external reset
        st <= S_GAMEOVER;
    end
    default: st <= S_IDLE;
endcase
end
end
endmodule

```

VGA 640×480 timing (25 MHz enable from 100 MHz)

```

module vga_640x480_timing(
    input wire clk, input wire rst_n, input wire pix_en,
    output reg hs, output reg vs, output reg video_on,
    output reg [9:0] px, output reg [9:0] py
);
    // 640x480@60Hz nominal timing: H: 640,16,96,48; V: 480,10,2,33
    localparam H_VISIBLE=640, H_FP=16, H_SYNC=96, H_BP=48, H_TOTAL=640+16+96+48;
    localparam V_VISIBLE=480, V_FP=10, V_SYNC=2, V_BP=33, V_TOTAL=480+10+2+33;
    reg [9:0] x; reg [9:0] y;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin x<=0; y<=0; end
        else if (pix_en) begin
            if (x==H_TOTAL-1) begin x<=0; if (y==V_TOTAL-1) y<=0; else y<=y+10'd1; end
            else x<=x+10'd1;
        end
    end
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin hs<=1; vs<=1; video_on<=0; px<=0; py<=0; end
        else begin
            hs <= ~((x >= H_VISIBLE+H_FP) && (x < H_VISIBLE+H_FP+H_SYNC));
            vs <= ~((y >= V_VISIBLE+V_FP) && (y < V_VISIBLE+V_FP+V_SYNC));
            video_on <= (x < H_VISIBLE) && (y < V_VISIBLE);
            px <= x; py <= y;
        end
    end

```

```
end  
endmodule
```

Simple tile/score renderer

```
module renderer_2048_drop(  
    input wire      clk,  
    input wire      video_on,  
    input wire [9:0] px, input wire [9:0] py,  
    input wire [5:0] board_e00, input wire [5:0] board_e01, input wire [5:0] board_e02, i  
    input wire [5:0] board_e10, input wire [5:0] board_e11, input wire [5:0] board_e12, i  
    input wire [5:0] board_e20, input wire [5:0] board_e21, input wire [5:0] board_e22, i  
    input wire [5:0] board_e30, input wire [5:0] board_e31, input wire [5:0] board_e32, i  
    input wire [31:0] score,  
    input wire      game_over,  
    input wire      game_win,  
    output reg [11:0] rgb  
);  
// Board area: centered 400x400 box => each cell 100x100 pixels  
localparam X0=120, Y0=40, CELL=100, GAP=4;  
  
// Determine cell indices  
wire in_board = (px >= X0) && (px < X0+4*CELL) && (py >= Y0) && (py < Y0+4*CELL);  
wire [1:0] cx = (px - X0)/CELL;  
wire [1:0] cy = (py - Y0)/CELL;  
wire [6:0] lx = (px - X0)%CELL;  
wire [6:0] ly = (py - Y0)%CELL;  
  
// Fetch exponent for current pixel's cell  
reg [5:0] e;  
always @* begin  
    case ({cy,cx})  
        4'h0: e=board_e00; 4'h1: e=board_e01; 4'h2: e=board_e02; 4'h3: e=board_e03;  
        4'h4: e=board_e10; 4'h5: e=board_e11; 4'h6: e=board_e12; 4'h7: e=board_e13;  
        4'h8: e=board_e20; 4'h9: e=board_e21; 4'hA: e=board_e22; 4'hB: e=board_e23;  
        4'hC: e=board_e30; 4'hD: e=board_e31; 4'hE: e=board_e32; 4'hF: e=board_e33;  
    endcase  
end  
  
// Color LUT by exponent (simple)  
function [11:0] color_lut(input [5:0] ee);  
begin  
    case (ee)  
        0: color_lut = 12'hCCC; // empty  
        1: color_lut = 12'hEEE; // 2  
        2: color_lut = 12'hDDC; // 4  
        3: color_lut = 12'hCCB; // 8  
        4: color_lut = 12'hBB9; // 16  
        5: color_lut = 12'hAA7; // 32  
        6: color_lut = 12'h996; // 64  
        7: color_lut = 12'hC84; // 128  
        8: color_lut = 12'hC64; // 256  
        9: color_lut = 12'hC44; // 512  
    default: color_lut = 12'hC22; // 1024+, etc.  
    endcase  
end
```

```

    end
endfunction

// Borders
wire border = (lx < GAP) || (ly < GAP);

// Simple number glyph placeholder: draw a small square inside for non-empty
wire glyph = (lx > 20 && lx < 80 && ly > 20 && ly < 80) && (e!=0);

always @* begin
    if (!video_on) rgb = 12'h000;
    else if (!in_board) begin
        // HUD background
        if (game_over) rgb = 12'hF00; else if (game_win) rgb = 12'h0F0; else rgb = 12'h111;
    end else begin
        if (border) rgb = 12'h000;
        else begin
            rgb = color_lut(e);
            if (glyph) rgb = 12'h000;
        end
    end
end
endmodule

```

Notes and guidance

- Exponent storage and compress-merge-compress mirror canonical 2048 rules, with a single merge pass and skip-on-merge to ensure a tile merges at most once per drop along the landing column. This matches standard 2048 semantics for scoring $+ 2^{e_{\text{new}}}$ per created tile and triple-equal resolution favoring the pair nearest the landing side.^{[38] [39] [40]}
- The VGA timing module uses the well-known 640×480@60 Hz porch/sync counts and a 25 MHz pixel enable derived from 100 MHz; most monitors accept 25.000 MHz vs 25.175 MHz. Adjust with an MMCM if strict timing is needed.^{[41] [42] [43] [44]}
- The LFSR uses a common 16-bit maximal polynomial; substitute different taps as preferred, or use a parameterized LFSR design.^{[45] [46]}

What to extend next:

- Replace the simple glyph with a ROM font to render decimal values and score, or use seven-seg on-board for score mirroring.^[41]
- Add an “ignore full column” policy instead of instant game-over by branching S_CHECK_FULL back to S_IDLE.^[40]
- Integrate keyboard input if your board supports PS/2/USB HID in your lab framework.^[41]

**

1. <https://www.instructables.com/Video-Interfacing-With-FPGA-Using-VGA/>

2. <https://www.youtube.com/watch?v=4enWoVHCykI>

3. [https://en.wikipedia.org/wiki/2048_\(video_game\)](https://en.wikipedia.org/wiki/2048_(video_game))

4. <https://www.youtube.com/watch?v=MVu5OAFZhKA>

5. <https://www.youtube.com/watch?v=BcJU234Jh3k>
6. <https://www.youtube.com/watch?v=wnNGzMn5csQ>
7. <https://www.youtube.com/watch?v=igJqCPeWTlg>
8. <https://stackoverflow.com/questions/34523091/how-do-vga-control-signals-work-in-verilog-hdl>
9. <https://www.wikihow.com/Beat-2048>
10. <https://www.youtube.com/watch?v=eD4EyVq42mQ>
11. <https://www.instructables.com/Image-from-FPGA-to-VGA/>
12. <https://www.cs.columbia.edu/~sedwards/classes/2024/4840-spring/proposals/2048-Game.pdf>
13. https://www.reddit.com/r/FPGA/comments/7slqlh/implementing_a_vga_interface_in_verilog/
14. <https://aleksdev.github.io/2048>
15. https://github.com/BlagojeBlagojevic/vga_verilog
16. https://blog.waynejohnson.net/doku.php/generating_vga_with_an_fpga
17. https://www.reddit.com/r/2048/comments/1dxeobq/how_do_i_make_custom_merge_formulas_with_n_tiles/
18. <https://github.com/OmarMongy/VGA>
19. <https://narendiran1996.github.io/project-blogs/jekyll/update/2020/08/14/vgaController.html>
20. [https://en.wikipedia.org/wiki/2048_\(video_game\)](https://en.wikipedia.org/wiki/2048_(video_game))
21. <https://www.geeksforgeeks.org/utilities/2048-game/>
22. <https://www.prat0318.com/2048-tetris/>
23. <https://www.coolmathgames.com/0-number-drop>
24. <https://ktln2.org/2018/01/23/implementing-vga-in-verilog/>
25. https://github.com/BlagojeBlagojevic/vga_verilog
26. <https://www.youtube.com/watch?v=kQhkkqjGkFA>
27. <https://wizardofvegas.com/forum/questions-and-answers/math/31919-2048-game/>
28. <https://mathcookie17.github.io/The-2048-Power-Compendium/>
29. <https://www.dailytoolz.com/tetris-2048/>
30. <https://embdev.net/topic/469814>
31. <https://www.bluestacks.com/blog/game-guides/2248-number-puzzle-2048/np-beginners-guide-en.html>
32. <https://www.wikihow.com/Beat-2048>
33. <https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048>
34. https://www.reddit.com/r/FPGA/comments/1b6x0al/beginner_vga_controller_640x480_input_signal_out/
35. <https://www.youtube.com/watch?v=wnNGzMn5csQ>
36. https://blog.waynejohnson.net/doku.php/generating_vga_with_an_fpga
37. https://www.reddit.com/r/2048/comments/27nov1/so_i_merged_the_mechanics_of_tap_games_with_the/
38. <https://github.com/xiaowu0162/2048-on-FPGA>
39. <https://www.cs.columbia.edu/~sedwards/classes/2024/4840-spring/reports/2048-Game-report.pdf>
40. [https://en.wikipedia.org/wiki/2048_\(video_game\)](https://en.wikipedia.org/wiki/2048_(video_game))
41. <https://github.com/OmarMongy/VGA>

42. https://blog.waynejohnson.net/doku.php/generating_vga_with_an_fpga
43. <https://ktln2.org/2018/01/23/implementing-vga-in-verilog/>
44. <https://narendiran1996.github.io/project-blogs/jekyll/update/2020/08/14/vgaController.html>
45. <https://handland.com/lfsr-linear-feedback-shift-register/>
46. <https://vlsiverify.com/verilog/verilog-codes/lfsr/>
47. https://www.reddit.com/r/FPGA/comments/mjcnmf/how_can_i_make_a_merge_sort_for_fpga/
48. <https://arxiv.org/pdf/2504.14560.pdf>
49. <https://gregorulm.com/2048-in-90-lines-haskell/>
50. <https://www.youtube.com/watch?v=FtdlileVVag>
51. <https://dl.acm.org/doi/full/10.1145/3643681>
52. <https://www.analog.com/en/resources/design-notes/random-number-generation-using-lfsr.html>
53. <https://docs.amd.com/r/en-US/ug901-vivado-synthesis/Verilog-Coding-Example>
54. https://www.youtube.com/watch?v=Ea_KlhRqLkQ
55. <https://www.edaplayground.com/x/Uct>