Space Invaders on Basys3 FPGA

Don D. Le and Gaurav Kumar

Spring 2026

Contents

List of Figures

1 Introduction and Requirements

1.1 Lab Objective

The goal of this lab is to implement a simplified version of the classic Japanese arcade game Space Invaders on the Artix-7 FPGA of the Basys3 development board. The game runs on a 640×480 @ 60 Hz VGA display and is controlled using the onboard push buttons. The project builds on skills from earlier labs, clock division, synchronous state machines, input debouncing and introduces new challenges: VGA signal generation, sprite rendering from ROM bitmaps, pseudo-random number generation, real-time AABB collision detection, and a shield system.

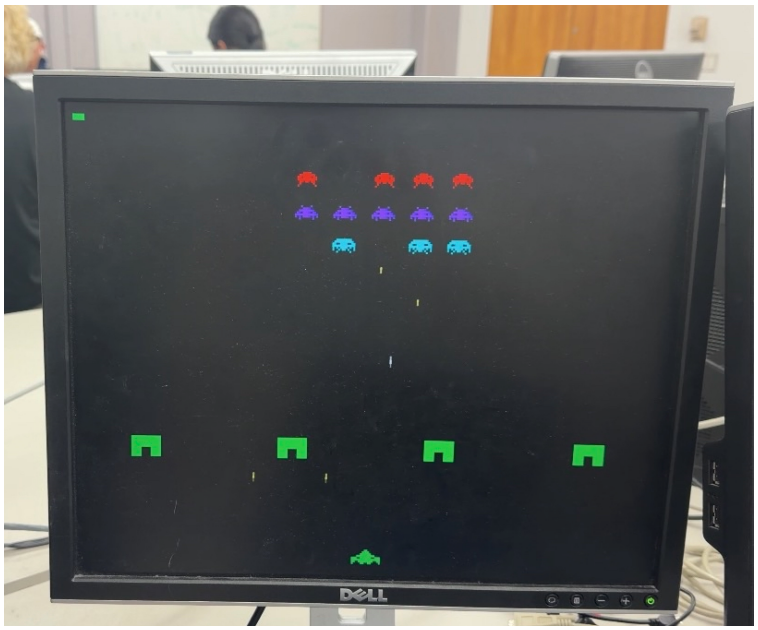


Figure 1: An illustration of our game

1.2 Design Requirements

Functional Requirements

The game consists of a single level. The player controls a spaceship at the bottom of the screen and must shoot down a 5×3 grid of villain ships that move side-to-side across the display. Four green shields provide cover in the middle of the play field. Villain ships periodically shoot back. The game ends when all villain ships are destroyed (player wins) or the player loses all three lives (game over). Table 1 summarizes the control inputs.

Table 1: Control input summary

Input	Type	Action
`BTNL`	Push button	Move player left
`BTNR`	Push button	Move player right
`BTNU`	Push button	Fire a player bullet
`BTND`	Push button	Hold 3 seconds to soft-reset the game
`BTNC`	Push button	Hard reset

Display Requirements

All objects are rendered combinationally on a 640×480 @ 60 Hz VGA display using a 25 MHz pixel clock. Table 2 lists every visible element.

Table 2: On-screen objects and their display properties

Object	Size (px)	Color	Notes
Player ship	32×16	Green	2× scaled from 16×8 ROM
Villain row 0	24×16	Red	Squid; 2× scaled from 12×8 ROM
Villain row 1	24×16	Purple	Crab
Villain row 2	24×16	Cyan	Octopus
Player bullets	2×8	White	Up to 3 simultaneous
Villain bullets	2×6	Yellow	Up to 12 simultaneous
Shields	32×22	Green	Arch cutout; 4 shields total
Lives HUD	12×6 each	Green	Top-left, 1 icon per remaining life
GAME OVER text	288×32	Red-white	8×8 glyphs scaled 4×
YOU WIN text	224×32	Yellow	Same glyph ROM
Background	full screen	Black	

Gameplay Requirements

- **Player**: moves ±4 px/tick, clamped to [0, 624]. Up to 3 player bullets may be active simultaneously. After taking damage the player is invulnerable and blinks for approximately 1 second (60 game ticks).
- **Villains**: 15 ships in a 5×3 grid share one base position. The group moves right 2 px/tick, reverses at the screen edge, and moves left. Destroyed villains are permanently removed.
- **Villain shooting**: every 45 game ticks (≈0.75 s at 60 Hz), up to three LFSR-selected living villains fire bullets downward. Up to 12 villain bullets may be active simultaneously.
- **Shields**: four arch-shaped shields at fixed horizontal positions stop bullets from both sides on contact. Shields are indestructible in this implementation.
- **Score**: displayed on the 7-segment display. Each villain destroyed adds 10 points.
- **Lives**: player starts with 3 lives shown as HUD icons on the VGA display.
- **End conditions**: all villains destroyed ⇒ "YOU WIN"; all lives lost ⇒ "GAME OVER". Both freeze all movement and overlay text on the VGA output.

2 Design Description

2.1 Top-Level Module Overview

The top-level module `top.v` is a structural wrapper that instantiates every submodule and connects them with wires. It also contains two pieces of logic that do not fit neatly into any submodule: the hard-reset synchronizer.

Hard reset synchronizer. The center button (`reset_btn`) is an asynchronous external input. A two-stage flip-flop chain synchronizes it into the `pix_clk` domain before it can be used anywhere in the design.

Soft reset. Holding `BTND` for 3 seconds triggers a game reset without interrupting VGA timing. A 27-bit counter running at 25 MHz saturates at 75,000,000 cycles (= 3 s). When saturated it asserts `soft_rst` for one cycle, which is ORed with the synchronized hard reset to form `game_rst`.

```
// Synchronize hard reset into pix_clk domain
reg [1:0] rst_sync;
always @(posedge pix_clk) rst_sync <= {rst_sync[0], reset_btn};
wire rst = rst_sync[1];

// Hold BTND for 3 sec -> soft reset (27-bit counter @ 25 MHz)
reg [26:0] hold_cnt;
reg        soft_rst;
always @(posedge pix_clk) begin
    if (rst) begin
        hold_cnt <= 27'd0;  soft_rst <= 1'b0;
    end else if (btnD_db) begin
        if (hold_cnt >= 27'd75_000_000) begin
            soft_rst <= 1'b1;           // assert for one cycle
            hold_cnt <= hold_cnt;       // saturate counter
        end else begin
            hold_cnt <= hold_cnt + 27'd1;
            soft_rst <= 1'b0;
        end
    end else begin
        hold_cnt <= 27'd0;  soft_rst <= 1'b0;
    end
end

wire game_rst = rst | soft_rst;
```

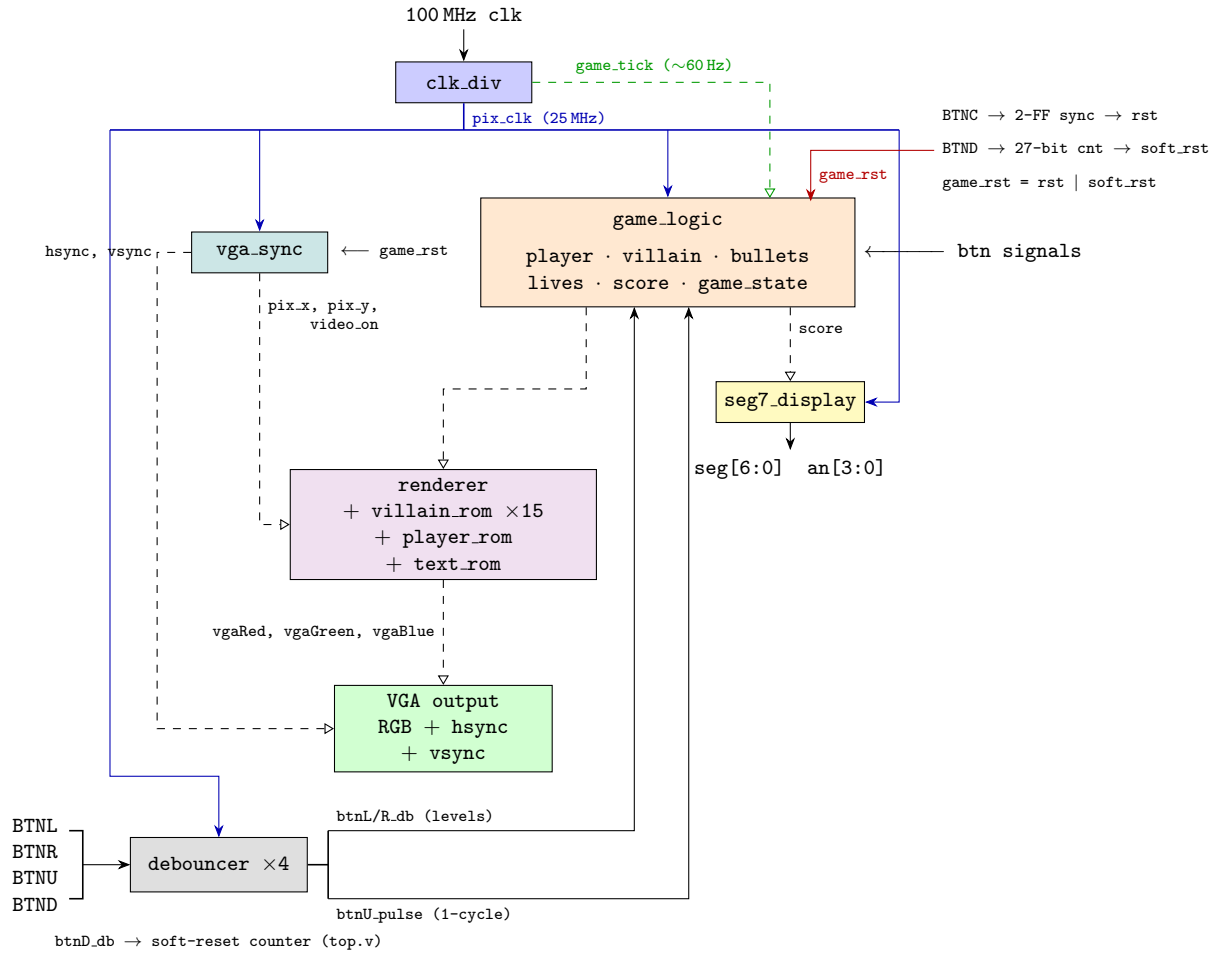


Figure 2: System architecture and main signal paths. Blue lines: `pix_clk` bus (25 MHz). Dashed arrows: data signals. Red arrow: reset.

Table 3: Key internal signal routing in `top.v`

Signal	Produced by	Consumed by
`pix_clk` (25 MHz)	`clk_div`	`vga_sync`, `game_logic`, `renderer`
`game_tick` (≈60 Hz)	`clk_div`	`game_logic`
`pix_x`, `pix_y`	`vga_sync`	`renderer`, `seg7_display`
`hsync`, `vsync`	`vga_sync`	VGA output pins
`video_on`	`vga_sync`	`renderer`
`btnL_db`, `btnR_db`	`debouncer`	`game_logic` (levels)
`btnU_pulse`	`debouncer`	`game_logic` (fire strobe)
game state bundle	`game_logic`	`renderer`, `seg7_display`
`vgaRed/Green/Blue`	`renderer`	VGA output pins

2.2 Clock Divider (`clk_div.v`)

Two derived clocks are generated from the 100 MHz board oscillator.

25 MHz pixel clock. A 2-bit counter toggles `pix_clk` on every odd count (counts 1 and 3), dividing the board clock by 4 with a clean 50% duty cycle.

Game tick (≈60 Hz). A 19-bit counter running on `pix_clk` asserts a single-cycle pulse every 416,666 pixel clocks (25 MHz/416,666 ≈ 60 Hz). All game-object positions and state registers update exactly once per `game_tick`; the renderer runs every pixel clock (25 MHz) but game objects only move at 60 Hz.

```
// 25 MHz pixel clock: toggle on odd counts of the 100 MHz clock
reg [1:0] div_cnt;
always @(posedge clk_in) begin
    div_cnt <= div_cnt + 2'd1;
    if (div_cnt[0]) pix_clk <= ~pix_clk;   // toggles at 1 and 3
end

// ~60 Hz game tick: single-cycle pulse every 416,666 pix_clk cycles
reg [18:0] tick_cnt;
always @(posedge pix_clk) begin
    if (tick_cnt >= 19'd416_666) begin
        tick_cnt  <= 19'd0;
        game_tick <= 1'b1;
    end else begin
        tick_cnt  <= tick_cnt + 19'd1;
        game_tick <= 1'b0;
    end
end
```

2.3 VGA Sync Generator (`vga_sync.v`)

Generates horizontal and vertical timing signals for a 640×480 @ 60 Hz VGA monitor. Two counters: `h_cnt` (0–799) and `v_cnt` (0–524) increment every pixel clock. When `h_cnt` reaches 799 it resets to 0 and `v_cnt` advances; when `v_cnt` reaches 524 it also resets.

Table 4: 640×480 @ 60 Hz VGA timing (implemented with `localparam`)

Region	Horizontal (px)	Vertical (lines)
Visible area	640	480
Front porch	16	10
Sync pulse	96	2
Back porch	48	33
Total	**800**	**525**

Both sync pulses are active-low. `video_on` is asserted only during the 640×480 visible window; the renderer outputs black whenever `video_on` is low, producing blanking borders automatically.

```
// Active-low hsync: low while h_cnt is in the sync-pulse window
hsync <= ~((h_cnt >= H_DISPLAY + H_FRONT) &&
           (h_cnt <  H_DISPLAY + H_FRONT + H_SYNC));
vsync <= ~((v_cnt >= V_DISPLAY + V_FRONT) &&
           (v_cnt <  V_DISPLAY + V_FRONT + V_SYNC));

assign video_on = (h_cnt < H_DISPLAY) && (v_cnt < V_DISPLAY);
assign pix_x    = h_cnt;
assign pix_y    = v_cnt;
```

2.4 Debouncer (`debouncer.v`)

A parameterized three-stage debouncer is instantiated once per button. The `CNT_WIDTH` parameter (set to 16) determines the debounce window; at 25 MHz, 2^{16} cycles = 2.6 ms.

1. **Two-stage synchronizer.** A shift register `sync[1:0]` moves the raw asynchronous button pin safely into the `pix_clk` domain.
2. **Saturating counter.** The counter increments while the synchronized input disagrees with the accepted output. When all bits are 1 (saturated), the new level is latched into `btn_out`.
3. **Edge detector.** A one-cycle `btn_pulse` is produced on the rising edge of `btn_out` for use as a fire strobe.

```
// Stage 1 - synchronize into pix_clk domain
reg [1:0] sync;
always @(posedge clk)
    sync <= {sync[0], btn_in};
wire btn_sync = sync[1];

// Stage 2 - saturating debounce counter
if (btn_sync != btn_out) begin
    cnt <= cnt + 1'b1;
    if (&cnt) btn_out <= btn_sync;   // all bits set -> accept
end else
    cnt <= {CNT_WIDTH{1'b0}};

// Stage 3 - single-cycle rising-edge pulse
btn_prev  <= btn_out;
btn_pulse <= btn_out & ~btn_prev;
```

Movement buttons (`BTNL`, `BTNR`) use `btn_out` (sustained level) since the game logic needs to know whether the button is currently held. The fire button (`BTNU`) uses `btn_pulse` so that holding the button fires exactly one bullet per press.

2.5 Game Logic (`game_logic.v`)

The central state machine. All game-object positions and flags are registered; the sequential block advances them once per `game_tick`. The LFSR advances every clock cycle regardless of `game_tick`, ensuring the RNG

always has a fresh value when a volley fires.

Villain Position Functions

Rather than storing 15 individual (x,y) pairs, all villains share a single base position. Column and row are recovered from the villain index using a lookup table function, keeping the movement logic to a single register update:

```
// Col look-up: index -> 0..4, then scale by cell width
function [9:0] vill_x;
    input [3:0] idx;
    reg [2:0] col;
    begin
        case (idx)
            4'd0,4'd5, 4'd10: col = 3'd0;
            4'd1,4'd6, 4'd11: col = 3'd1;
            4'd2,4'd7, 4'd12: col = 3'd2;
            4'd3,4'd8, 4'd13: col = 3'd3;
            default:          col = 3'd4;
        endcase
        vill_x = villain_base_x + ({7'd0, col} * V_CELL_W);
    end
endfunction

// Row look-up: groups of 5 -> row 0, 1, 2
function [9:0] vill_y;
    input [3:0] idx;
    reg [1:0] row;
    begin
        if      (idx <= 4'd4) row = 2'd0;
        else if (idx <= 4'd9) row = 2'd1;
        else                  row = 2'd2;
        vill_y = villain_base_y + ({8'd0, row} * V_CELL_H);
    end
endfunction
```

AABB Collision Detection

All hit tests use an axis-aligned bounding box (AABB) function. The wires are purely combinational and evaluated every clock cycle against the current registered positions; the sequential block then acts on them once per `game_tick`.

Two axis-aligned rectangles A and B overlap if and only if all four interval tests pass simultaneously: A's right edge must be to the right of B's left edge, A's left edge to the left of B's right edge, and the same in the vertical axis. Figure 3 illustrates the geometry.

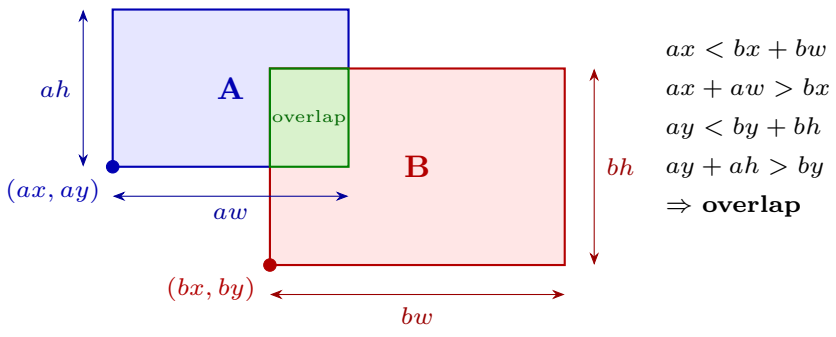


Figure 3: AABB overlap test. Rectangles A (blue) and B (red) overlap (green region) when all four interval conditions hold simultaneously.

```
// Returns true if rectangle A overlaps rectangle B.
function aabb_overlap;
    input [9:0] ax, ay, aw, ah;
    input [9:0] bx, by, bw, bh;
    begin
        aabb_overlap = (ax < bx + bw) && (ax + aw > bx) &&
                       (ay < by + bh) && (ay + ah > by);
    end
endfunction

// Per-villain hit vectors (combinational, generated in a for-loop)
generate
    for (gi = 0; gi < 15; gi = gi + 1) begin : g_pb_hit
        assign pb_vill_hit_vec_0[gi] = pb_active[0] && villain_alive[gi] &&
            aabb_overlap(pbx[0], pby[0], PB_W, PB_H,
                         vx[gi],  vy[gi],  V_W,  V_H);
    end
endgenerate
wire pb_hit_0 = |pb_vill_hit_vec_0;   // any villain hit by bullet 0?
```

When `pb_hit_0` is asserted the sequential block deactivates the bullet and uses a `casez` on `pb_vill_hit_vec_0` to clear the lowest-priority matching `villain_alive` bit.

LFSR Pseudo-Random Villain Shooter

A 16-bit Galois LFSR advances every clock cycle. Three 4-bit slices are extracted and clamped to the range 0–14 to select which villains fire each volley:

```
// Galois LFSR: taps at bits 15, 13, 12, 10
lfsr <= {lfsr[14:0], lfsr[15] ^ lfsr[13] ^ lfsr[12] ^ lfsr[10]};

// Clamp each 4-bit slice to 0..14 (avoid mod-15 in synthesis)
wire [3:0] shoot1 = (lfsr[3:0]  >= 4'd15) ? lfsr[3:0]  - 4'd15 : lfsr[3:0];
wire [3:0] shoot2 = (lfsr[7:4]  >= 4'd15) ? lfsr[7:4]  - 4'd15 : lfsr[7:4];
wire [3:0] shoot3 = (lfsr[11:8] >= 4'd15) ? lfsr[11:8] - 4'd15 : lfsr[11:8];
```

Every 45 game ticks (`SHOOT_PERIOD`), if the selected villain is alive its bullet is placed into the first free slot of a dedicated slot range (shoot1 → slots 0–3, shoot2 → slots 4–7, shoot3 → slots 8–11). Separating the slot ranges prevents concurrent volley assignments from overwriting each other.

Player Bullet Firing

A fire pulse from the debouncer latches a new bullet into the lowest free slot. Firing is allowed on any clock cycle, not just game ticks, so the response feels immediate:

```
if (game_state == ST_PLAY && fire_pulse && player_alive) begin
    if (!pb_active[0]) begin
        pb_active[0] <= 1'b1;
        pb_x_0 <= player_x + (PLAYER_W >> 1) - (PB_W >> 1);
        pb_y_0 <= PLAYER_Y - PB_H;
    end else if (!pb_active[1]) begin
        pb_active[1] <= 1'b1;
        pb_x_1 <= player_x + (PLAYER_W >> 1) - (PB_W >> 1);
        pb_y_1 <= PLAYER_Y - PB_H;
    end else if (!pb_active[2]) begin
        pb_active[2] <= 1'b1;
        pb_x_2 <= player_x + (PLAYER_W >> 1) - (PB_W >> 1);
        pb_y_2 <= PLAYER_Y - PB_H;
    end
end
```

Shield (Wall) Collision

Four indestructible shields are placed at fixed horizontal positions (X = 64, 224, 384, 544) at Y = 330. Each shield is 32×22 px with an arch cutout (10 px wide, 10 px tall) removed from the bottom center, mimicking the classic Space Invaders bunker shape. In the real space invaders, the shield is supposed to be breakable by both villain and player spaceship bullets. Implementing the shield this way was beyond the scope of this

lab, so we decided to make the shield indestructible. Bullets from both sides deactivate on contact with any solid region of a shield:

```
// Check if bullet (bx,by,bw,bh) hits any solid region of the wall at wx.
// Solid = top block (full width, above arch) + left & right pillars.
function bullet_hits_wall;
    input [9:0] bx, by, bw, bh, wx;
    begin
        bullet_hits_wall =
            aabb_overlap(bx,by,bw,bh, wx, WALL_Y, WALL_W, WALL_ARCH_Y) ||
            aabb_overlap(bx,by,bw,bh, wx, WALL_Y+WALL_ARCH_Y,
                         WALL_ARCH_X, WALL_H-WALL_ARCH_Y) ||
            aabb_overlap(bx,by,bw,bh,
                         wx+WALL_ARCH_X+WALL_ARCH_W, WALL_Y+WALL_ARCH_Y,
                         WALL_W-WALL_ARCH_X-WALL_ARCH_W, WALL_H-WALL_ARCH_Y);
    end
endfunction

function any_wall_hit;
    input [9:0] bx, by, bw, bh;
    begin
        any_wall_hit = bullet_hits_wall(bx,by,bw,bh, WALL0_X) ||
                       bullet_hits_wall(bx,by,bw,bh, WALL1_X) ||
                       bullet_hits_wall(bx,by,bw,bh, WALL2_X) ||
                       bullet_hits_wall(bx,by,bw,bh, WALL3_X);
    end
endfunction
```

Damage Flash and Invulnerability

When a villain bullet hits the player a 7-bit `blink_cnt` is loaded with 60 and counts down one step per game tick. Bit 2 of the counter toggles every 4 ticks (≈67 ms), creating a visible blink. While `blink_cnt` is nonzero the player is also invulnerable meaning subsequent villain bullets pass through:

```
// Blink timer: counts down at game-tick rate, bit[2] drives the blink signal
if (blink_cnt > 7'd0) begin
    blink_cnt    <= blink_cnt - 7'd1;
    player_blink <= blink_cnt[2];   // toggles every 4 ticks (~67 ms)
end else
    player_blink <= 1'b0;

// Villain bullet hit: only register if player is NOT currently flashing
if (vb_any_hit && player_alive && blink_cnt == 7'd0) begin
    blink_cnt <= 7'd60;             // ~1 sec invulnerability window
    if (lives > 2'd0) lives <= lives - 2'd1;
    // deactivate the hitting bullet (lowest-index priority via casez)
end
```

Game State Machine

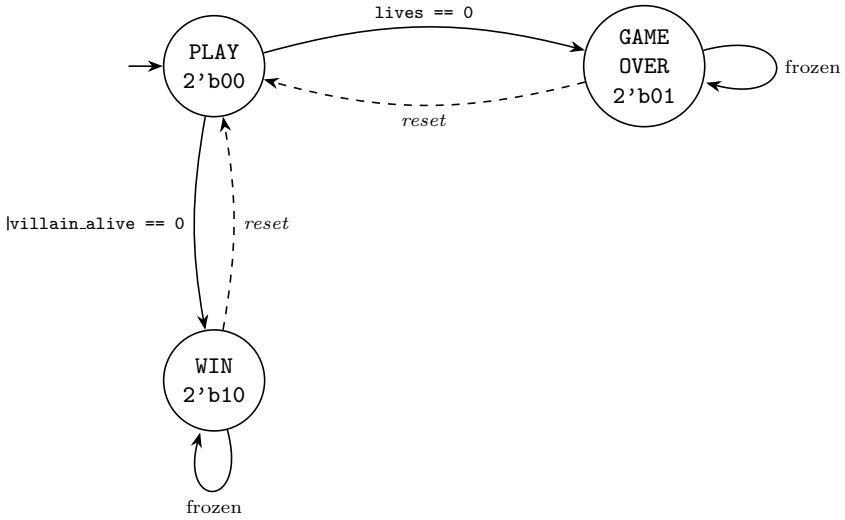


Figure 4: Top-level game FSM. Dashed arrows indicate reset transitions.

Table 5: Top-level game FSM state summary

State	Encoding	Transition
`PLAY`	`2'b00`	→ `GAME_OVER` when `lives == 0`
		→ `WIN` when `|villain_alive == 0`
`GAME_OVER`	`2'b01`	frozen until reset
`WIN`	`2'b10`	frozen until reset

2.6 Renderer (`renderer.v`)

A large purely combinational module. Every pixel clock it receives `pix_x`, `pix_y`, and the full game state, and outputs a 12-bit RGB value (three 4-bit channels). The final color is chosen by a priority-encoded `if/else` chain:

Table 6: Renderer priority order (highest wins)

Priority	Layer	Color
1 (highest)	Blanking (`video_on` low)	Black
2	Text overlay (GAME OVER / YOU WIN)	Red-white / Yellow
3	Lives HUD icons	Green
4	Player ship (hidden while `player_blink`)	Green
5	Player bullets	White
6	Villain bullets	Yellow
7	Shields (arch-shaped, 4 total)	Green
8	Villain row 0 (squid)	Red
9	Villain row 1 (crab)	Purple
10	Villain row 2 (octopus)	Cyan
11 (lowest)	Background	Black

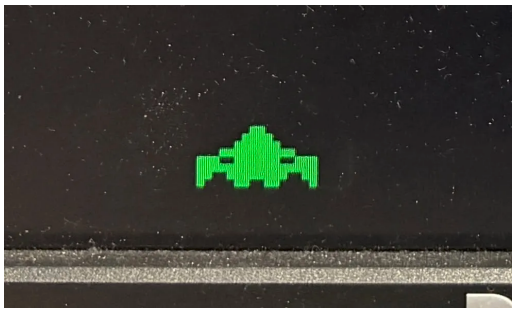
Sprite Rendering

For every sprite-based object the renderer checks whether the current pixel is inside the bounding box, computes the sub-pixel offset, and looks up the corresponding bit in the sprite ROM. Player and villain sprites are displayed at 2× scale; the sub-pixel coordinates are right-shifted by 1 before the ROM lookup so each ROM pixel maps to a 2×2 block:

```
// Player sprite: 32x16 on screen from a 16x8 ROM (2x scale)
wire [4:0] player_sp_x = pix_x[4:0] - player_x[4:0];
wire [3:0] player_sp_y = pix_y[3:0] - PLAYER_Y[3:0];
player_rom u_player_rom (
    .col (player_sp_x[4:1]),   // >> 1 to get 0..15
    .row (player_sp_y[3:1]),   // >> 1 to get 0..7
    .pix (player_rom_pix)
);
wire in_player = player_alive && !player_blink &&
    (pix_x >= player_x) && (pix_x < player_x + PLAYER_W) &&
    (pix_y >= PLAYER_Y) && (pix_y < PLAYER_Y + PLAYER_H) &&
    player_rom_pix;
```

Fifteen villain ROM instances are generated in a `generate` loop, one per villain, each with the appropriate `ATYPE` parameter (`0`=squid, `1`=crab, `2`=octopus) baked in at synthesis time. All 15 are evaluated in parallel every pixel clock.

(a) Three different types of aliens



(b) Our spaceship

Figure 5: Game sprites.

Shield Rendering

Shields are drawn with the same arch cutout used in the game logic collision checks, so the visual shape and the collision boundary are guaranteed to match:

```
function in_wall_at;
    input [9:0] px, py, wx;
    reg in_box, in_arch;
    begin
        in_box  = (py >= WALL_Y) && (py < WALL_Y + WALL_H) &&
                  (px >= wx)     && (px < wx + WALL_W);
        in_arch = (py >= WALL_Y + WALL_ARCH_Y) &&
                  (px >= wx + WALL_ARCH_X) &&
                  (px < wx + WALL_ARCH_X + WALL_ARCH_W);
        in_wall_at = in_box && !in_arch;
    end
endfunction

wire in_wall = in_wall_at(pix_x, pix_y, WALL0_X) |
               in_wall_at(pix_x, pix_y, WALL1_X) |
               in_wall_at(pix_x, pix_y, WALL2_X) |
               in_wall_at(pix_x, pix_y, WALL3_X);
```

Illustration below:

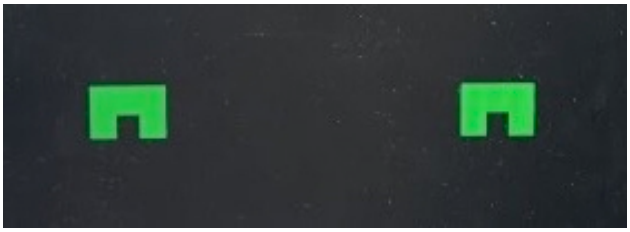


Figure 6: The Shields

Text Overlay

"GAME OVER" (9 characters) and "YOU WIN" (7 characters) are rendered by looking up 8×8 glyphs from `text_rom.v` and scaling each glyph 4× on screen (to 32×32 px per character). The renderer computes which character cell `pix_x` falls in, maps the column index to a glyph code, and reads a single pixel from the ROM:

```
// Column index within the banner string (divide x-offset by 32 = >> 5)
wire [4:0] col_idx5 = (pix_x - txt_x_base) >> 5;
// Sub-pixel within the 32x32 cell, scaled down to 8x8 ROM coords (>> 2)
wire [2:0] glyph_col = (pix_x - txt_x_base)[4:2];
wire [2:0] glyph_row = (pix_y - TXT_Y)[4:2];

// "GAME OVER": G A M E ' ' O V E R  (character codes from text_rom)
// "YOU WIN":   Y O U ' ' W I N
text_rom u_txt (.ch(glyph_ch), .col(glyph_col), .row(glyph_row), .pix(glyph_pix));
wire in_text = show_text && txt_x_in && txt_y_in && glyph_pix;
```

Illustration below:

(a) Winning Screen

(b) Game Over Screen

Figure 7: End-of-game screens.

2.7 7-Segment Display (`seg7_display.v`)

Displays the score (0–9999) on the four-digit 7-segment display. The 14-bit binary score is converted to four BCD digits using constant-divisor division (synthesized into LUT arithmetic by Vivado). An 18-bit counter selects which digit is active at ≈95 Hz per digit, well above the 50–700 Hz scan range needed for persistence-of-vision.

```
// BCD extraction (constant divisors -> synthesized as LUT chains)
wire [13:0] sc = (score > 14'd9999) ? 14'd9999 : score;
wire [3:0] d0 =  sc        % 14'd10;
wire [3:0] d1 = (sc / 14'd10)   % 14'd10;
```

```
wire [3:0] d2 = (sc / 14'd100)  % 14'd10;
wire [3:0] d3 = (sc / 14'd1000) % 14'd10;

// 18-bit mux counter, top 2 bits select active digit (~95 Hz per digit)
wire [1:0] digit_sel = mux_cnt[17:16];
always @* begin
    case (digit_sel)
        2'd0: begin an = 4'b1110; cur_digit = d0; end
        2'd1: begin an = 4'b1101; cur_digit = d1; end
        2'd2: begin an = 4'b1011; cur_digit = d2; end
        2'd3: begin an = 4'b0111; cur_digit = d3; end
    endcase
end
```

3 Simulation Documentation

3.1 Test Strategy

Before programming the board we verified the design in Vivado's behavioral simulator. Because a full 640×480 @ 60 Hz frame takes 420,000 pixel clocks, we ran simulations at full clock speeds but reduced the VGA counter terminal values and game-tick interval by a factor of 10–100 to keep run times manageable. The simulations targeted three areas:

- **VGA timing**: confirming `h_cnt` wraps at 800, `v_cnt` wraps at 525, `hsync`/`vsync` assert for the correct number of cycles, and `video_on` goes low during all blanking intervals.
- **Game logic**: player movement clamping, bullet firing and travel, villain movement reversal at screen edges, LFSR-driven volley spacing, wall/shield collision, and AABB collision correctness.
- **End conditions**: FSM transition to `GAME_OVER` on lives reaching zero and to `WIN` on all `villain_alive` bits clearing.

Specific test cases included:

- `h_cnt` and `v_cnt` roll over at 799 and 524 respectively on every pixel clock.
- `video_on` is low during all blanking intervals and high in the visible 640×480 region.
- Player moves 4 px/tick when a direction button is held; stops at 0 (left) and 624 (right).
- Player bullet activates in the lowest free slot on `btnU_pulse`; travels upward 4 px/tick.
- Player bullet deactivates on exiting the top of the screen or hitting a wall.
- Villain base position shifts right 2 px/tick; reverses when the full grid width would exceed 640 px.
- Villain bullet volley fires after exactly 45 game ticks; three bullets appear in distinct slot ranges.
- LFSR value changes on every clock cycle and is non-static.
- AABB collision registers a hit when a player bullet overlaps a villain bounding box.
- Villain `alive` bit clears and score increments by 10 on a confirmed hit.
- Lives decrement when a villain bullet hits the player (only when `blink_cnt == 0`).
- `player_blink` toggles at ≈67 ms and the player becomes invulnerable for 60 ticks.
- FSM enters `GAME_OVER` when `lives` reaches 0; enters `WIN` when all alive bits clear.
- Bullet hitting a wall (any of the four shield positions) deactivates the bullet on the next tick.

3.2 Bugs Found

Villain Direction Reversal Using Full Grid Width

The initial boundary check used the full $5 \times 40 = 200$ px column span regardless of how many villains were still alive. Once some were destroyed, the rightmost *living* column moved closer to the center, but the reversal threshold was still computed from the fixed 200 px span, so the group reversed earlier than it should, shrinking the movement range as villains were killed.

The fix was to move the boundary check into `game_logic` and compute the effective right edge from the fixed `V_GRID_W` against `villain_base_x` only, ignoring dead villains:

```
if (villain_dir == DIR_RIGHT) begin
    // Reverse when the full grid's right edge would leave the screen
    if (villain_base_x + V_GRID_W + VILLAIN_SPD >= SCREEN_W)
        villain_dir    <= DIR_LEFT;
    else
        villain_base_x <= villain_base_x + VILLAIN_SPD;
end else begin
    if (villain_base_x < VILLAIN_SPD)
        villain_dir    <= DIR_RIGHT;
    else
        villain_base_x <= villain_base_x - VILLAIN_SPD;
end
```

Villain Bullet Slot Overwrite on Duplicate LFSR Picks

When two of the three LFSR slices produced the same villain index, both volley spawns would write to overlapping slot ranges. In practice the second write would clobber the first bullet's starting position, leaving one slot permanently occupied and one expected bullet not appearing.

The fix was to assign each pick to a dedicated, non-overlapping slot range (shoot1 $\rightarrow$ slots 0–3, shoot2 $\rightarrow$ slots 4–7, shoot3 $\rightarrow$ slots 8–11). Even if two picks select the same villain, their bullets are placed independently in separate ranges and cannot overwrite each other.

Text Overlay Rendered Behind Villains

In the initial priority ordering the villain rendering check appeared before the text overlay check. When the game transitioned to `GAME_OVER` or `WIN`, villain sprite pixels that happened to overlap the text banner would punch holes through the characters, making them partially unreadable.

The fix was to move the text-overlay `if` clause above the villain clause in the combinational color selector, so the banner always wins over any sprite pixel.

3.3 Discussion of Results

After applying all three fixes the simulation confirmed correct behavior for every test case. The VGA timing waveforms showed `hsync` pulsing at exactly 800 pixel-clock intervals and `vsync` at $800 \times 525 = 420{,}000$ intervals. The game logic produced correct AABB collision responses, clean FSM transitions, and properly spaced villain volleys. On hardware, the full 640×480 image was stable with no display artifacts, collision responses were immediate, the shields blocked bullets correctly, and both end-screen overlays were fully legible without sprite bleed-through.

4 Conclusion

This lab resulted in a fully functional Space Invaders game running on the Basys3 FPGA board, rendering to a 640×480 VGA monitor at 60 Hz with real-time game logic, sprite ROM rendering, shield collision, and a score display on the 7-segment display.

Three main difficulties were encountered. The villain boundary-reversal bug looked like an off-by-one in the boundary constant, but the real cause was that the fixed grid width was correct, the issue was only

visible once some villains were destroyed and the effective extent changed. The bullet-slot overwrite was an infrequent, subtle glitch (one expected bullet occasionally not appearing) that only surfaced when two LFSR slices produced the same index; segregating slot ranges eliminated the race entirely. The text-overlay priority issue was straightforward once we listed the priority layers in order and noticed the villain check appeared first.

The most important architectural decision was separating the pixel clock (25 MHz) from the game-tick clock ($\approx$60 Hz). The renderer must produce a color for every single pixel in the 640$\times$480 frame, while game objects only need to move 60 times per second. Mixing these two rates would have either made the display unusable or the game unplayably fast. Using a single shared base position for the entire villain grid also kept movement logic to one register update per tick regardless of how many villains remained alive.