CSEE 4840 · EMBEDDED SYSTEMS

Plants vs Zombies on the DE1-SoC

A Cyclone V SoC final-project report

TARGET Terasic DE1-SoC · 640×480 @ 60 Hz

Hao Cai
hc3612@columbia.edu

Yabkun Li
yl6022@columbia.edu

Chenhao Yang
cy2822@columbia.edu

Zhenghang Zhao
zz3410@columbia.edu

Table of Contents

1	Introduction	The game in one paragraph, board-level architecture, and scope.
2	System Architecture	HPS / FPGA split, one register file, 60 Hz on both sides.
3	Hardware	VGA timing, sprite ROMs, the two-stage entity drawer, palette.
4	Software	60 Hz main loop, evdev input, the simulation, the renderer, the driver.
5	The Register Interface	The 51-word ABI: ioctl, layout, one call traced end to end.
6	One Frame, End to End	From A-button press to a peashooter lighting up on the monitor.
7	Build and Run	Hardware and software build flows, getting it onto the board, controls.
8	Appendix: Source Code	Every team-written .c / .sv / .h file inlined for offline reading.

1 Introduction

Press **Space** on a keyboard plugged into the DE1-SoC and a peashooter pops up at the lawn cell your cursor was hovering over. A few seconds later it spits a green pea across the row at a zombie that we drew in real time by composing sprite pixels on the FPGA fabric while the VGA beam scanned past. There is no frame buffer in DRAM and no GPU library involved; every pixel on the 640×480 @ 60 Hz monitor is decided by a SystemVerilog module the same cycle it leaves the DAC. We built that, and the rest of this report explains how.

The game in one paragraph

The lawn is a 4-row by 8-column grid of 64-pixel cells (`sw/pvz.h:29-33`, `hw/bg_grid.sv:28-31`). You start with 100 sun (`sw/game.h:41`) and earn 25 more every 8 seconds, plus whatever your sunflowers donate (`sw/game.h:42-43`). Plants cost 50 sun each: a peashooter fires one pea every 2 seconds (`sw/game.h:18-20`) and a sunflower passively boosts income. Five zombies spawn from the right edge at random 8–15 second intervals (`sw/game.h:29-31`), shamble leftward at roughly 20 pixels per second (`sw/game.h:26`), and take three hits to die (`sw/game.h:25`). You drive a cursor with the keyboard arrow keys or a gamepad D-pad; Space or A plants, D or B removes, Tab or shoulder buttons flips the selected plant type, and Esc or Start quits. Kill all five zombies and you win. Let one reach column zero and you lose.

Board-level architecture

The Cyclone V SoC fuses a dual-core ARM Cortex-A9 (the "HPS") with FPGA fabric on one die. We run Linux on the HPS — userspace game logic in `sw/main.c` plus a small kernel driver in `sw/pvz_driver.c` — and host a custom GPU peripheral on the FPGA, defined by `hw/pvz_top.sv:1-25`. The two halves talk through a single Avalon-MM register file mapped into the HPS physical address space at `0xff200000`; software writes one 32-bit word per game object per frame, and the FPGA reads those words combinationally as the VGA beam asks "what color goes at (x, y)?"

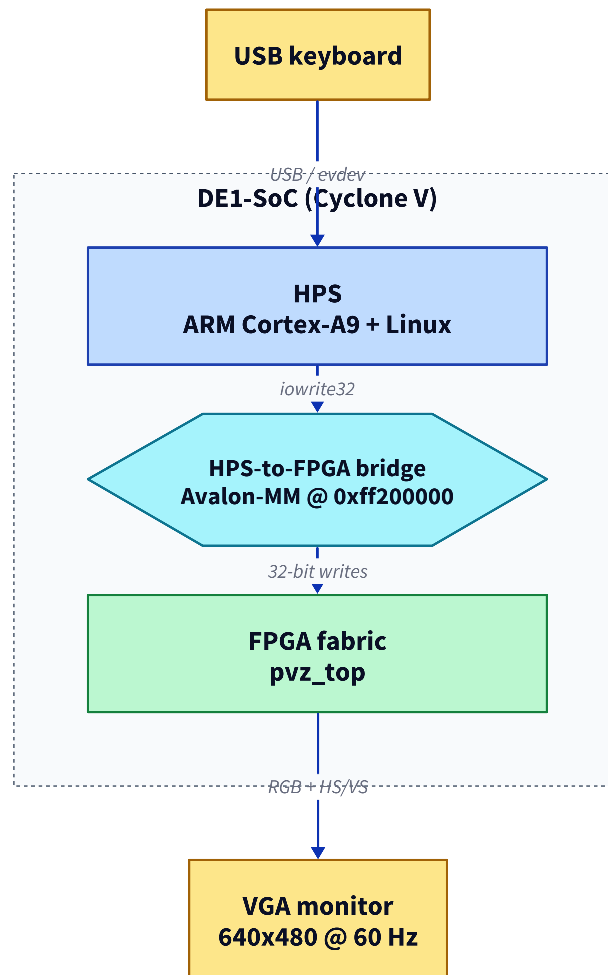


Figure 1.1 — How the HPS half and FPGA half of the DE1-SoC cooperate to produce one frame.

What is in this report

The remaining chapters tour the architecture top-down, then split into an FPGA deep dive and an HPS deep dive, then close with the register map, a frame walkthrough that traces one button press all the way to a pixel, and notes on building and running the design.

2 System Architecture

Before diving into either half of the SoC, it helps to see how the work is split, what flows between the two halves, and why the protocol is as thin as it is. The short version: the HPS owns *what is in the world* and the FPGA owns *what is on the screen*, and they talk through one Avalon-MM register file with no handshake of any kind.

Who owns what

The two halves of the chip have completely disjoint responsibilities:

Half	Owns	Doesn't touch
HPS (Linux + C)	All gameplay state: the 4×8 grid, zombies, peas, sun economy, win/lose. Polls input. Decides what should be on screen.	Has no idea what a pixel is. Generates no video signal.
FPGA (SystemVerilog)	All pixels. Reads the current scene description from its register file and rasterizes it to VGA every frame.	Has no game logic, no collision detection, no notion of time beyond "what cycle is it."

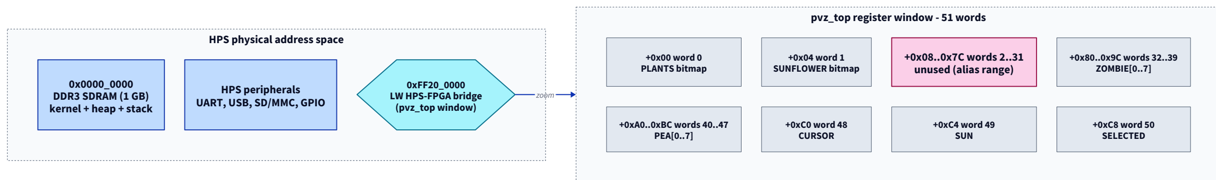
That separation is real. If you grep the HPS code you will not find a pixel constant; if you grep the FPGA you will not find a zombie-speed constant. The HPS describes the scene, the FPGA draws it, and the seam between them is a 51-word register file living inside `pvz_top.sv` (`hw/pvz_top.sv:9-20`, `sw/pvz.h:42-49`).

One register file, one direction of writes

There is no shared memory in this design. No DMA, no interrupts, no vsync handshake, no mailbox. The only path between the two halves is the Avalon-MM slave on `pvz_top`: the HPS writes 32-bit words at word indices 0..50, and the FPGA reads those flopped values combinationally as the VGA beam scans (`hw/pvz_top.sv:30-45`, `hw/pvz_top.sv:95-148`).

The 51 words are densely packed. Two of them are 32-bit bitmaps that describe every plant slot at once — bit `i` of word 0 is "peashooter at cell `i`", bit `i` of word 1 is "sunflower at cell `i`", where `i = row*8 + col` (`hw/pvz_top.sv:9-11`). Sixteen more words describe up to 8 zombies and 8 peas, with alive/row/x packed into one word each. Three words handle the HUD: cursor position and visibility, the sun counter, and the currently selected plant type. Word indices 2..31 are reserved gap space — they exist in the address space but back nothing.

That layout is the entire interface. Chapter 5 walks through every bit, for this chapter it's enough to know that there are 51 32-bit words, and that the HPS rewrites all of them every frame regardless of whether anything changed.



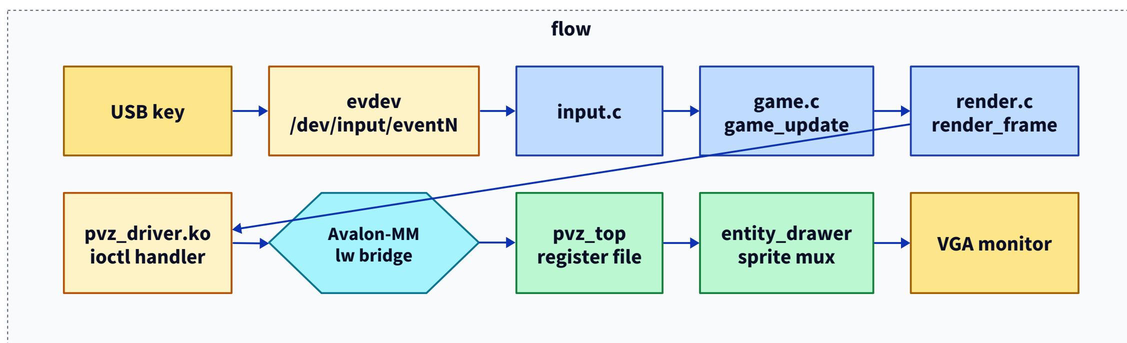
The HPS physical address space, zoomed into the `pvz_top` register window — 51 words at `+0x000..+0x0C8` inside the lightweight HPS-FPGA bridge at `0xff200000`.

The frame at 60 Hz on the HPS

`sw/main.c` runs a fixed-rate loop. On every iteration it grabs the start time, then does four things in this order (`sw/main.c:107-150`):

1. **Poll input.** Read any pending keyboard events from `/dev/input/event0`.
2. **Update game state.** Step every subsystem — sun timer, zombie spawner, peashooter cooldowns, pea movement, zombie eating, collisions, win/lose check.
3. **Render to FPGA.** Walk the `game_state_t` struct and issue ~50 ioctl writes, one per register. The renderer doesn't bother diffing; it just writes everything.
4. **Sleep.** Whatever fraction of the 16.667 ms frame budget is left, `usleep` burns it.

The ordering matters: input-before-update means this frame's keypress takes effect this frame; update-before-render means the FPGA always sees a self-consistent post-step snapshot rather than a half-updated one.



The HPS modules feed `pvz_top` on the left, the FPGA modules drain it on the right — the register file in the middle is the only coupling between the two clock domains.

And meanwhile on the FPGA

On the other side of the chip, the FPGA is running its own loop on a 25 MHz pixel clock derived from the 50 MHz fabric clock (`hw/vga_counters.sv:23-33` declares the 1600×525 timing that drops to 640×480 once you halve `hcount`). Every 40 ns, `hcount` advances; every line, `vcount` advances. For each (x, y) inside the active region, the combinational logic in `entity_drawer.sv` figures out which sprite, if any, covers that pixel, reads a color index from the appropriate sprite ROM, runs it through the palette LUT, and drives the analog R/G/B pins (`hw/pvz_top.sv:201-253`).

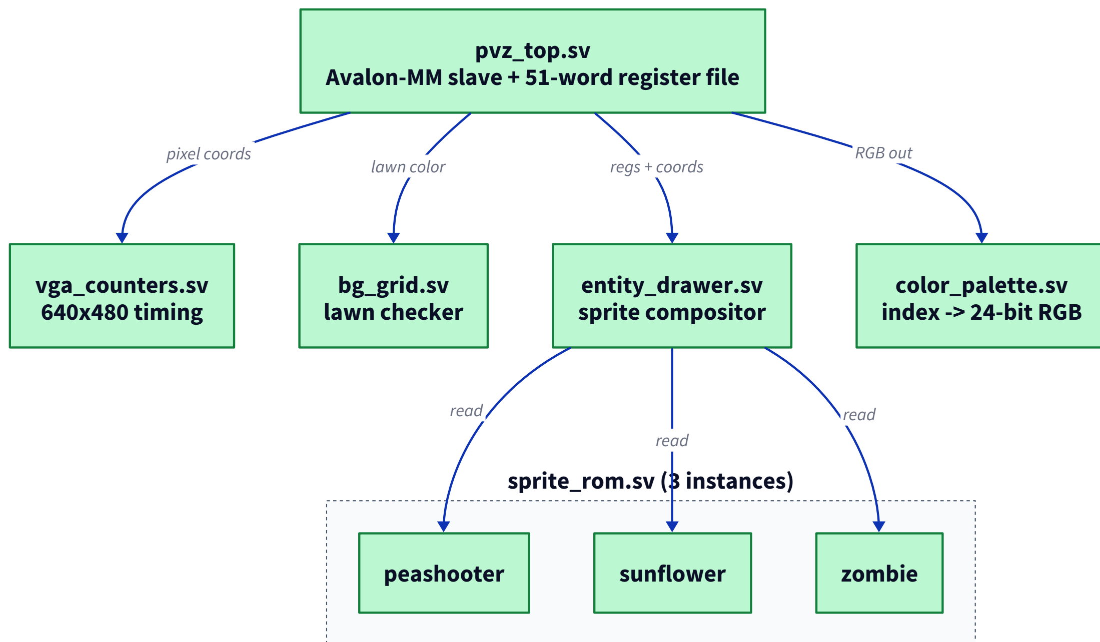
The HPS clock and the FPGA pixel clock never synchronize. The HPS writes whenever it wants; the FPGA reads whenever its beam happens to need that value. That is the whole protocol.

The "racing the beam" choice

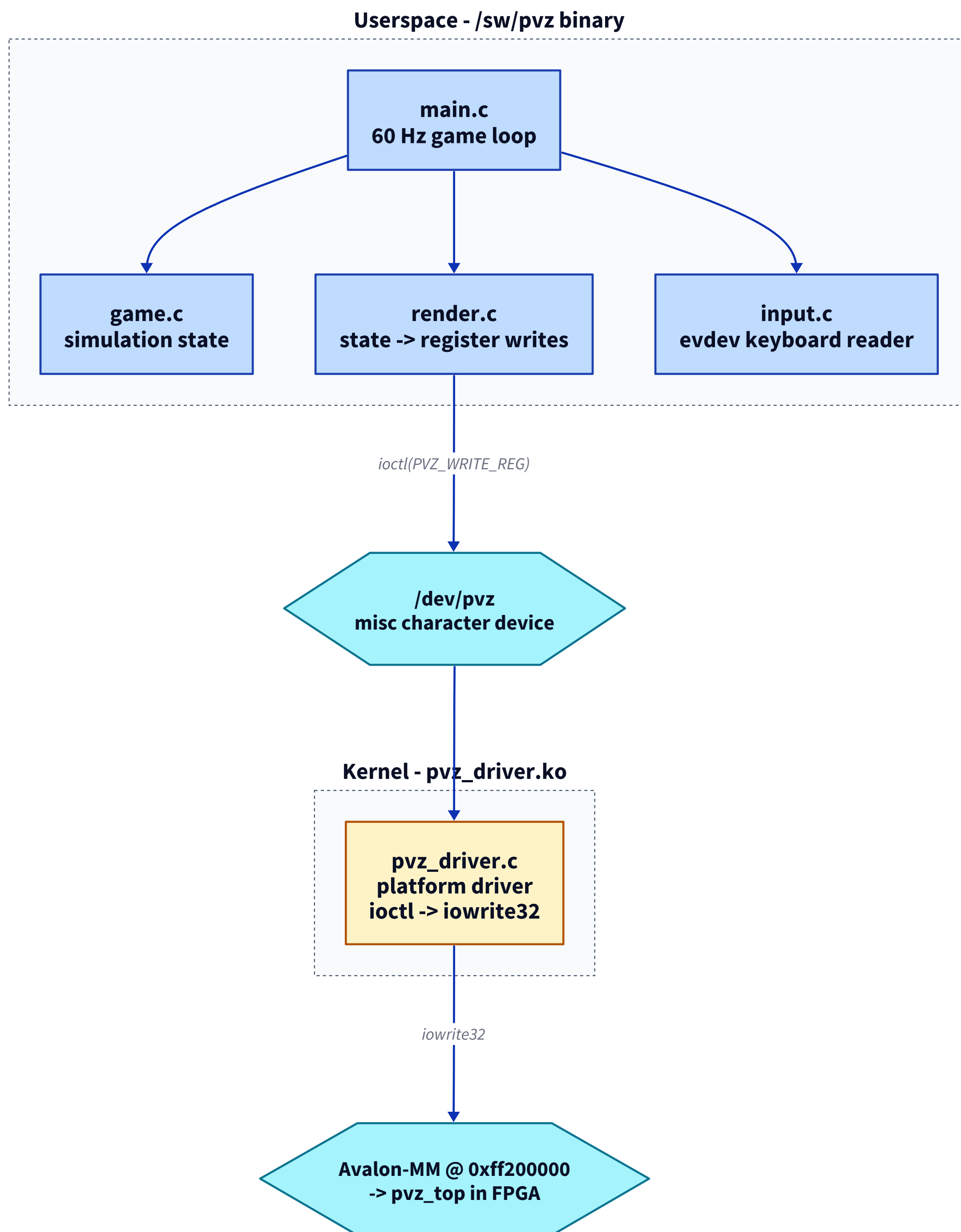
A direct consequence of no synchronization is that a register write that lands mid-scan can split the frame: rows above the current scanline get drawn from the old value, rows below from the new one. The comment block on the register file calls this out explicitly — "there is no vsync latching, so a write that races the scan can produce one frame of tearing. Acceptable for ~60 Hz game state." (`hw/pvz_top.sv:25-27`.)

We took this deal on purpose. At 60 Hz with sprites moving on the order of a pixel or two per frame, a tear is invisible to a human eye. In exchange we get a register file that is just 51 flip-flop banks and a write-decode case statement — no shadow registers, no vsync edge detector, no consistency machinery. If a future revision ever needed clean visuals (cutscenes, fast-moving sprites, palette swaps), the fix is the well-known one from lab 3's `vga_ball.sv`: latch every register into a "shadow" copy at the rising edge of vsync, and read the shadow rather than the live register from the drawer. The tradeoff is one frame of latency and roughly double the register-file storage.

Module hierarchy at a glance



`pvz_top` is the only custom RTL block on the fabric — it instantiates every other module (VGA timing, background, three sprite ROMs, entity drawer, palette LUT) and exposes the Avalon slave to the HPS bridge.

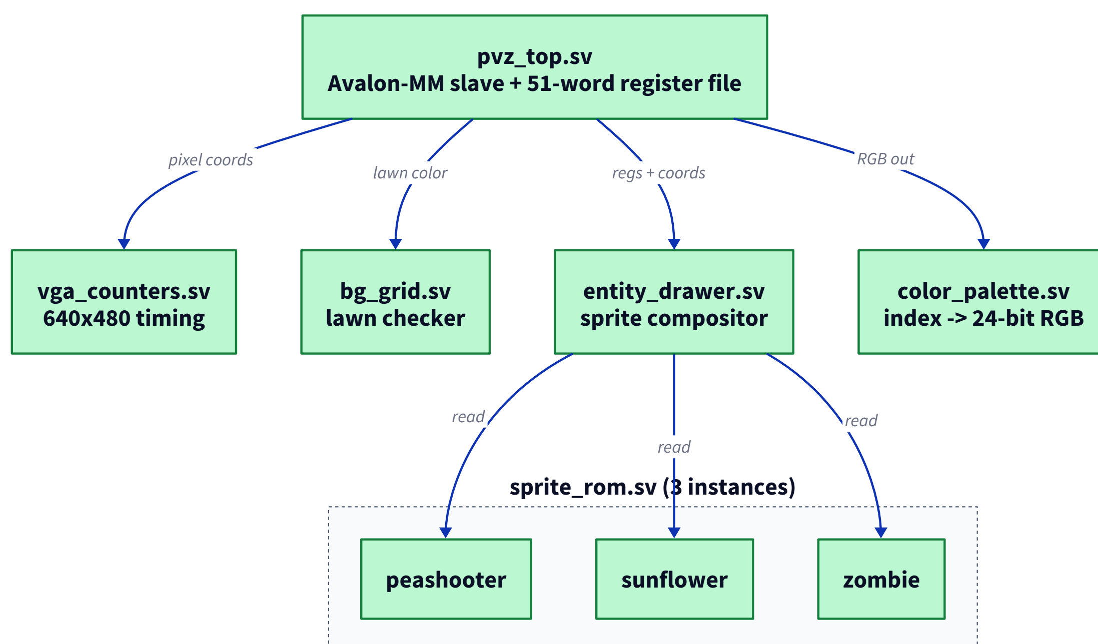


`sw/main.c` orchestrates three userspace modules (`game.c`, `render.c`, `input.c`) that talk to `pvz_driver.ko` through `/dev/pvz`; all five files compile against the same `pvz.h`, so the userspace ioctl struct and the kernel ioctl struct can never drift.

3 Hardware

The FPGA half of PvZ is six SystemVerilog files that turn a register file into a 640x480 VGA picture sixty times a second. There is no frame buffer and no line buffer. Every pixel is computed on the fly from the current register file, in step with the VGA scan. Below is a tour of the pieces and the two-stage pipeline that holds them together.

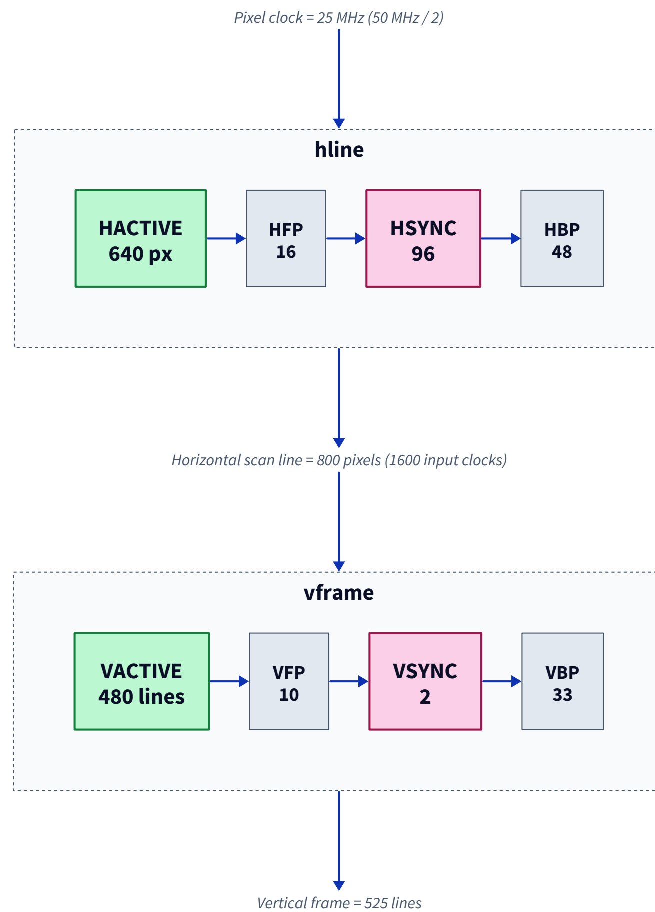
Module hierarchy



Module hierarchy: `pvz_top` instantiates VGA timing, background, three sprite ROMs, the entity drawer, and the palette.

`pvz_top.sv` is the Avalon-MM peripheral that Platform Designer hands to the HPS. It owns the register file (plant bitmaps, zombie slots, pea slots, cursor, sun, selected plant) and instantiates everything else: `vga_counters` for timing, `bg_grid` for the lawn, three `sprite_rom` instances for the peashooter / sunflower / zombie art, `entity_drawer` for the compositor, and `color_palette` for the index-to-RGB conversion (`hw/pvz_top.sv:53-241`). One Avalon write port comes in, one VGA conduit goes out, no interrupts, no readback (`hw/pvz_top.sv:30-45`).

VGA timing



640×480 @ 60 Hz: 800 pixels per line × 525 lines per frame, 25 MHz pixel clock (every other 50 MHz fabric tick).

`vga_counters.sv` is adapted from the lab 3 timing block. It takes the board's 50 MHz clock and produces `hcount[10:0]` (half-pixel ticks within a line, 0..1599), `vcount[9:0]` (line 0..524), and the four VGA control signals (`hw/vga_counters.sv:23-33`). The counters wrap at `HTOTAL-1` and `VTOTAL-1` (`hw/vga_counters.sv:41-50`). The 25 MHz pixel clock comes out almost for free as `hcount[0]` (`hw/vga_counters.sv:66`) — every other 50 MHz tick is a pixel edge. `pvz_top.sv` then takes `hcount[10:1]` as the pixel x and `vcount` as the pixel y (`hw/pvz_top.sv:65-66`), so everything downstream sees a clean 10-bit `(px, py)` that advances once per pixel. The 800-pixel slot decomposition above is what the slide depicts; the `hcount[10:0]` you see in code is the same scan, counted at the doubled 50 MHz fabric clock.

Background

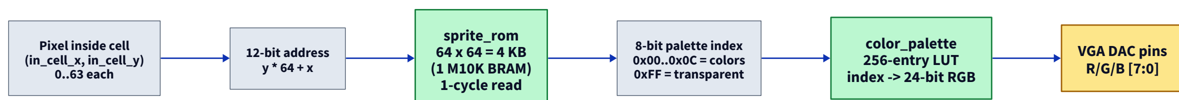
`bg_grid.sv` is the simplest module in the design: pure combinational, no clock, no registers. Given `(px, py)` it returns the 8-bit palette index for the lawn beneath that pixel (`hw/bg_grid.sv:15-19`). The visible play area is hardcoded as an 8x4 grid of 64-pixel cells starting at `(64, 112)` (`hw/bg_grid.sv:28-31`). Outside that rectangle the answer is sky blue; inside, the cells alternate dark and light green by checker parity.

The parity bit is computed by slicing, not arithmetic:

```
wire light_cell = gx[6] ^ gy[6];   // bg_grid.sv:45
```

Because the cell size is 64 = 2^6, bit 6 of the in-grid offset is the LSB of the column index, and bit 6 of the y offset is the LSB of the row index. XORing them is the same as `(col + row) & 1` but takes zero adders (`hw/bg_grid.sv:43-54`). The same `GRID_X`, `GRID_Y`, and cell size constants reappear in `entity_drawer.sv:92-94` and again in software at `sw/pvz.h:30-33`. Keeping the three in sync by hand is fragile but unavoidable given the build flow.

Sprite ROMs



Per-pixel path from `(in_cell_x, in_cell_y)` through a 64×64 sprite ROM (1-cycle synchronous read, one M10K block) and the 256-entry palette LUT, ending at the VGA DAC pins.

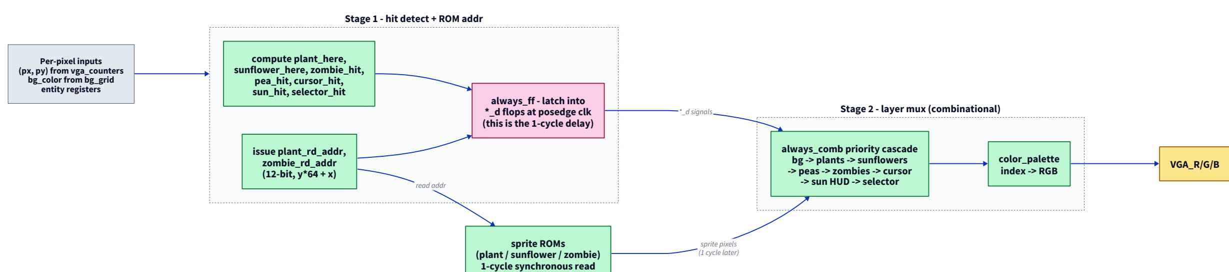
`sprite_rom.sv` is the whole sprite engine on this branch. Twenty-eight lines, a parameterized 4096 x 8-bit synchronous ROM whose contents come from a `.mem` file at synthesis time (`hw/sprite_rom.sv:11-28`):

```
module sprite_rom #(
    parameter MEM_FILE = "peashooter_idx.mem"
) (
    input  logic        clk,
    input  logic [11:0] addr,   // 0..4095 = y*64 + x
    output logic [7:0]  pixel
);

    logic [7:0] rom [0:4095];

    initial begin
        $readmemh(MEM_FILE, rom);
    end

    always_ff @(posedge clk)
        pixel <= rom[addr];

endmodule
```

Two things matter most. `$readmemh` bakes the art into the bitstream — there is no runtime load, so changing a sprite means re-synthesizing. The `always_ff` read makes this a synchronous ROM, which Quartus infers as an M10K block RAM with a registered output: exactly one cycle of read latency. Each byte is either a palette index in `0x00..0x0C` or `0xFF` for transparent (`hw/sprite_rom.sv:5-8`).

`pvz_top.sv` instantiates three of these with different `MEM_FILE` parameters: `peashooter_idx.mem`, `sunflower_idx.mem`, and `zombie_idx.mem` (`hw/pvz_top.sv:179-199`). The peashooter and sunflower ROMs share the same address bus (`plant_addr`), because at most one of `plant_present[i]` or `sunflower_present[i]` is ever set for a given cell, so only one ROM's output wins in the final mux. A fourth `.mem` file, `peas_idx.mem`, is committed but unused — peas are drawn procedurally as small bright-green squares in `entity_drawer.sv`.

The entity drawer

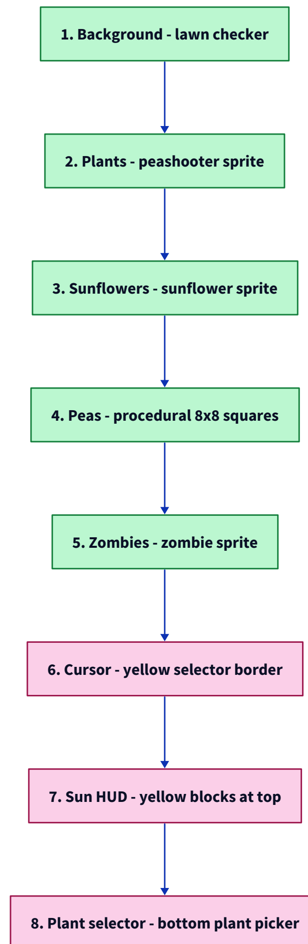


Two-stage drawer pipeline: stage 1 computes hits and ROM addresses; stage 2 composites the registered hits with the ROM outputs that arrived one cycle later.

`entity_drawer.sv` is the heart of the design — 360 lines, no FSM, no buffer, just a pile of combinational hit tests followed by a one-cycle register stage and a final layer mux.

Stage 1 is pure combinational. It answers, in parallel, every "is this pixel part of entity X?" question. The grid math comes first (`hw/entity_drawer.sv:140-166`): subtract `GRID_X` / `GRID_Y` to get an in-grid offset, slice the high bits to get the cell row and column, slice the low six bits to get the in-cell pixel, and pack `(y, x)` as a 12-bit sprite ROM address. The plant and sunflower bitmaps are indexed by `plant_idx = {cell_row, cell_col}` (`hw/entity_drawer.sv:161-163`). Zombie hit detection is a priority encoder over eight slots, picking the first alive zombie whose 64x64 bounding box covers the pixel and emitting the in-zombie offset as `zombie_rd_addr` (`hw/entity_drawer.sv:173-194`). Peas are 8x8 boxes vertically centred in their row, OR-reduced because they are flat color (`hw/entity_drawer.sv:199-211`). The cursor is a hollow yellow rectangle four pixels thick (`hw/entity_drawer.sv:213-230`). The selector HUD is two 48x48 boxes near the top-left with a yellow border on the chosen one (`hw/entity_drawer.sv:240-264`). The sun HUD is up to ten yellow blocks across the top, one per 50 sun (`hw/entity_drawer.sv:268-280`).

Stage 2 runs one cycle later. Every stage-1 boolean is registered into a `_d` version at the next clock edge (`hw/entity_drawer.sv:297-325`). The point is alignment: by stage 2, the sprite ROMs have completed their one-cycle reads, so `plant_rd_pixel`, `sunflower_rd_pixel`, and `zombie_rd_pixel` now carry the right bytes for the pixel whose hit tests were computed last cycle. The final mux paints layers bottom-up — background, plant, sunflower, pea, zombie, cursor, sun HUD, plant selector — with later writes overwriting earlier ones (`hw/entity_drawer.sv:330-358`).



Z-order: background at the bottom, selector border at the top.

The `0xFF` transparent-pixel convention is enforced at every sprite layer:

```
if (plant_here_d && plant_rd_pixel != COL_TRANSPARENT)
    color_out = plant_rd_pixel;          // entity_drawer.sv:332-333
```

If the sprite byte is `0xFF` the layer underneath shows through unchanged — round plants sit cleanly on the checker lawn without any masking logic, because the renderer just doesn't write transparent pixels.

Palette and VGA output

`color_palette.sv` is a 256-entry combinational case statement mapping 8-bit indices to 24-bit RGB (`hw/color_palette.sv:31-49`). Only indices 0..13 are populated — dark and light green for the lawn, brown, yellow for cursor and sun, red and dark red for zombies, the two greens of the

peashooter, bright green for peas, white and gray reserved for HUD digits, orange for the sunflower selector, and sky blue outside the grid. Everything else falls through to black via the `default` arm (`hw/color_palette.sv:47`). Indices 10 and 11 are wired but unused on this branch.

The last step is in `pvz_top.sv`: a small mux that drives palette RGB out during the visible area and forces black during blanking (`hw/pvz_top.sv:243-253`). Without that, the porches would carry whatever the palette happened to compute and the monitor would refuse to lock.

Why no frame buffer

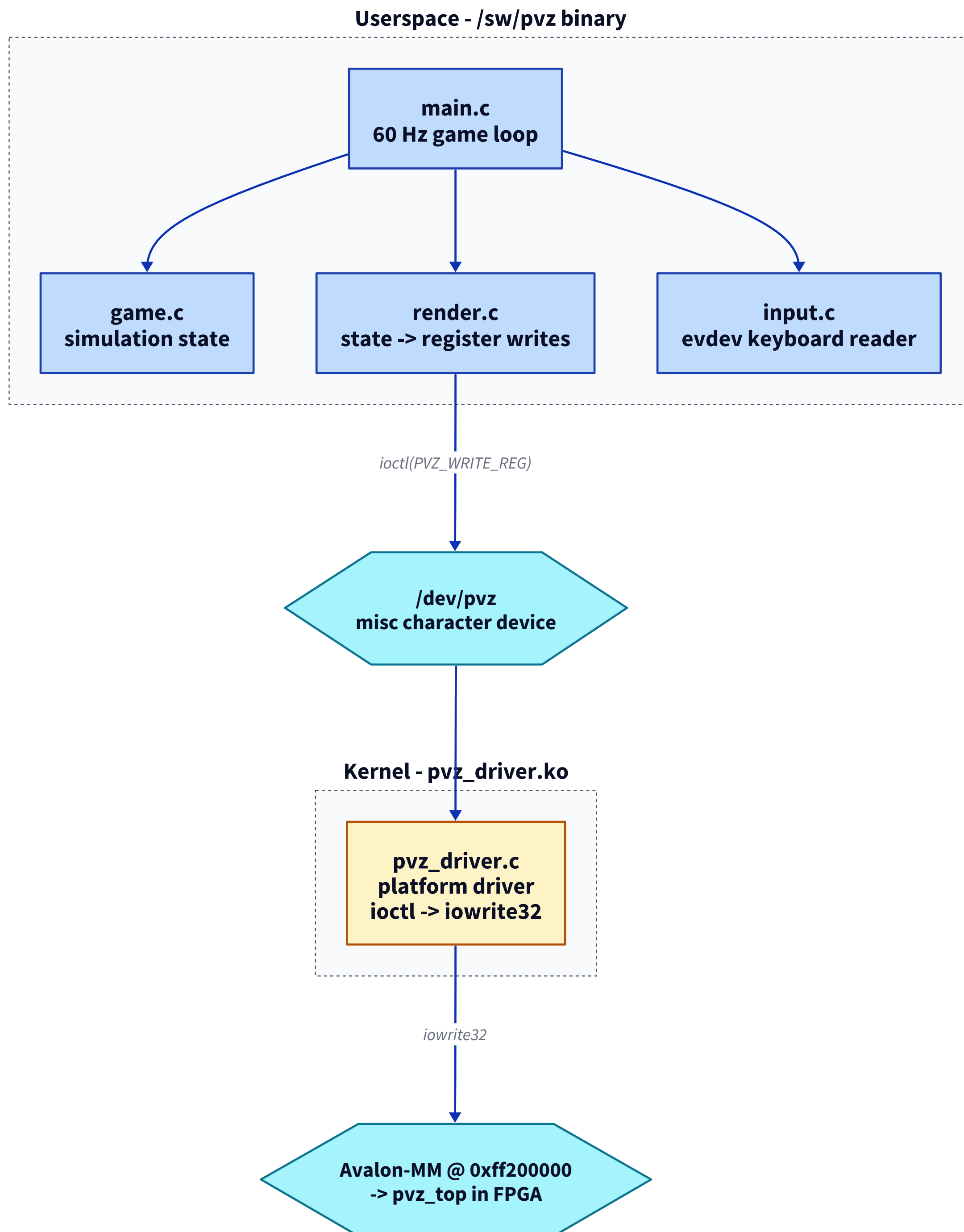
There is no memory of past frames anywhere in the FPGA. The instant software writes a new register value, the next pixel that depends on it reflects the change. The full pipeline from `(px, py)` to RGB is only two clocks deep, which fits comfortably in the 25 MHz pixel budget.

The cost of this simplicity is no vsync latching on this branch. A write that races the scan can produce one frame of tearing — a zombie can shear mid-frame if its `x` register is updated while the beam is on its row (`hw/pvz_top.sv:25-27`). At 60 Hz game state that is rarely visible, but a shadow-register update on `vcount==VACTIVE && hcount==0` (the pattern from lab 3's `vga_ball.sv`) would close the gap. Chapter 2 names that tradeoff.

4 Software

The HPS side is where the game actually plays out. The FPGA draws what it is told to draw; the ARM core decides what that is. Below: the userspace game, the kernel driver, and the single 32-bit ioctl that connects them.

Two binaries, one ABI



Userspace `pvz` on the left, `pvz_driver.ko` on the right, `/dev/pvz` between them; all five files compile against `sw/pvz.h`.

The software ships as two cooperating pieces. `pvz_driver.ko` is a Linux kernel module that owns the FPGA's memory-mapped Avalon slave and exposes it as `/dev/pvz`. `pvz` is the userspace executable that runs the simulation and pushes one frame of register writes per display refresh. Both compile against `sw/pvz.h`, which is the only file included from both sides of the build — it defines the register map (`sw/pvz.h:42-49`), the two packing helpers (`sw/pvz.h:52-65`), the ioctl argument struct, and the `PVZ_WRITE_REG` magic (`sw/pvz.h:68-74`). There is exactly one ioctl, and there is no read path back from hardware. Keeping the ABI in one header means a change to a register's bit layout fails to compile on both sides at once rather than silently desyncing.

The 60 Hz main loop

`sw/main.c` is short by design — most of its 156 lines are setup and teardown around a tight per-frame loop. After parsing an optional input-device path, seeding `rand`, opening `/dev/pvz`, and calling `input_init` / `render_init` / `game_init`, control reaches `sw/main.c:107-150`. Each iteration runs five steps in order: drain queued input events through `process_input`, advance the simulation with `game_update`, push the new scene to the FPGA via `render_frame`, optionally print a status line on `frame_count % 60`, then sleep the rest of the 16.667 ms frame budget. The budget is `FRAME_USEC = 16667` (`sw/main.c:22`) and the sleep call only runs if there is time left, so a slow frame just continues without catching up.

`process_input` (`sw/main.c:34-68`) is the cursor and selection state machine: arrow events nudge `cursor_row` / `cursor_col` with edge clamping, Space calls `game_place_plant`, D calls `game_remove_plant`, Tab toggles `selected_plant_type`, and Esc sets `gs.state = -1`, which is the sentinel the `while (gs.state >= 0)` loop watches for. WIN and LOSE are handled inline: each one prints a banner, rerenders so the cursor disappears, sleeps 5 s, and breaks.

Input via evdev

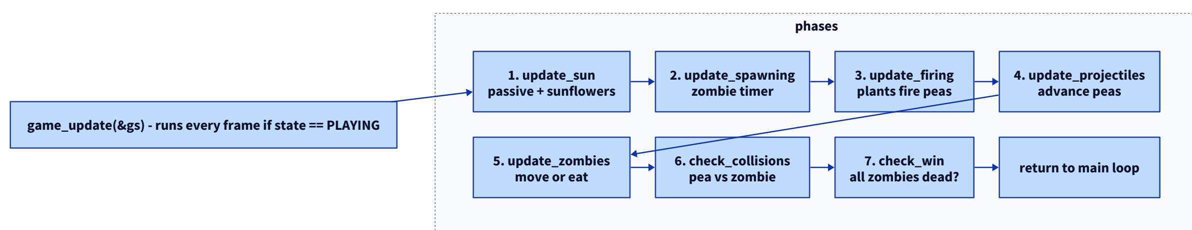
Input lives in `sw/input.c` and is plain Linux evdev — no userspace USB stack. `input_init` opens the device path with `O_RDONLY | O_NONBLOCK` (`sw/input.c:21-29`), and `input_poll` reads `struct input_event` records in a loop until `read` returns `EAGAIN`. Each event is dispatched through a switch that pairs a keyboard keycode with the equivalent Xbox-style gamepad code:

```
case KEY_UP:    case BTN_DPAD_UP:    return INPUT_UP;
case KEY_DOWN:  case BTN_DPAD_DOWN:  return INPUT_DOWN;
case KEY_LEFT:  case BTN_DPAD_LEFT:  return INPUT_LEFT;
case KEY_RIGHT: case BTN_DPAD_RIGHT: return INPUT_RIGHT;
case KEY_SPACE: case BTN_SOUTH:      return INPUT_SPACE;
case KEY_D:     case BTN_EAST:       return INPUT_D;
case KEY_ESC:   case BTN_START:      return INPUT_ESC;
case KEY_TAB:   case BTN_TL: case BTN_TR: return INPUT_TAB;
```

Space on a keyboard and A on a controller hit the same case, so the rest of the codebase only ever sees abstract `INPUT_*` action codes. The function bails after the first action it produces, which is why `process_input` calls it in a `while` loop — a single frame may drain several queued events before the simulation runs. Some xpad variants report the D-pad as an absolute hat axis instead of discrete buttons, so `sw/input.c:42-51` also handles `EV_ABS` events on `ABS_HAT0X` and `ABS_HAT0Y`, ignoring the zero-valued release event so an idle hat doesn't spam the queue.

The simulation: `sw/game.c`

`sw/game.c` is the entire game world, written as a pure state transformation over `game_state_t` (`sw/game.h:77-95`). It owns no file descriptors and never calls into the kernel — that separation makes the simulation trivial to reason about and means a single frame can be replayed deterministically given the same RNG seed.



`game_update` runs seven helpers in a fixed order each frame.

The public entry point is `game_update`, and its body is the entire contract for what a frame means:

```
void game_update(game_state_t *gs)
{
    if (gs->state != STATE_PLAYING)
        return;

    gs->frame_count++;

    update_sun(gs);
    update_spawning(gs);
    update_firing(gs);
    update_projectiles(gs);
    update_zombies(gs);
    check_collisions(gs);
    check_win(gs);
}
```

That is `sw/game.c:288-302`. The order is load-bearing. Spawning runs before zombie movement so a newly placed zombie can take its first step on the same frame; collisions run after both pea and zombie motion so the AABB test sees post-step bounding boxes; the win check runs last so it observes any deactivations that just happened.

A few details worth flagging. Sunflowers compound passive income: on each interval, `update_sun` adds `SUN_INCREMENT * (1 + count_sunflowers(gs))` (`sw/game.c:264-272`), so one sunflower doubles the rate, two triple it, and so on. Peashooters only fire when there is an active zombie in the same row (`zombie_in_row` test inside `update_firing`, `sw/game.c:175-192`), which keeps the projectile pool from saturating in the opening seconds. Collision uses `ZOMBIE_WIDTH = 32` (`sw/game.h:27`) even though the visible sprite is 64 pixels wide; the asymmetry is intentional in `sw/game.c:208-213` so that peas register hits as soon as they reach the zombie's center of mass rather than the back of its head.

Renderer: `sw/render.c`

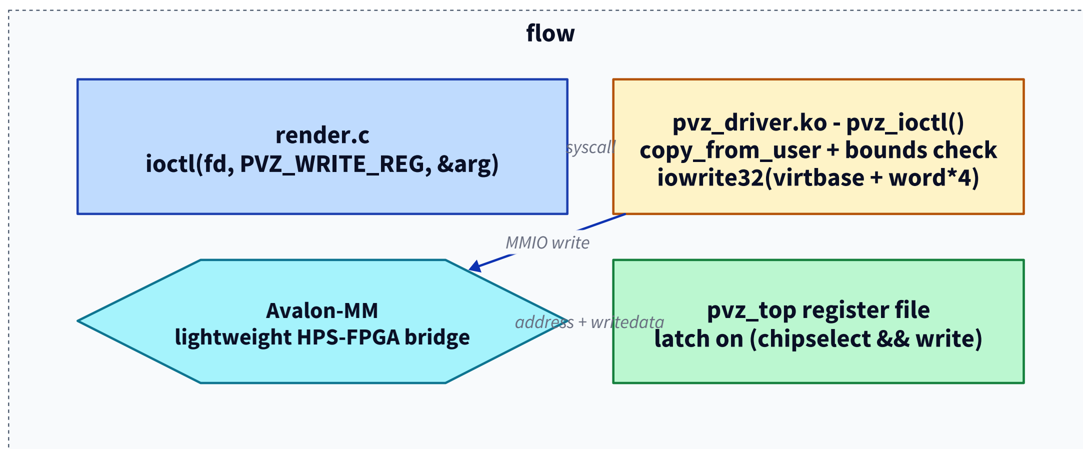
`render_frame` (`sw/render.c:108-124`) is roughly fifty register writes per frame. It dispatches to per-subsystem helpers, each of which calls the tiny `write_reg` wrapper (`sw/render.c:20-24`) that packs a `pvz_write_arg_t` and fires a `PVZ_WRITE_REG` ioctl.

`render_plants` (`sw/render.c:32-47`) walks the 32 grid cells and packs two 32-bit bitmaps in a single pass: one for peashooters, one for sunflowers. The whole plant layer collapses to two register writes. `render_zombies` (`sw/render.c:55-66`) writes all eight hardware zombie slots every frame; inactive slots get zero, which clears the `alive` bit and hides the sprite. Hardware slot `i` always mirrors game slot `i`, so the renderer never compacts.

`render_peas` (`sw/render.c:68-81`) does something slightly different. The simulation has `MAX_PROJECTILES = 16` slots (`sw/game.h:35`) but hardware only renders `PVZ_MAX_PEAS = 8`. The renderer scans the 16-entry array, copies the first eight active projectiles into hardware slots 0–7, and writes zero to any leftover slot. Excess peas are silently dropped. The asymmetry is deliberate: the simulation can keep a fuller projectile pool for collision accuracy even when the screen happens to have more than eight in flight. Cursor, sun, and selected-plant indicator each take one write (`sw/render.c:83-95`).

Kernel driver: `sw/pvz_driver.c`

The kernel side is a 145-line `platform_driver` plus `miscdevice`, following the lab 3 pattern.



`/dev/pvz` ioctl traversal: `copy_from_user` then `iowrite32` at byte offset `word_index * 4`.

`pvz_probe` (`sw/pvz_driver.c:61-99`) runs when the kernel matches a devicetree node whose `compatible` string equals `csee4840,pvz_gpu-1.0`. It calls `misc_register` to create `/dev/pvz`, pulls the physical address out of the devicetree with `of_address_to_resource`, reserves the range via `request_mem_region`, and uses `of_iomap` to build a kernel-virtual mapping in `dev.virtbase`. The compatible string is duplicated in `sw/pvz_driver.c:112` and `hw/pvz_top_hw.tcl:23`; the two must match exactly or the driver will load but never probe.

The ioctl itself is the whole runtime interface:

```
case PVZ_WRITE_REG:
    if (copy_from_user(&w, (pvz_write_arg_t *)arg, sizeof(w)))
        return -EACCES;
    if (w.word_index >= PVZ_NUM_REGS)
        return -EINVAL;
    iowrite32(w.value, dev.virtbase + (w.word_index * 4));
    break;
```

That excerpt is `sw/pvz_driver.c:35-41`. The `* 4` is the gotcha: the Avalon slave was configured with `addressUnits = WORDS`, but `iowrite32` takes a byte offset, so userspace passes a word index and the driver converts it. `copy_from_user` keeps the kernel from trusting a userspace pointer, and the `PVZ_NUM_REGS` bounds check prevents a bad index from scribbling on whatever happens to sit past the slave in the address map.

5 The Register Interface

One channel, one direction

The HPS and the FPGA agree on exactly one thing: a 51-word register file sitting at Avalon byte address `0xff200000`. There is no DMA, no shared frame buffer, no interrupt line, no vsync handshake. The HPS writes to those 51 words once per frame; the FPGA reads them combinationally while the VGA scan walks the screen. That is the protocol in its entirety, and the comment block at the top of `hw/pvz_top.sv:22-27` says so explicitly — writes take effect on the next clock with no commit handshake, and the worst case is one frame of tearing.

This is deliberate. The whole scene at 60 Hz fits in 51 words (32 plant cells packed as bitmaps, 8 zombies, 8 peas, plus a cursor and a couple of HUD scalars). At that size, the cheapest thing the hardware can do is hold all of it in flops and let software repaint everything every frame.

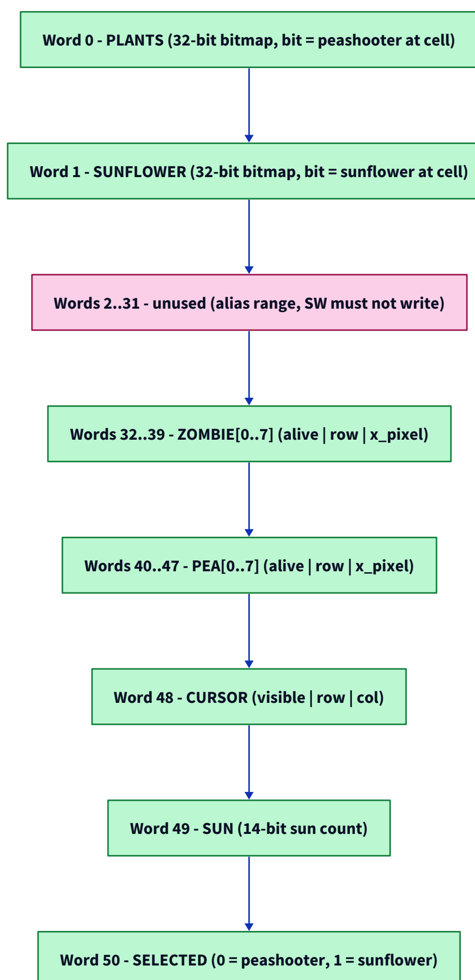


Figure 5.1: The 51 32-bit words of `pvz_top` — two plant bitmaps, an unused alias range, eight zombie slots, eight pea slots, then cursor / sun / selected-plant scalars.

The ABI in two definitions

The userspace-kernel contract lives in five lines of `sw/pvz.h:68-74`:

```
typedef struct {
    unsigned int word_index;    /* 0..PVZ_NUM_REGS-1 */
    unsigned int value;
} pvz_write_arg_t;

#define PVZ_MAGIC 'p'
#define PVZ_WRITE_REG  _IOW(PVZ_MAGIC, 1, pvz_write_arg_t)
```

That is the whole ABI. One ioctl, one struct with a word index and a 32-bit value. There is no read path: software never asks the FPGA "what did you draw last frame?" because nothing the FPGA computes is needed by the game loop — collisions and AI all live in C (`sw/game.c`). The kernel module exposes the same struct at `/dev/pvz` and dispatches a single `case` in `pvz_ioctl` (`sw/pvz_driver.c:30-48`).

A walk through the layout

The 51 words break into five zones. The full bit-by-bit table is in `doc/guide/06-register-map.md` — what follows is the shape, not the field list.

- **Word 0 — peashooter bitmap.** Bit `i = row * 8 + col` is high when a peashooter sits at that cell, packed exactly the way `render_plants` builds it (`sw/render.c:32-46`).
- **Word 1 — sunflower bitmap.** Same encoding. A cell is in at most one bitmap; `game_place_plant` enforces that invariant in software.
- **Words 2–31 — reserved.** The decoder's address ranges (`hw/pvz_top.sv:114-148`) skip over these, so they currently alias into the zombie range if written. Software simply never writes them.
- **Words 32–39 — eight zombie slots.** Each word packs `alive` in bit 31, `row` in `[11:10]`, and `x_pixel` in `[9:0]`. The packing helper is `pvz_pack_entity` (`sw/pvz.h:52-57`).
- **Words 40–47 — eight pea slots.** Identical encoding to zombies; same helper.
- **Word 48 — cursor.** `visible` in bit 31, `col` in `[4:2]`, `row` in `[1:0]`, packed by `pvz_pack_cursor` (`sw/pvz.h:60-65`).
- **Word 49 — sun count.** 14 bits of cumulative sun.
- **Word 50 — selected plant.** Two bits: 0 for peashooter, 1 for sunflower.

The two packing helpers (`pvz_pack_entity` , `pvz_pack_cursor`) exist so the bit positions are stated once in C and referenced from anywhere that needs them. The FPGA-side unpacking lives a few lines away in `hw/pvz_top.sv:123-137` and uses the same slices.

One call site, end to end

Placing a peashooter is the easiest path to trace. When `render_plants` runs, it walks the 4×8 grid and folds the peashooter cells into a single 32-bit bitmap (`sw/render.c:36-44`), then hands it to a thin wrapper:

```
static void write_reg(unsigned int word_index, unsigned int value)
{
    pvz_write_arg_t w = { .word_index = word_index, .value = value };
    ioctl(fd, PVZ_WRITE_REG, &w);
}
```

That wrapper at `sw/render.c:20-24` is the only path software uses to touch hardware. Every `render_*` function in the file ends up here.

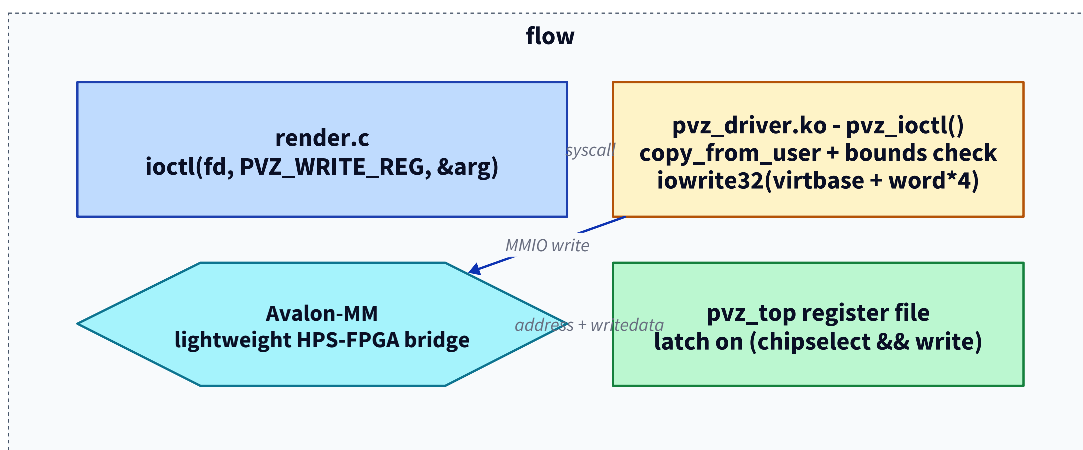


Figure 5.2: One ioctl crossing user → kernel → Avalon → SystemVerilog decoder.

Inside the kernel, `pvz_ioctl` copies the struct, range-checks `word_index` , and converts to a byte offset before the MMIO write:

```
iowrite32(w.value, dev.virtbase + (w.word_index * 4));
```

That single line at `sw/pvz_driver.c:40` is the boundary. The Avalon bus delivers the transaction to `pvz_top.sv` with `address = word_index` (because `addressUnits = WORDS`), and the decoder at `hw/pvz_top.sv:114` matches `address == 6'd0` and latches `writedata` into `plant_present`. On the next pixel where `entity_drawer` is composing the peashooter layer, it sees the new bit and draws the sprite.

No flush, no fence, no acknowledgement. The write completes on the next clock and the FPGA picks it up whenever the scan reaches that cell.

The byte-vs-word thing

The Avalon slave on `pvz_top` is configured with `addressUnits = WORDS` in `hw/pvz_top_hw.tcl:52`, which means the slave's `address` port carries word indices 0..63, not byte offsets. The CPU side does not get to know that — it speaks bytes. So the kernel has to do the conversion itself: `iowrite32(w.value, dev.virtbase + w.word_index * 4)`. If you forget the `* 4`, every write goes to a quarter of the right register (word 0 four times in a row, then word 1 four times, etc.) and the rendering looks correct for the lowest indices and silently wrong for the rest. If you go the other way and configure the slave as `BYTES` while keeping the kernel arithmetic the same, you spread one logical word across four physical writes. Both the C header (`sw/pvz_driver.c:5-7`) and the Verilog header (`hw/pvz_top.sv:22-24`) carry an explicit reminder for this exact reason — it cost real debugging time the first time the slave width was changed.

Why it works without a handshake

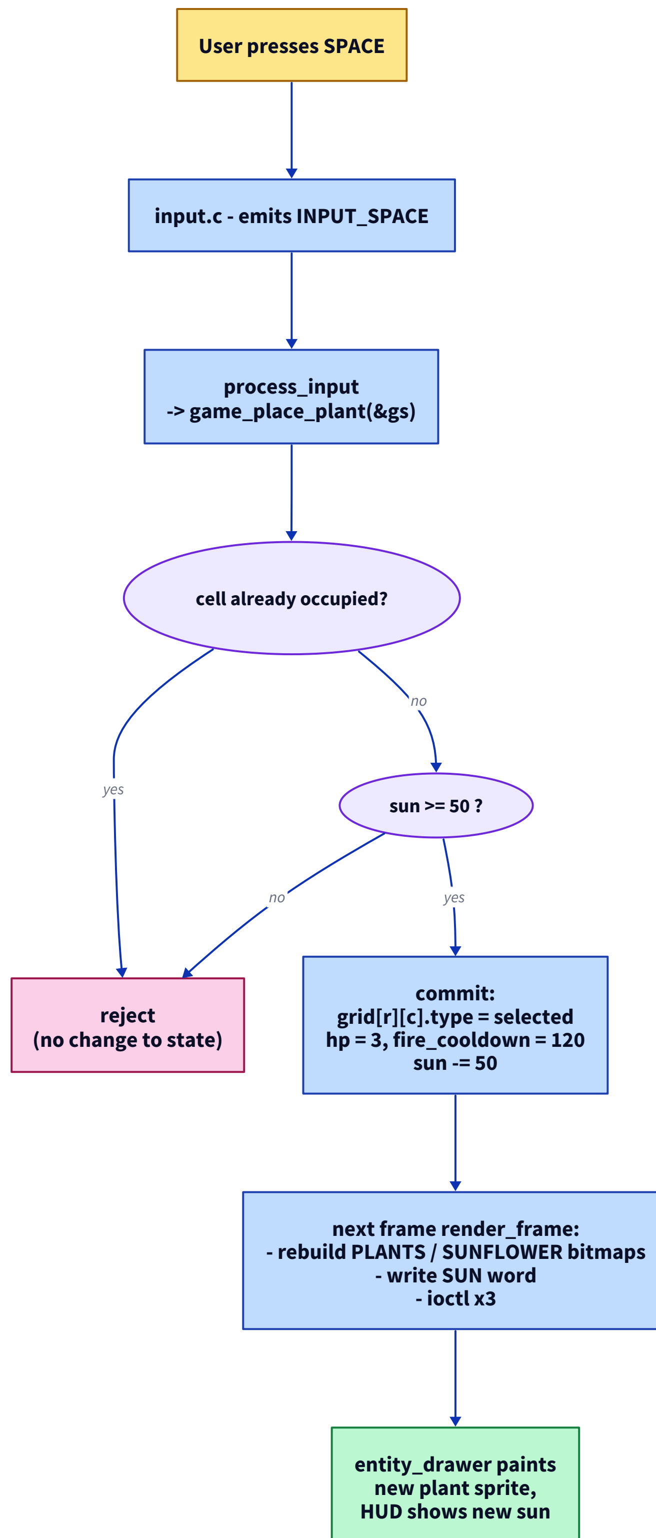
The HPS writes whenever it wants; the FPGA reads whenever it scans. A torn frame — say, half the screen drawn from the old zombie positions and half from the new — is possible if a write lands mid-scan, but at 60 Hz nobody notices, and the game does not depend on hardware acknowledging anything. The next call to `render_frame` (`sw/render.c:108-124`) will overwrite everything anyway. That is the contract: software owns the truth, hardware owns the pixels, and 51 words are enough wire between them.

6 One Frame, End to End

Previous chapters discussed the components in isolation. This one wires them together by following a single action — pressing A to place a peashooter at row 1, column 3 — from gamepad cable to glowing pixel. The five-decade-old VGA scan and the brand-new ioctl meet in the middle.

The scenario

The cursor is parked at row 1, column 3. The sun counter sits at 70, comfortably above the 50-sun peashooter cost (`sw/game.h:25` for `PLANT_COST`). The selected plant type is `PLANT_PEASHOOTER`. The player thumbs the A button on an Xbox 360 controller.



End-to-end trace of one button press, from gamepad to monitor.

1. USB to evdev to `input_poll`

The `xpad` kernel driver — already loaded by the Linux image we boot — sees the controller's HID report and emits a `struct input_event` with `type = EV_KEY`, `code = BTN_SOUTH`, `value = 1` to `/dev/input/event0`. On the next iteration of our game loop, `input_poll` (`sw/input.c:31-71`) does a non-blocking read, hits the `BTN_SOUTH` case at `sw/input.c:62`, and returns `INPUT_SPACE`. The gamepad's A button and the keyboard's space bar funnel into the same action code by design, so the rest of the stack is input-device-agnostic.

2. `process_input` mutates the grid

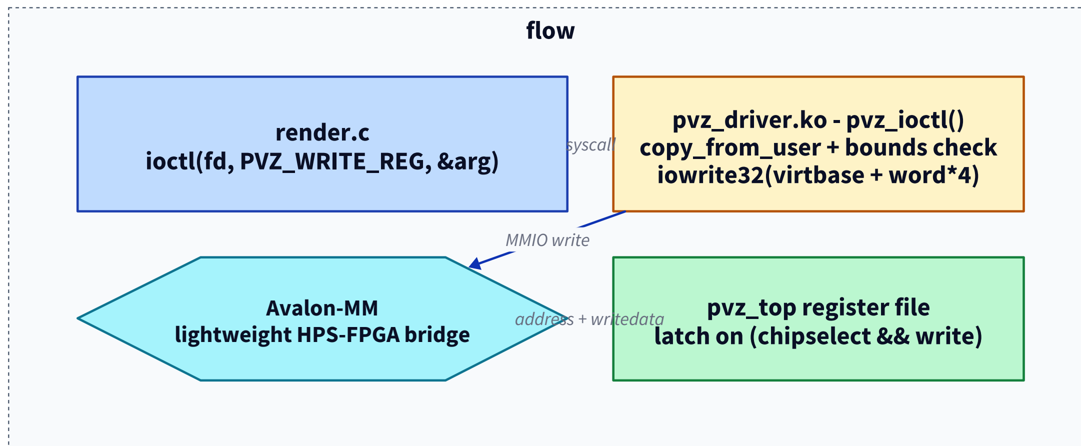
Back in `process_input` (`sw/main.c:34-68`), the `INPUT_SPACE` case calls `game_place_plant`. That function reads `gs->cursor_row = 1`, `gs->cursor_col = 3`, checks that `grid[1][3].type == PLANT_NONE` and that `sun >= 50`, and then writes `type = PLANT_PEASHOOTER`, `fire_cooldown = 120`, and `hp = 3` into the cell while subtracting 50 from `sun` (`sw/game.c:34-51`). No hardware has been touched yet; this is purely a struct edit on the HPS.

3. `game_update` ticks the simulation

`sw/main.c:119` then calls `game_update`, which runs the seven per-frame helpers in order (`sw/game.c:288-302`). The relevant one for our new plant is `update_firing` (`sw/game.c:175-192`): it walks the 4x8 grid, finds the freshly placed peashooter, and decrements its `fire_cooldown` from 120 to 119. The cooldown is still positive, so no pea spawns on this frame. The plant exists in `gs` but is still invisible.

4. `render_frame`, `write_reg`, `ioctl`, `iowrite32`

`sw/main.c:122` calls `render_frame`, which dispatches to `render_plants` (`sw/render.c:32-47`). That helper walks all 32 cells, accumulates a 32-bit bitmap, and for our cell sets bit `1 * 8 + 3 = 11` in `pea_bits`. It then calls `write_reg(PVZ_REG_PLANTS, pea_bits)`, which packs a `pvz_write_arg_t` and issues `ioctl(fd, PVZ_WRITE_REG, &w)`. The call traps into `pvz_ioctl` (`sw/pvz_driver.c:30-48`), which `copy_from_user`s the argument and executes `iowrite32(w.value, dev.virtbase + w.word_index * 4)` — a single store on the lightweight HPS-to-FPGA bridge addressed at byte 0 of the peripheral.



The kernel ioctl path: one userspace call, one `iowrite32`, one Avalon write.

The Avalon bus, configured with word address units (`hw/pvz_top.sv:22-23`), presents `address = 6'd0`, `writedata = pea_bits`, `write = 1`, `chipselect = 1` to `pvz_top.sv`. On the next 50 MHz clock edge the case at `hw/pvz_top.sv:114-116` fires and `plant_present <= writedata`, which is wired straight into `entity_drawer` (`hw/pvz_top.sv:211`) with no shadow register in between.

5. The VGA beam catches up

The beam was scanning the whole time, ignorant of any of the above. Eventually it lands inside cell (1, 3) — pixels where `px ∈ [256, 320)` and `py ∈ [176, 240)`. Stage 1 of `entity_drawer` (`hw/entity_drawer.sv:140-166`) computes `cell_col = gx[8:6] = 3`, `cell_row = gy[7:6] = 1`, so `plant_idx = {cell_row, cell_col} = 5'd11`. `plant_present[11]` is now 1, so `plant_here` goes high. The address `{in_cell_y, in_cell_x}` goes to the peashooter sprite ROM, which returns its pixel one clock later. Stage 2 (`hw/entity_drawer.sv:282-358`) registers `plant_here` into `plant_here_d` and the mux at line 332 fires: `if (plant_here_d && plant_rd_pixel != COL_TRANSPARENT) color_out = plant_rd_pixel`. The 8-bit color index threads through `color_palette` (`hw/pvz_top.sv:236-241`) into the VGA DAC pins (`hw/pvz_top.sv:243-253`), and the monitor lights up a green peashooter at (1, 3).

What was actually parallel

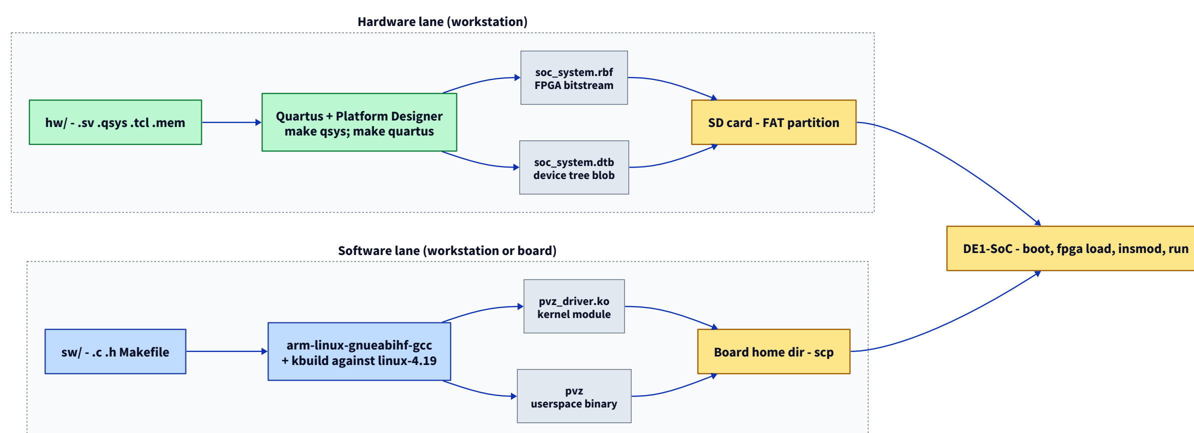
It is easy to read this list and picture a serial pipeline, but most of it ran in parallel. Steps 1 through 4 took the HPS roughly 100 μs once per frame; the rest of its 16.667 ms budget was spent in `usleep` (`sw/main.c:147-149`). Step 5 ran continuously at the 25 MHz pixel rate — every two 50 MHz clocks the drawer emitted a fresh pixel, whether the register file had just changed or not. The HPS never waits for the FPGA, the FPGA never waits for the HPS, and the only synchronization is "the next clock edge happens." That decoupling is what lets a 16 ms game tick share a wire with a 40 ns pixel clock.

7 Build and Run

The earlier chapters described the design as it sits on the branch. This one is the operator's manual: how to turn the source tree into a bitstream, a kernel module, and a running game, and which of the things you may have read about elsewhere are not actually here.

Hardware build flow

The FPGA half is built on the Columbia EE workstations (`micro1.ee.columbia.edu` through `micro10.ee.columbia.edu`). They have Quartus Prime, Platform Designer, the ARM cross-compile toolchain, and `embedded_command_shell.sh` pre-installed (`doc/guide/08-build-and-run.md:9-16`). Source that shell script first; otherwise `quartus_sh`, `qsys-generate`, `sopc2dts`, and `dtc` are not on your `PATH`.



Two build lanes feeding the board: hardware lane (Quartus → RBF + DTB → SD card) and software lane (`gcc` + kbuild → `pvz_driver.ko` + `pvz` → home dir).

All commands run from `hw/`:

```
cd hw/
make project   # regenerate Quartus project files (.qpf, .qsf, .sdc)
make qsys      # regenerate Verilog from soc_system.qsys + pvz_top_hw.tcl
make quartus   # full synth/place/route, 30-60 min, ->
output_files/soc_system.sof
make rbf       # convert .sof -> soc_system.rbf for SD-card boot
make dtb       # sopc2dts + dtc -> soc_system.dtb
```

Re-run from the step matching your change: edit `pvz_top_hw.tcl` and you need `qsys` onward; edit `entity_drawer.sv` and you can skip straight to `quartus`. The full sequence is laid out in `doc/guide/08-build-and-run.md:28-51`. Two artifacts come out the other side: `hw/output_files/soc_system.rbf` and `hw/soc_system.dtb`. The first is the FPGA configuration; the second tells the kernel where to find our peripheral.

If `make dtb` fails with a missing-command error, you forgot to source `embedded_command_shell.sh` (`doc/guide/08-build-and-run.md:59-60`). If `make quartus` finishes but the design behaves wrong, open `hw/quartus_build.log` and look for negative-slack warnings — the design closes timing at 50 MHz and the stage-1 combinational path through `entity_drawer.sv` is the usual suspect.

Software build flow

The simplest case is building natively on the board. Once Linux is up and the kernel headers are in place, edit `.c` files on the board and rebuild in place:

```
cd /home/root/pvz/sw
make
```

That single `make` invokes both `make module` (kbuild against `/usr/src/linux-headers-$(uname -r)`, producing `pvz_driver.ko`) and `make pvz` (a plain `gcc -Wall -O2 -o pvz main.c game.c render.c input.c -lpthread`). See `doc/guide/08-build-and-run.md:75-86` for the exact rules.

To build on a workstation and `scp` the result over, override the toolchain variables (`doc/guide/08-build-and-run.md:94-100`):

```
cd sw/
make CC=arm-linux-gnueabihf-gcc \
     ARCH=arm \
     CROSS_COMPILE=arm-linux-gnueabihf- \
     KERNEL_SOURCE=/path/to/linux-headers-4.19.0-arm
```

The outputs are the same in either flow: `sw/pvz_driver.ko` and `sw/pvz`.

Getting it onto the board

The board boots from an SD card. The FAT partition holds the bitstream and the devicetree; U-Boot loads them before handing off to Linux. Mount the card on your workstation and copy:

```
cp hw/output_files/soc_system.rbf /media/$USER/de1soc_boot/
cp hw/soc_system.dtb               /media/$USER/de1soc_boot/
```

Then unmount, re-seat the card in the board, and power-cycle (`doc/guide/08-build-and-run.md:118-124`). The `.ko` and the `pvz` binary live on the rootfs partition and can travel over the network instead — `scp sw/pvz_driver.ko sw/pvz root@<board-ip>:/home/root/pvz/`.

Once the board is up, connect to its serial console (`screen /dev/ttyUSB0 115200`), log in as `root`, and run:

```
cd /home/root/pvz
insmod pvz_driver.ko
dmesg | tail
#   expected: pvz: initialized at 0xff200000
cat /proc/iomem | grep ff20
ls /dev/input/
cat /dev/input/event0   # press a button; bytes appear if this is the gamepad
./pvz /dev/input/event0
```

The full sequence with expected output is in `doc/guide/08-build-and-run.md:149-170`. If `dmesg` does not show the `initialized at 0xff200000` line, the devicetree did not match the driver's compatible string — usually because you forgot to copy a freshly-built `soc_system.dtb` to the SD card.

Controls

The keyboard and Xbox 360 mappings are identical in behavior; pick whichever input device shows up under `/dev/input/`. Source: `doc/guide/08-build-and-run.md:175-181`.

Action	Keyboard	Xbox 360
Move cursor	Arrow keys	D-pad
Place plant	Space	A
Remove plant	D	B
Toggle peashooter / sunflower	Tab	LB or RB
Quit	Esc	Start

8 Appendix: Source Code

This section contains only the code we wrote. Excludes lab3-derived files (`vga_counters.sv`, the HPS/pin boilerplate in `soc_system_top.sv`), tool-generated Platform Designer output, sprite `.mem` data, and Makefiles.

Software (HPS, C)

`sw/pvz.h`

```
#ifndef _PVZ_H
#define _PVZ_H

#include <linux/ioctl.h>

/*
 * Shared header for PvZ GPU kernel driver and userspace programs.
 *
 * The hardware exposes a flat 32-bit register file via Avalon-MM.
 * Software writes one word at a time using the PVZ_WRITE_REG ioctl.
 *
 * Register map (word index -> meaning):
 *    0       PLANTS              32 bits, bit i = peashooter at cell i
 *    1       SUNFLOWERS          32 bits, bit i = sunflower at cell i
 *   32..39   ZOMBIE[i]           bit 31 = alive
 *                                bits [9:0]   = x_pixel (0..639)
 *                                bits [11:10] = row (0..3)
 *   40..47   PEA[i]              same encoding as ZOMBIE
 *   48       CURSOR              bit 31 = visible
 *                                bits [4:2] = col (0..7)
 *                                bits [1:0] = row (0..3)
 *   49       SUN                 bits [13:0] = sun count
 *   50       SELECTED            bits [1:0]  = selected plant (0=pea,
1=sunflower)
 *
 * Layout constants must match hw/bg_grid.sv and hw/entity_drawer.sv.
 */

/* Grid + screen layout */
#define PVZ_GRID_ROWS    4
#define PVZ_GRID_COLS    8
#define PVZ_CELL_SIZE    64
#define PVZ_GRID_X       64
#define PVZ_GRID_Y       112
#define PVZ_SCREEN_W     640
#define PVZ_SCREEN_H     480

/* Per-entity capacity exposed by hardware */
#define PVZ_MAX_ZOMBIES  8
#define PVZ_MAX_PEAS     8

/* Word indices in the register file */
#define PVZ_REG_PLANTS           0                      /* peashooter bitmap */
```

```
#define PVZ_REG_SUNFLOWER        1                    /* sunflower bitmap */
#define PVZ_REG_ZOMBIE(idx)      (32 + (idx))         /* 32..39 */
#define PVZ_REG_PEA(idx)         (40 + (idx))         /* 40..47 */
#define PVZ_REG_CURSOR           48
#define PVZ_REG_SUN              49
#define PVZ_REG_SELECTED         50
#define PVZ_NUM_REGS             51

/* Pack a zombie/pea word: alive in bit 31, row in [11:10], x in [9:0] */
static inline unsigned int pvz_pack_entity(int alive, int row, int x_pixel)
{
    return ((unsigned)(alive & 1) << 31) |
           ((unsigned)(row   & 3) << 10) |
           ((unsigned)(x_pixel & 0x3FF));
}

/* Pack a cursor word: visible in bit 31, col in [4:2], row in [1:0] */
static inline unsigned int pvz_pack_cursor(int visible, int row, int col)
{
    return ((unsigned)(visible & 1) << 31) |
           ((unsigned)(col     & 7) << 2)  |
           ((unsigned)(row     & 3));
}

/* ioctl argument: write `value` to register `word_index` */
typedef struct {
    unsigned int word_index;   /* 0..PVZ_NUM_REGS-1 */
    unsigned int value;
} pvz_write_arg_t;

#define PVZ_MAGIC 'p'
#define PVZ_WRITE_REG  _IOW(PVZ_MAGIC, 1, pvz_write_arg_t)

#endif /* _PVZ_H */
```

`sw/game.h`

```
#ifndef _GAME_H
#define _GAME_H

#include "pvz.h"

/* Grid dimensions (must match hw/entity_drawer.sv) */
#define GRID_ROWS      PVZ_GRID_ROWS
#define GRID_COLS      PVZ_GRID_COLS
#define CELL_SIZE      PVZ_CELL_SIZE
#define GAME_AREA_X    PVZ_GRID_X
#define GAME_AREA_Y    PVZ_GRID_Y

/* Screen dimensions */
#define SCREEN_W       PVZ_SCREEN_W
#define SCREEN_H       PVZ_SCREEN_H

/* Plant constants */
#define PLANT_COST           50
#define SUNFLOWER_COST       50
#define PLANT_FIRE_COOLDOWN 120  /* frames (2 seconds at 60fps) */
#define PLANT_HP             3    /* hits before a plant is destroyed */

/* Zombie constants (sprite size matches hardware: 32x64) */
#define MAX_ZOMBIES           PVZ_MAX_ZOMBIES
#define ZOMBIE_HP             3
#define ZOMBIE_SPEED_FRAMES  3    /* move 1 pixel every N frames (~20 px/s) */
#define ZOMBIE_WIDTH          32
#define ZOMBIE_HEIGHT         64
#define TOTAL_ZOMBIES         5
#define ZOMBIE_SPAWN_MIN     (8 * 60)  /* 8 seconds in frames */
#define ZOMBIE_SPAWN_MAX     (15 * 60) /* 15 seconds in frames */
#define ZOMBIE_EAT_COOLDOWN 60       /* frames between bites */

/* Projectile constants */
#define MAX_PROJECTILES      16
#define PEA_SPEED            2   /* pixels per frame */
#define PEA_DAMAGE           1
#define PEA_SIZE             8

/* Sun economy */
#define INITIAL_SUN          100
#define SUN_INCREMENT        25
#define SUN_INTERVAL         (8 * 60)  /* 8 seconds in frames */

/* Game states */
#define STATE_PLAYING   0
```

```
#define STATE_WIN      1
#define STATE_LOSE     2

/* Plant types */
#define PLANT_NONE        0
#define PLANT_PEASHOOTER  1
#define PLANT_SUNFLOWER   2

typedef struct {
    int type;           /* PLANT_NONE or PLANT_PEASHOOTER */
    int fire_cooldown;  /* frames until next shot */
    int hp;             /* hit points remaining */
} plant_t;

typedef struct {
    int active;
    int row;
    int x_pixel;        /* screen pixel x; moves leftward */
    int hp;
    int move_counter;   /* counts frames until next pixel move */
    int eating;         /* 1 if currently eating a plant */
    int eat_timer;      /* frames until next bite */
} zombie_t;

typedef struct {
    int active;
    int row;
    int x_pixel;        /* screen pixel x; moves rightward */
} projectile_t;

typedef struct {
    plant_t      grid[GRID_ROWS][GRID_COLS];
    zombie_t     zombies[MAX_ZOMBIES];
    projectile_t projectiles[MAX_PROJECTILES];

    int cursor_row;
    int cursor_col;

    int selected_plant_type;  /* PLANT_PEASHOOTER or PLANT_SUNFLOWER */

    int sun;
    int sun_timer;

    int zombies_spawned;
    int spawn_timer;

    int state;          /* STATE_PLAYING, STATE_WIN, STATE_LOSE */
    int frame_count;
} game_state_t;
```

```
void game_init(game_state_t *gs);
void game_update(game_state_t *gs);
int  game_place_plant(game_state_t *gs);
int  game_remove_plant(game_state_t *gs);

#endif /* _GAME_H */
```

`sw/game.c`

```c
/*
 * PvZ game logic
 *
 * Manages the 4x8 grid, zombie spawning/movement, peashooter firing,
 * projectile movement, collision detection, sun economy, and
 * win/lose conditions.
 */

#include <stdlib.h>
#include <string.h>
#include "game.h"

/* Simple pseudo-random using the C library rand() */
static int random_range(int min, int max)
{
    return min + (rand() % (max - min + 1));
}

void game_init(game_state_t *gs)
{
    memset(gs, 0, sizeof(*gs));

    gs->sun = INITIAL_SUN;
    gs->sun_timer = SUN_INTERVAL;
    gs->state = STATE_PLAYING;
    gs->cursor_row = 0;
    gs->cursor_col = 0;
    gs->selected_plant_type = PLANT_PEASHOOTER;
    gs->zombies_spawned = 0;
    gs->spawn_timer = random_range(ZOMBIE_SPAWN_MIN, ZOMBIE_SPAWN_MAX);
    gs->frame_count = 0;
}

int game_place_plant(game_state_t *gs)
{
    int r = gs->cursor_row;
    int c = gs->cursor_col;
    int type = gs->selected_plant_type;
    int cost = (type == PLANT_SUNFLOWER) ? SUNFLOWER_COST : PLANT_COST;

    if (gs->grid[r][c].type != PLANT_NONE)
        return 0;
    if (gs->sun < cost)
        return 0;

    gs->grid[r][c].type = type;
```

```
    gs->grid[r][c].fire_cooldown = PLANT_FIRE_COOLDOWN;
    gs->grid[r][c].hp = PLANT_HP;
    gs->sun -= cost;
    return 1;
}

int game_remove_plant(game_state_t *gs)
{
    int r = gs->cursor_row;
    int c = gs->cursor_col;

    if (gs->grid[r][c].type == PLANT_NONE)
        return 0;

    gs->grid[r][c].type = PLANT_NONE;
    gs->grid[r][c].fire_cooldown = 0;
    return 1;
}

/* Check if any active zombie is in the given row */
static int zombie_in_row(game_state_t *gs, int row)
{
    for (int i = 0; i < MAX_ZOMBIES; i++) {
        if (gs->zombies[i].active && gs->zombies[i].row == row)
            return 1;
    }
    return 0;
}

/* Convert a zombie's screen x to a grid column.
 * Returns -1 if the zombie is not over the lawn. */
static int zombie_col(int x_pixel)
{
    int gx = x_pixel - GAME_AREA_X;
    if (gx < 0 || gx >= GRID_COLS * CELL_SIZE)
        return -1;
    return gx / CELL_SIZE;
}

/* Spawn a new pea projectile at the given grid cell */
static void spawn_pea(game_state_t *gs, int row, int col)
{
    for (int i = 0; i < MAX_PROJECTILES; i++) {
        if (!gs->projectiles[i].active) {
            gs->projectiles[i].active = 1;
            gs->projectiles[i].row = row;
            /* Start at the right edge of the plant's cell */
            gs->projectiles[i].x_pixel =
                GAME_AREA_X + (col + 1) * CELL_SIZE;
```

```
                return;
            }
        }
        /* No free slot; pea is lost */
    }

    /* Update zombie positions, eating, and check for lose condition */
    static void update_zombies(game_state_t *gs)
    {
        for (int i = 0; i < MAX_ZOMBIES; i++) {
            zombie_t *z = &gs->zombies[i];
            if (!z->active)
                continue;

            if (z->eating) {
                /* Re-check that the plant still exists (another zombie may
                 * have destroyed it) */
                int col = zombie_col(z->x_pixel);
                if (col < 0 || gs->grid[z->row][col].type == PLANT_NONE) {
                    z->eating = 0;
                    z->eat_timer = 0;
                    /* Fall through to movement below */
                } else {
                    /* Continue eating: deal damage on timer */
                    z->eat_timer--;
                    if (z->eat_timer <= 0) {
                        gs->grid[z->row][col].hp--;
                        if (gs->grid[z->row][col].hp <= 0) {
                            gs->grid[z->row][col].type = PLANT_NONE;
                            gs->grid[z->row][col].fire_cooldown = 0;
                            gs->grid[z->row][col].hp = 0;
                            z->eating = 0;
                        } else {
                            z->eat_timer = ZOMBIE_EAT_COOLDOWN;
                        }
                    }
                    continue; /* Don't move while eating */
                }
            }

            /* Movement */
            z->move_counter++;
            if (z->move_counter >= ZOMBIE_SPEED_FRAMES) {
                z->move_counter = 0;
                z->x_pixel--;

                /* Lose condition: zombie reached the lawn's left edge */
                if (z->x_pixel <= GAME_AREA_X) {
                    gs->state = STATE_LOSE;
```

```
                return;
            }

            /* Check for plant collision after moving */
            int col = zombie_col(z->x_pixel);
            if (col >= 0 && gs->grid[z->row][col].type != PLANT_NONE) {
                z->eating = 1;
                z->eat_timer = ZOMBIE_EAT_COOLDOWN;
            }
        }
    }
}

/* Update projectile positions */
static void update_projectiles(game_state_t *gs)
{
    for (int i = 0; i < MAX_PROJECTILES; i++) {
        projectile_t *p = &gs->projectiles[i];
        if (!p->active)
            continue;

        p->x_pixel += PEA_SPEED;

        /* Remove if off-screen */
        if (p->x_pixel > SCREEN_W)
            p->active = 0;
    }
}

/* Fire peas from peashooters that have zombies in their row */
static void update_firing(game_state_t *gs)
{
    for (int r = 0; r < GRID_ROWS; r++) {
        for (int c = 0; c < GRID_COLS; c++) {
            plant_t *p = &gs->grid[r][c];
            if (p->type != PLANT_PEASHOOTER)
                continue;

            if (p->fire_cooldown > 0)
                p->fire_cooldown--;

            if (p->fire_cooldown == 0 && zombie_in_row(gs, r)) {
                spawn_pea(gs, r, c);
                p->fire_cooldown = PLANT_FIRE_COOLDOWN;
            }
        }
    }
}
```

```
/* Check pea-zombie collisions */
static void check_collisions(game_state_t *gs)
{
    for (int i = 0; i < MAX_PROJECTILES; i++) {
        projectile_t *p = &gs->projectiles[i];
        if (!p->active)
            continue;

        for (int j = 0; j < MAX_ZOMBIES; j++) {
            zombie_t *z = &gs->zombies[j];
            if (!z->active || z->row != p->row)
                continue;

            /* Collision: pea overlaps zombie bounding box */
            int z_left = z->x_pixel;
            int z_right = z->x_pixel + ZOMBIE_WIDTH;
            int p_left = p->x_pixel;
            int p_right = p->x_pixel + PEA_SIZE;

            if (p_right >= z_left && p_left <= z_right) {
                /* Hit! */
                z->hp -= PEA_DAMAGE;
                p->active = 0;

                if (z->hp <= 0)
                    z->active = 0;

                break; /* Each pea hits only one zombie */
            }
        }
    }
}

/* Spawn zombies on a timer */
static void update_spawning(game_state_t *gs)
{
    if (gs->zombies_spawned >= TOTAL_ZOMBIES)
        return;

    gs->spawn_timer--;
    if (gs->spawn_timer <= 0) {
        /* Find a free zombie slot */
        for (int i = 0; i < MAX_ZOMBIES; i++) {
            if (!gs->zombies[i].active) {
                gs->zombies[i].active = 1;
                gs->zombies[i].row = random_range(0, GRID_ROWS - 1);
                gs->zombies[i].x_pixel = SCREEN_W - 1;
                gs->zombies[i].hp = ZOMBIE_HP;
                gs->zombies[i].move_counter = 0;
```

```
                gs->zombies[i].eating = 0;
                gs->zombies[i].eat_timer = 0;
                gs->zombies_spawned++;
                break;
            }
        }
        gs->spawn_timer = random_range(ZOMBIE_SPAWN_MIN, ZOMBIE_SPAWN_MAX);
    }
}

/* Count sunflowers currently on the field */
static int count_sunflowers(const game_state_t *gs)
{
    int n = 0;
    for (int r = 0; r < GRID_ROWS; r++)
        for (int c = 0; c < GRID_COLS; c++)
            if (gs->grid[r][c].type == PLANT_SUNFLOWER)
                n++;
    return n;
}

/* Update sun economy.  Base rate plus one extra increment per sunflower. */
static void update_sun(game_state_t *gs)
{
    gs->sun_timer--;
    if (gs->sun_timer <= 0) {
        gs->sun += SUN_INCREMENT * (1 + count_sunflowers(gs));
        gs->sun_timer = SUN_INTERVAL;
    }
}

/* Check win condition */
static void check_win(game_state_t *gs)
{
    if (gs->zombies_spawned < TOTAL_ZOMBIES)
        return;

    for (int i = 0; i < MAX_ZOMBIES; i++) {
        if (gs->zombies[i].active)
            return;
    }

    gs->state = STATE_WIN;
}

void game_update(game_state_t *gs)
{
    if (gs->state != STATE_PLAYING)
        return;
```

```
    gs->frame_count++;

    update_sun(gs);
    update_spawning(gs);
    update_firing(gs);
    update_projectiles(gs);
    update_zombies(gs);
    check_collisions(gs);
    check_win(gs);
}
```

`sw/input.h`

```
#ifndef PVZ_INPUT_H
#define PVZ_INPUT_H

/* Input event codes returned by input_poll() */
#define INPUT_NONE   0
#define INPUT_UP     1
#define INPUT_DOWN   2
#define INPUT_LEFT   3
#define INPUT_RIGHT  4
#define INPUT_SPACE  5
#define INPUT_D      6
#define INPUT_ESC    7
#define INPUT_TAB    8

/*
 * Initialize keyboard input from a Linux input device.
 * Returns 0 on success, -1 on failure.
 */
int input_init(const char *device_path);

/*
 * Poll for a keyboard event (non-blocking).
 * Returns one of the INPUT_* codes, or INPUT_NONE if no key pressed.
 */
int input_poll(void);

/*
 * Close the input device.
 */
void input_close(void);

#endif /* PVZ_INPUT_H */
```

`sw/input.c`

```
/*
 * Keyboard / gamepad input via /dev/input/eventX
 *
 * Uses non-blocking I/O to read events from a Linux input device.
 * Maps both keyboard keys and Xbox 360-compatible gamepad buttons
 * (xpad: BTN_DPAD_*, BTN_SOUTH/EAST, BTN_TL/TR, BTN_START, plus
 * ABS_HAT0X/Y for variants that report the D-pad as hat axes) to
 * the same INPUT_* action codes consumed by main.c.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/input.h>
#include "input.h"

static int input_fd = -1;

int input_init(const char *device_path)
{
    input_fd = open(device_path, O_RDONLY | O_NONBLOCK);
    if (input_fd < 0) {
        perror("input_init: open");
        return -1;
    }
    return 0;
}

int input_poll(void)
{
    struct input_event ev;

    if (input_fd < 0)
        return INPUT_NONE;

    while (read(input_fd, &ev, sizeof(ev)) == sizeof(ev)) {
        /* Some xpad variants report the D-pad as absolute hat axes
         * rather than BTN_DPAD_* keys. value == 0 is the release
         * event; ignore it so an idle hat doesn't spam INPUT_NONE. */
        if (ev.type == EV_ABS) {
            if (ev.code == ABS_HAT0X) {
                if (ev.value < 0) return INPUT_LEFT;
                if (ev.value > 0) return INPUT_RIGHT;
            } else if (ev.code == ABS_HAT0Y) {
```

```
                if (ev.value < 0) return INPUT_UP;
                if (ev.value > 0) return INPUT_DOWN;
            }
            continue;
        }

        /* Only handle key/button press events (value == 1) */
        if (ev.type != EV_KEY || ev.value != 1)
            continue;

        switch (ev.code) {
        case KEY_UP:    case BTN_DPAD_UP:    return INPUT_UP;
        case KEY_DOWN:  case BTN_DPAD_DOWN:  return INPUT_DOWN;
        case KEY_LEFT:  case BTN_DPAD_LEFT:  return INPUT_LEFT;
        case KEY_RIGHT: case BTN_DPAD_RIGHT: return INPUT_RIGHT;
        case KEY_SPACE: case BTN_SOUTH:      return INPUT_SPACE;
        case KEY_D:     case BTN_EAST:       return INPUT_D;
        case KEY_ESC:   case BTN_START:      return INPUT_ESC;
        case KEY_TAB:   case BTN_TL: case BTN_TR: return INPUT_TAB;
        default:        break;
        }
    }

    return INPUT_NONE;
}

void input_close(void)
{
    if (input_fd >= 0) {
        close(input_fd);
        input_fd = -1;
    }
}
```

`sw/render.h`

```
#ifndef _RENDER_H
#define _RENDER_H

#include "game.h"

/*
 * Initialize the renderer (set up the FPGA fd).
 * Returns 0 on success, -1 on failure.
 */
int render_init(int fpga_fd);

/*
 * Render the full game state to the FPGA.
 * Converts game state into background cells + shape table entries
 * and writes them via ioctls.
 */
void render_frame(const game_state_t *gs);

#endif /* _RENDER_H */
```

sw/render.c

```
/*
 * Game state -> FPGA register file
 *
 * Each frame we walk the game state and write one 32-bit value per
 * entity into the hardware register file via the PVZ_WRITE_REG ioctl.
 * No commit handshake: the entity_drawer reads whatever is in the
 * registers as it scans each line.
 *
 * Hardware capacity: 32 plant cells, 8 zombies, 8 peas, 1 cursor.
 */

#include <stdio.h>
#include <stdint.h>
#include <sys/ioctl.h>
#include "render.h"
#include "pvz.h"

static int fd;

static void write_reg(unsigned int word_index, unsigned int value)
{
    pvz_write_arg_t w = { .word_index = word_index, .value = value };
    ioctl(fd, PVZ_WRITE_REG, &w);
}

int render_init(int fpga_fd)
{
    fd = fpga_fd;
    return 0;
}

static void render_plants(const game_state_t *gs)
{
    uint32_t pea_bits = 0;
    uint32_t sun_bits = 0;
    for (int r = 0; r < GRID_ROWS; r++) {
        for (int c = 0; c < GRID_COLS; c++) {
            int t = gs->grid[r][c].type;
            if (t == PLANT_PEASHOOTER)
                pea_bits |= (1u << (r * 8 + c));
            else if (t == PLANT_SUNFLOWER)
                sun_bits |= (1u << (r * 8 + c));
        }
    }
    write_reg(PVZ_REG_PLANTS,    pea_bits);
    write_reg(PVZ_REG_SUNFLOWER, sun_bits);
```

```
}

static void render_selected(const game_state_t *gs)
{
    int sel = (gs->selected_plant_type == PLANT_SUNFLOWER) ? 1 : 0;
    write_reg(PVZ_REG_SELECTED, sel);
}

static void render_zombies(const game_state_t *gs)
{
    for (int i = 0; i < PVZ_MAX_ZOMBIES; i++) {
        int alive = 0, row = 0, x = 0;
        if (i < MAX_ZOMBIES && gs->zombies[i].active) {
            alive = 1;
            row   = gs->zombies[i].row;
            x     = gs->zombies[i].x_pixel;
        }
        write_reg(PVZ_REG_ZOMBIE(i), pvz_pack_entity(alive, row, x));
    }
}

static void render_peas(const game_state_t *gs)
{
    /* Walk active projectiles in order, stop after PVZ_MAX_PEAS.
     * Hide hardware slots that don't get filled. */
    int slot = 0;
    for (int i = 0; i < MAX_PROJECTILES && slot < PVZ_MAX_PEAS; i++) {
        const projectile_t *p = &gs->projectiles[i];
        if (!p->active) continue;
        write_reg(PVZ_REG_PEA(slot), pvz_pack_entity(1, p->row, p->x_pixel));
        slot++;
    }
    for (; slot < PVZ_MAX_PEAS; slot++)
        write_reg(PVZ_REG_PEA(slot), 0);
}

static void render_cursor(const game_state_t *gs)
{
    int visible = (gs->state == STATE_PLAYING) ? 1 : 0;
    write_reg(PVZ_REG_CURSOR,
              pvz_pack_cursor(visible, gs->cursor_row, gs->cursor_col));
}

static void render_sun(const game_state_t *gs)
{
    /* HW currently doesn't draw the sun count; the register is reserved
     * so a future HUD module can pick it up. */
    write_reg(PVZ_REG_SUN, gs->sun & 0x3FFF);
}
```

```c
static void hide_all_entities(void)
{
    write_reg(PVZ_REG_PLANTS,    0);
    write_reg(PVZ_REG_SUNFLOWER, 0);
    for (int i = 0; i < PVZ_MAX_ZOMBIES; i++)
        write_reg(PVZ_REG_ZOMBIE(i), 0);
    for (int i = 0; i < PVZ_MAX_PEAS; i++)
        write_reg(PVZ_REG_PEA(i), 0);
    write_reg(PVZ_REG_CURSOR, 0);
}

void render_frame(const game_state_t *gs)
{
    if (gs->state == STATE_PLAYING) {
        render_plants(gs);
        render_zombies(gs);
        render_peas(gs);
        render_cursor(gs);
        render_sun(gs);
        render_selected(gs);
    } else {
        /* WIN / LOSE: clear everything; main loop prints a banner and
         * exits.  A future HUD pass can render a result indicator. */
        hide_all_entities();
        render_sun(gs);
        render_selected(gs);
    }
}
```

`sw/pvz_driver.h`

```
#ifndef _PVZ_DRIVER_H
#define _PVZ_DRIVER_H

#include <linux/miscdevice.h>
#include <linux/platform_device.h>

#define DRIVER_NAME "pvz"

struct pvz_dev {
    struct resource res;
    void __iomem *virtbase;
};

#endif /* _PVZ_DRIVER_H */
```

`sw/pvz_driver.c`

```c
/*
 * PvZ GPU kernel driver
 *
 * Misc device at /dev/pvz with a single ioctl, PVZ_WRITE_REG, that
 * writes one 32-bit value into the FPGA register file.  Avalon
 * addressing is in WORDS (set in pvz_top_hw.tcl) so the byte offset
 * is `word_index * 4`.
 *
 * Compatible: csee4840,pvz_gpu-1.0
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "pvz.h"
#include "pvz_driver.h"

static struct pvz_dev dev;

static long pvz_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    pvz_write_arg_t w;

    switch (cmd) {
    case PVZ_WRITE_REG:
        if (copy_from_user(&w, (pvz_write_arg_t *)arg, sizeof(w)))
            return -EACCES;
        if (w.word_index >= PVZ_NUM_REGS)
            return -EINVAL;
        iowrite32(w.value, dev.virtbase + (w.word_index * 4));
        break;

    default:
        return -EINVAL;
    }
```

```
    return 0;
}

static const struct file_operations pvz_fops = {
    .owner          = THIS_MODULE,
    .unlocked_ioctl = pvz_ioctl,
};

static struct miscdevice pvz_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = DRIVER_NAME,
    .fops  = &pvz_fops,
};

static int __init pvz_probe(struct platform_device *pdev)
{
    int ret;

    ret = misc_register(&pvz_misc_device);
    if (ret) {
        pr_err(DRIVER_NAME ": misc_register failed\n");
        return ret;
    }

    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                           DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    pr_info(DRIVER_NAME ": initialized at 0x%08lx\n",
            (unsigned long)dev.res.start);

    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
```

```
out_deregister:
    misc_deregister(&pvz_misc_device);
    return ret;
}

static int pvz_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&pvz_misc_device);
    pr_info(DRIVER_NAME ": removed\n");
    return 0;
}

#ifdef CONFIG_OF
static const struct of_device_id pvz_of_match[] = {
    { .compatible = "csee4840,pvz_gpu-1.0" },
    {},
};
MODULE_DEVICE_TABLE(of, pvz_of_match);
#endif

static struct platform_driver pvz_driver = {
    .driver = {
        .name           = DRIVER_NAME,
        .owner          = THIS_MODULE,
        .of_match_table = of_match_ptr(pvz_of_match),
    },
    .remove = __exit_p(pvz_remove),
};

static int __init pvz_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&pvz_driver, pvz_probe);
}

static void __exit pvz_exit(void)
{
    platform_driver_unregister(&pvz_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(pvz_init);
module_exit(pvz_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("CSEE4840 Team");
MODULE_DESCRIPTION("PvZ GPU display engine driver");
```

`sw/main.c`

```c
/*
 * Plants vs Zombies — Main game loop
 *
 * Initializes the FPGA driver, keyboard input, and game state,
 * then runs a 60 Hz loop: input -> update -> render -> write to FPGA.
 *
 * Usage: ./pvz [/dev/input/eventN]
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

#include "pvz.h"
#include "game.h"
#include "input.h"
#include "render.h"

#define FRAME_USEC 16667  /* ~60 fps */

static int pvz_fd;

/* Get current time in microseconds */
static long long get_time_usec(void)
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (long long)tv.tv_sec * 1000000LL + tv.tv_usec;
}

static void process_input(game_state_t *gs)
{
    int key;

    while ((key = input_poll()) != INPUT_NONE) {
        switch (key) {
        case INPUT_UP:
            if (gs->cursor_row > 0) gs->cursor_row--;
            break;
        case INPUT_DOWN:
            if (gs->cursor_row < GRID_ROWS - 1) gs->cursor_row++;
            break;
        case INPUT_LEFT:
```

```
            if (gs->cursor_col > 0) gs->cursor_col--;
            break;
        case INPUT_RIGHT:
            if (gs->cursor_col < GRID_COLS - 1) gs->cursor_col++;
            break;
        case INPUT_SPACE:
            game_place_plant(gs);
            break;
        case INPUT_D:
            game_remove_plant(gs);
            break;
        case INPUT_TAB:
            gs->selected_plant_type =
                (gs->selected_plant_type == PLANT_PEASHOOTER)
                ? PLANT_SUNFLOWER : PLANT_PEASHOOTER;
            break;
        case INPUT_ESC:
            gs->state = -1; /* signal exit */
            return;
        }
    }
}

int main(int argc, char *argv[])
{
    const char *input_dev = "/dev/input/event0";
    game_state_t gs;

    if (argc > 1)
        input_dev = argv[1];

    /* Seed random number generator */
    srand((unsigned)time(NULL));

    /* Open FPGA device */
    pvz_fd = open("/dev/pvz", O_RDWR);
    if (pvz_fd < 0) {
        perror("open /dev/pvz");
        fprintf(stderr, "Is the pvz_driver module loaded?\n");
        return 1;
    }

    /* Initialize keyboard input */
    if (input_init(input_dev) < 0) {
        fprintf(stderr, "Failed to open %s\n", input_dev);
        fprintf(stderr, "Usage: %s [/dev/input/eventN]\n", argv[0]);
        close(pvz_fd);
        return 1;
    }
```

```
    /* Initialize renderer */
    render_init(pvz_fd);

    /* Initialize game state */
    game_init(&gs);

    printf("Plants vs Zombies MVP\n");
    printf("Controls: Arrows=move, Tab=toggle plant, Space=place, D=remove,
ESC=quit\n");
    printf("Sun: %d | Plant cost: %d\n\n", gs.sun, PLANT_COST);

    /* Main game loop */
    long long frame_start;

    while (gs.state >= 0) {
        frame_start = get_time_usec();

        /* 1. Process input */
        process_input(&gs);
        if (gs.state < 0)
            break;

        /* 2. Update game logic */
        game_update(&gs);

        /* 3. Render to FPGA */
        render_frame(&gs);

        /* Print status periodically */
        if (gs.frame_count % 60 == 0) {
            printf("\rSun: %3d | Zombies: %d/%d | Frame: %d",
                   gs.sun, gs.zombies_spawned, TOTAL_ZOMBIES,
                   gs.frame_count);
            fflush(stdout);
        }

        /* Check game over */
        if (gs.state == STATE_WIN) {
            printf("\n\n*** YOU WIN! All zombies defeated! ***\n");
            render_frame(&gs); /* Render win state */
            sleep(5);
            break;
        }
        if (gs.state == STATE_LOSE) {
            printf("\n\n*** GAME OVER! A zombie reached your house! ***\n");
            render_frame(&gs); /* Render lose state */
            sleep(5);
            break;
```

```
        }

        /* 4. Frame timing: sleep for remainder of frame */
        long long elapsed = get_time_usec() - frame_start;
        if (elapsed < FRAME_USEC)
            usleep(FRAME_USEC - elapsed);
    }

    printf("\nCleaning up...\n");
    input_close();
    close(pvz_fd);
    return 0;
}
```

Hardware (FPGA, SystemVerilog)

hw/pvz_top.sv

```
/*
 * PvZ GPU — top-level Avalon-MM peripheral
 *
 * Wires together VGA timing, background grid, sprite ROM, and the
 * entity drawer.  Holds the entity register file that software writes
 * via Avalon.
 *
 * Register map (32-bit words, byte offset = 4 * word index):
 *   word  0       PVZ_PLANTS    32 bits, bit i = peashooter at cell i
 *                                (i = row*8 + col, row in 0..3, col in 0..7)
 *   word  1       PVZ_SUNFLOWER 32 bits, bit i = sunflower at cell i
 *   word 32..39  PVZ_ZOMBIE[i]  bit 31 = alive
 *                                bits [9:0]   = x_pixel (0..639)
 *                                bits [11:10] = row (0..3)
 *   word 40..47  PVZ_PEA[i]     same encoding as zombie
 *   word 48      PVZ_CURSOR     bit 31 = visible
 *                                bits [4:2] = col (0..7)
 *                                bits [1:0] = row (0..3)
 *   word 49      PVZ_SUN        bits [13:0] = sun value
 *   word 50      PVZ_SELECTED   bits [1:0]  = selected plant (0=pea,
1=sunflower)
 *
 * Avalon notes:
 *   - The address port is in WORDS (qsys addressUnits = WORDS) so a
 *     CPU byte offset N maps to address = N >> 2.
 *   - Writes take effect on the next clock; no commit handshake.
 *   - There is no vsync latching, so a write that races the scan can
 *     produce one frame of tearing.  Acceptable for ~60 Hz game state.
 */

module pvz_top(
    input  logic        clk,
    input  logic        reset,

    // Avalon-MM slave
    input  logic [5:0]  address,    // word index, 0..49 used, 64 max
    input  logic [31:0] writedata,
    input  logic        write,
    input  logic        chipselect,

    // VGA output
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS,
```

```
    output logic        VGA_BLANK_n,
    output logic        VGA_SYNC_n
);

    // ----------------------------------------------------------------
    // VGA timing
    // ----------------------------------------------------------------
    logic [10:0] hcount;
    logic [9:0]  vcount;

    vga_counters counters(
        .clk50      (clk),
        .reset      (reset),
        .hcount     (hcount),
        .vcount     (vcount),
        .VGA_CLK    (VGA_CLK),
        .VGA_HS     (VGA_HS),
        .VGA_VS     (VGA_VS),
        .VGA_BLANK_n(VGA_BLANK_n),
        .VGA_SYNC_n (VGA_SYNC_n)
    );

    wire [9:0] px = hcount[10:1];
    wire [9:0] py = vcount;

    // ----------------------------------------------------------------
    // Entity register file
    // ----------------------------------------------------------------
    // Plants: one bit per grid cell (32 cells)
    logic [31:0] plant_present;
    logic [31:0] sunflower_present;

    // Currently selected plant type for the top HUD box (0=pea, 1=sunflower)
    logic [1:0]  selected_plant;

    // Zombies and peas: 8 each.  Alive bits packed; x and row also
    // packed into wide buses for handing to entity_drawer.
    logic [7:0]  zombie_alive, pea_alive;
    logic [9:0]  zombie_x   [0:7];
    logic [1:0]  zombie_row [0:7];
    logic [9:0]  pea_x      [0:7];
    logic [1:0]  pea_row    [0:7];

    // Cursor + sun
    logic        cursor_visible;
    logic [2:0]  cursor_col;
    logic [1:0]  cursor_row;
    logic [13:0] sun_value;
```

```
    // ---------------------------------------------------------------
    // Avalon-MM write decode
    // ---------------------------------------------------------------
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            plant_present     <= 32'd0;
            sunflower_present <= 32'd0;
            selected_plant    <= 2'd0;
            zombie_alive   <= 8'd0;
            pea_alive      <= 8'd0;
            cursor_visible <= 1'b0;
            cursor_col     <= 3'd0;
            cursor_row     <= 2'd0;
            sun_value      <= 14'd0;
            for (int i = 0; i < 8; i++) begin
                zombie_x[i]   <= 10'd0;
                zombie_row[i] <= 2'd0;
                pea_x[i]      <= 10'd0;
                pea_row[i]    <= 2'd0;
            end
        end else if (chipselect && write) begin
            // Peashooter bitmap: word 0
            if (address == 6'd0) begin
                plant_present <= writedata;
            end
            // Sunflower bitmap: word 1
            else if (address == 6'd1) begin
                sunflower_present <= writedata;
            end
            // Zombies: word 32..39
            else if (address < 6'd40) begin
                zombie_alive[address[2:0]] <= writedata[31];
                zombie_x[address[2:0]]     <= writedata[9:0];
                zombie_row[address[2:0]]   <= writedata[11:10];
            end
            // Peas: word 40..47
            else if (address < 6'd48) begin
                pea_alive[address[2:0]] <= writedata[31];
                pea_x[address[2:0]]     <= writedata[9:0];
                pea_row[address[2:0]]   <= writedata[11:10];
            end
            // Cursor: word 48
            else if (address == 6'd48) begin
                cursor_visible <= writedata[31];
                cursor_col     <= writedata[4:2];
                cursor_row     <= writedata[1:0];
            end
            // Sun: word 49
            else if (address == 6'd49) begin
```

```
                sun_value <= writedata[13:0];
            end
            // Selected plant: word 50
            else if (address == 6'd50) begin
                selected_plant <= writedata[1:0];
            end
        end
    end

    // Pack arrays into wide buses for the drawer module
    logic [79:0] zombie_x_packed, pea_x_packed;
    logic [15:0] zombie_row_packed, pea_row_packed;
    genvar gi;
    generate
        for (gi = 0; gi < 8; gi++) begin : pack_entities
            assign zombie_x_packed[gi*10 +: 10]    = zombie_x[gi];
            assign zombie_row_packed[gi*2 +: 2]    = zombie_row[gi];
            assign pea_x_packed[gi*10 +: 10]       = pea_x[gi];
            assign pea_row_packed[gi*2 +: 2]       = pea_row[gi];
        end
    endgenerate

    // ---------------------------------------------------------------
    // Background grid (combinational, no state)
    // ---------------------------------------------------------------
    logic [7:0] bg_color;
    bg_grid bg_inst(
        .px       (px),
        .py       (py),
        .color_out(bg_color)
    );

    // ---------------------------------------------------------------
    // Sprite ROMs (64x64 each, 4096 bytes, 1-cycle read latency).
    // One ROM per sprite type; both share the same module.
    // ---------------------------------------------------------------
    logic [11:0] plant_addr;
    logic [7:0]  plant_pixel;
    sprite_rom #(.MEM_FILE("peashooter_idx.mem")) plant_rom_inst(
        .clk  (clk),
        .addr (plant_addr),
        .pixel(plant_pixel)
    );

    // Sunflower ROM shares the cell-local address with the peashooter ROM
    logic [7:0]  sunflower_pixel;
    sprite_rom #(.MEM_FILE("sunflower_idx.mem")) sunflower_rom_inst(
        .clk  (clk),
        .addr (plant_addr),
```

```
        .pixel(sunflower_pixel)
    );

    logic [11:0] zombie_addr;
    logic [7:0]  zombie_pixel;
    sprite_rom #(.MEM_FILE("zombie_idx.mem")) zombie_rom_inst(
        .clk  (clk),
        .addr (zombie_addr),
        .pixel(zombie_pixel)
    );

    // ---------------------------------------------------------------
    // Entity drawer: produces the final pixel color
    // ---------------------------------------------------------------
    logic [7:0] pixel_color;
    entity_drawer drawer_inst(
        .clk              (clk),
        .reset            (reset),
        .px               (px),
        .py               (py),
        .bg_color         (bg_color),
        .plant_present    (plant_present),
        .sunflower_present(sunflower_present),
        .selected_plant   (selected_plant),
        .zombie_alive     (zombie_alive),
        .zombie_x_packed  (zombie_x_packed),
        .zombie_row_packed(zombie_row_packed),
        .pea_alive        (pea_alive),
        .pea_x_packed     (pea_x_packed),
        .pea_row_packed   (pea_row_packed),
        .cursor_visible   (cursor_visible),
        .cursor_col       (cursor_col),
        .cursor_row       (cursor_row),
        .sun_value        (sun_value),
        .plant_rd_addr    (plant_addr),
        .plant_rd_pixel   (plant_pixel),
        .sunflower_rd_pixel(sunflower_pixel),
        .zombie_rd_addr   (zombie_addr),
        .zombie_rd_pixel  (zombie_pixel),
        .color_out        (pixel_color)
    );

    // ---------------------------------------------------------------
    // Palette: 8-bit color index -> 24-bit RGB.  Black during blanking.
    // ---------------------------------------------------------------
    logic [7:0] pal_r, pal_g, pal_b;
    color_palette pal_inst(
        .index(pixel_color),
        .r    (pal_r),
```

```
        .g     (pal_g),
        .b     (pal_b)
    );

    always_comb begin
        if (VGA_BLANK_n) begin
            VGA_R = pal_r;
            VGA_G = pal_g;
            VGA_B = pal_b;
        end else begin
            VGA_R = 8'h00;
            VGA_G = 8'h00;
            VGA_B = 8'h00;
        end
    end

endmodule
```

`hw/entity_drawer.sv`

```
/*
 * Entity drawer — "racing the beam" combinational pixel renderer
 *
 * For every VGA pixel (px, py), determines the final color by checking
 * which game entity (if any) covers that pixel.  No frame buffer, no
 * line buffer, no FSM: the rendering happens in lock-step with the VGA
 * scan.
 *
 * Layering (low to high; later overwrites earlier):
 *   1. bg           (lawn checker, from bg_grid)
 *   2. plant        (64x64 sprite ROM, 1:1)
 *   3. pea          (small bright-green square)
 *   4. zombie       (64x64 sprite ROM, 1:1, with transparency)
 *   5. cursor       (yellow border around cursor cell)
 *   6. sun HUD      (yellow blocks at top, one per 100 sun)
 *   7. selector     (two plant icons at top-left; the chosen one
 *                    wears the yellow border, which TAB cycles)
 *
 * Sprite ROMs have 1 clock of read latency.  Stage 1 issues addresses
 * combinationally, stage 2 (one cycle later) merges the ROM outputs with
 * the registered overlay hits.  Final color is registered so the VGA
 * data path stays clean.
 *
 * Layout constants (must match bg_grid.sv and software):
 *   GRID_X = 64, GRID_Y = 112, CELL = 64, GRID_COLS = 8, GRID_ROWS = 4
 */

module entity_drawer(
    input  logic        clk,
    input  logic        reset,

    // Pixel coordinates from VGA scan
    input  logic [9:0]  px,
    input  logic [9:0]  py,

    // Background color for this pixel (combinational from bg_grid)
    input  logic [7:0]  bg_color,

    // ---------- Entity registers (driven by pvz_top register file) ---------
-
    // Plants: one bit per grid cell, bit (row*8+col).
    input  logic [31:0] plant_present,
    input  logic [31:0] sunflower_present,

    // Currently selected plant for the top HUD box (0=pea, 1=sunflower)
    input  logic [1:0]  selected_plant,
```

```
    // Up to 8 zombies / 8 peas. Packed so we don't need unpacked-array
    // ports (more portable across synthesis tools).
    //   zombie_x[i]   = pixel x position (0..639)   width 10
    //   zombie_row[i] = grid row (0..3)             width 2
    //   zombie_alive  = bit i high if zombie i is on screen
    input  logic [7:0]  zombie_alive,
    input  logic [79:0] zombie_x_packed,    // {zombie_x[7], ..., zombie_x[0]}
    input  logic [15:0] zombie_row_packed,  // {zombie_row[7], ...,
zombie_row[0]}

    input  logic [7:0]  pea_alive,
    input  logic [79:0] pea_x_packed,
    input  logic [15:0] pea_row_packed,

    // Cursor: a hollow yellow border around one cell
    input  logic        cursor_visible,
    input  logic [2:0]  cursor_col,
    input  logic [1:0]  cursor_row,

    // Sun count for HUD (each block = 100 sun, up to 10 blocks)
    input  logic [13:0] sun_value,

    // Plant sprite ROM read interface (1-cycle read latency).
    // Sunflower ROM shares the same address (issued via plant_rd_addr).
    output logic [11:0] plant_rd_addr,
    input  logic [7:0]  plant_rd_pixel,
    input  logic [7:0]  sunflower_rd_pixel,

    // Zombie sprite ROM read interface (1-cycle read latency)
    output logic [11:0] zombie_rd_addr,
    input  logic [7:0]  zombie_rd_pixel,

    // Final pixel color (registered, 1 cycle of latency vs px/py)
    output logic [7:0]  color_out
);

    // ---------------------------------------------------------------
    // Color indices (must match color_palette.sv)
    // ---------------------------------------------------------------
    localparam logic [7:0] COL_YELLOW       = 8'd4;
    localparam logic [7:0] COL_GREEN        = 8'd7;
    localparam logic [7:0] COL_BRIGHT_GREEN = 8'd9;
    localparam logic [7:0] COL_ORANGE       = 8'd12;
    localparam logic [7:0] COL_TRANSPARENT  = 8'hFF;

    // Layout constants (mirror bg_grid.sv)
    localparam logic [9:0] GRID_X     = 10'd64;
    localparam logic [9:0] GRID_Y     = 10'd112;
```

```
    localparam logic [9:0] CELL       = 10'd64;

    // Entity sprite sizes
    localparam logic [9:0] ZOMBIE_W = 10'd64;
    localparam logic [9:0] ZOMBIE_H = 10'd64;
    localparam logic [9:0] PEA_SIZE = 10'd8;
    localparam logic [9:0] CURSOR_BORDER = 10'd4;

    // Sun HUD layout: 10 blocks across the top of the screen
    localparam logic [9:0]  SUN_X     = 10'd440;
    localparam logic [9:0]  SUN_Y     = 10'd24;
    localparam logic [9:0]  SUN_BW    = 10'd16;  // block width
    localparam logic [9:0]  SUN_BH    = 10'd24;  // block height
    localparam logic [9:0]  SUN_PITCH = 10'd18;  // block + 2 px gap
    localparam logic [13:0] SUN_PER_BLOCK = 14'd50;

    // Plant-selector HUD: two always-visible icon boxes at top-left,
    // one per plant type.  Box 0 (peashooter) is green; box 1
    // (sunflower) is orange.  Only the currently selected box wears
    // the yellow border, so TAB visibly cycles the "cursor" between
    // the two icons.  Both boxes sit above the grid (GRID_Y=112) and
    // well clear of the sun HUD on the right side of the screen.
    localparam logic [9:0] SEL_X0     = 10'd8;     // peashooter icon
    localparam logic [9:0] SEL_X1     = 10'd64;    // sunflower icon (8 + 48 +
8 px gap)
    localparam logic [9:0] SEL_Y      = 10'd8;
    localparam logic [9:0] SEL_SZ     = 10'd48;
    localparam logic [9:0] SEL_BORDER = 10'd4;

    // ---------------------------------------------------------------
    // Unpack the zombie/pea arrays into indexable arrays
    // ---------------------------------------------------------------
    logic [9:0] zombie_x   [0:7];
    logic [1:0] zombie_row [0:7];
    logic [9:0] pea_x      [0:7];
    logic [1:0] pea_row    [0:7];

    genvar gi;
    generate
        for (gi = 0; gi < 8; gi++) begin : unpack_entities
            assign zombie_x[gi]   = zombie_x_packed[gi*10 +: 10];
            assign zombie_row[gi] = zombie_row_packed[gi*2 +: 2];
            assign pea_x[gi]      = pea_x_packed[gi*10 +: 10];
            assign pea_row[gi]    = pea_row_packed[gi*2 +: 2];
        end
    endgenerate

    // ---------------------------------------------------------------
    // Stage 1: figure out which grid cell the current pixel is in,
```

```
// and issue the plant sprite ROM read for that cell.
// ---------------------------------------------------------------
wire in_grid_x = (px >= GRID_X) && (px < GRID_X + 10'd512);
wire in_grid_y = (py >= GRID_Y) && (py < GRID_Y + 10'd256);
wire in_grid   = in_grid_x && in_grid_y;

/* verilator lint_off UNUSED */
wire [9:0] gx = px - GRID_X;
wire [9:0] gy = py - GRID_Y;
/* verilator lint_on UNUSED */

// Cell index and within-cell pixel (cells are 64 px = 2^6).  Sprite
// is 64x64 so we use the full 6 bits of in_cell_{x,y} as the address.
wire [2:0] cell_col  = gx[8:6];
wire [1:0] cell_row  = gy[7:6];
wire [5:0] in_cell_x = gx[5:0];
wire [5:0] in_cell_y = gy[5:0];

// Plant / sunflower bits for this cell (false if outside grid)
wire [4:0] plant_idx = {cell_row, cell_col};
wire plant_here     = in_grid && plant_present[plant_idx];
wire sunflower_here = in_grid && sunflower_present[plant_idx];

// Plant sprite ROM address: 1:1 mapping (64x64 ROM into 64x64 cell).
assign plant_rd_addr = {in_cell_y, in_cell_x};

// ---------------------------------------------------------------
// Stage 1: zombie hit detection AND zombie sprite ROM address.
// Priority encoder: first alive zombie covering this pixel wins
// (zombies don't normally overlap on screen).
// ---------------------------------------------------------------
logic        zombie_hit_comb;
logic [5:0]  zombie_in_x, zombie_in_y;
always_comb begin
    zombie_hit_comb = 1'b0;
    zombie_in_x     = 6'd0;
    zombie_in_y     = 6'd0;
    for (int i = 0; i < 8; i++) begin
        logic [9:0] zy_top, dx, dy;
        zy_top = GRID_Y + ({8'd0, zombie_row[i]} << 6);
        dx     = px - zombie_x[i];
        dy     = py - zy_top;
        if (zombie_alive[i] && !zombie_hit_comb &&
            px >= zombie_x[i] && px < zombie_x[i] + ZOMBIE_W &&
            py >= zy_top      && py < zy_top      + ZOMBIE_H)
        begin
            zombie_hit_comb = 1'b1;
            zombie_in_x     = dx[5:0];
            zombie_in_y     = dy[5:0];
```

```
            end
        end
    end
    assign zombie_rd_addr = {zombie_in_y, zombie_in_x};

    // -----------------------------------------------------------------
    // Stage 1: combinational hit detection for non-sprite entities
    // -----------------------------------------------------------------
    logic pea_hit_comb;
    always_comb begin
        pea_hit_comb = 1'b0;
        for (int i = 0; i < 8; i++) begin
            // Center the pea vertically in its row (row*64 + 28..36)
            logic [9:0] py_top;
            py_top = GRID_Y + ({8'd0, pea_row[i]} << 6) + 10'd28;
            if (pea_alive[i] &&
                px >= pea_x[i] && px < pea_x[i] + PEA_SIZE &&
                py >= py_top && py < py_top + PEA_SIZE)
                pea_hit_comb = 1'b1;
        end
    end

    // Cursor: hollow border around the cursor cell
    logic cursor_hit_comb;
    always_comb begin
        logic [9:0] cur_left, cur_top;
        cur_left = GRID_X + ({7'd0, cursor_col} << 6);
        cur_top  = GRID_Y + ({8'd0, cursor_row} << 6);
        cursor_hit_comb = 1'b0;
        if (cursor_visible &&
            px >= cur_left && px < cur_left + CELL &&
            py >= cur_top  && py < cur_top  + CELL) begin
            // On border? (within CURSOR_BORDER pixels of any edge)
            if ( (px - cur_left)        < CURSOR_BORDER ||
                 (cur_left + CELL - px) <= CURSOR_BORDER ||
                 (py - cur_top)         < CURSOR_BORDER ||
                 (cur_top  + CELL - py) <= CURSOR_BORDER )
                cursor_hit_comb = 1'b1;
        end
    end

    // Plant selector HUD: two icon boxes, one per plant type.  Both
    // fills are always drawn so the player can see the available
    // plants at a glance; the border is gated by selected_plant
    // later in the mux so only the chosen box looks like a cursor.
    //   selN_hit_comb    = anywhere inside box N (fill region)
    //   selN_border_comb = on box N's border (yellow cursor look)
    logic sel0_hit_comb, sel0_border_comb;
    logic sel1_hit_comb, sel1_border_comb;
```

```
    always_comb begin
        // Box 0 — peashooter
        sel0_hit_comb = (px >= SEL_X0 && px < SEL_X0 + SEL_SZ &&
                         py >= SEL_Y  && py < SEL_Y  + SEL_SZ);
        sel0_border_comb = 1'b0;
        if (sel0_hit_comb) begin
            if ((px - SEL_X0)           < SEL_BORDER ||
                (SEL_X0 + SEL_SZ - px) <= SEL_BORDER ||
                (py - SEL_Y)            < SEL_BORDER ||
                (SEL_Y + SEL_SZ - py)  <= SEL_BORDER)
                sel0_border_comb = 1'b1;
        end

        // Box 1 — sunflower
        sel1_hit_comb = (px >= SEL_X1 && px < SEL_X1 + SEL_SZ &&
                         py >= SEL_Y  && py < SEL_Y  + SEL_SZ);
        sel1_border_comb = 1'b0;
        if (sel1_hit_comb) begin
            if ((px - SEL_X1)           < SEL_BORDER ||
                (SEL_X1 + SEL_SZ - px) <= SEL_BORDER ||
                (py - SEL_Y)            < SEL_BORDER ||
                (SEL_Y + SEL_SZ - py)  <= SEL_BORDER)
                sel1_border_comb = 1'b1;
        end
    end

    // Sun HUD: 10 yellow blocks across the top.  Block i is lit when
    // sun_value >= (i+1)*100.  Loop is unrolled at synthesis.
    logic sun_hit_comb;
    always_comb begin
        sun_hit_comb = 1'b0;
        if (py >= SUN_Y && py < SUN_Y + SUN_BH) begin
            for (int i = 0; i < 10; i++) begin
                logic [9:0] bx;
                bx = SUN_X + 10'(i) * SUN_PITCH;
                if (sun_value >= 14'((i+1) * 50) &&
                    px >= bx && px < bx + SUN_BW)
                    sun_hit_comb = 1'b1;
            end
        end
    end

    // -----------------------------------------------------------------
    // Stage 2: register everything to align with the 1-cycle sprite
    // ROM read latency.  Then mux to produce final color.
    // -----------------------------------------------------------------
    logic [7:0] bg_color_d;
    logic       plant_here_d;
    logic       sunflower_here_d;
```

```
logic       zombie_hit_d;
logic       pea_hit_d;
logic       cursor_hit_d;
logic       sun_hit_d;
logic       sel0_hit_d, sel0_border_d;
logic       sel1_hit_d, sel1_border_d;
logic [1:0] selected_plant_d;

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        bg_color_d        <= 8'd0;
        plant_here_d      <= 1'b0;
        sunflower_here_d <= 1'b0;
        zombie_hit_d      <= 1'b0;
        pea_hit_d         <= 1'b0;
        cursor_hit_d      <= 1'b0;
        sun_hit_d         <= 1'b0;
        sel0_hit_d        <= 1'b0;
        sel0_border_d     <= 1'b0;
        sel1_hit_d        <= 1'b0;
        sel1_border_d     <= 1'b0;
        selected_plant_d <= 2'd0;
    end else begin
        bg_color_d        <= bg_color;
        plant_here_d      <= plant_here;
        sunflower_here_d <= sunflower_here;
        zombie_hit_d      <= zombie_hit_comb;
        pea_hit_d         <= pea_hit_comb;
        cursor_hit_d      <= cursor_hit_comb;
        sun_hit_d         <= sun_hit_comb;
        sel0_hit_d        <= sel0_hit_comb;
        sel0_border_d     <= sel0_border_comb;
        sel1_hit_d        <= sel1_hit_comb;
        sel1_border_d     <= sel1_border_comb;
        selected_plant_d <= selected_plant;
    end
end

// Final mux: paint layers from bottom to top.  Sprite pixels are
// valid this cycle (issued from address registered last cycle by
// the sprite ROM, which has 1-cycle latency).
always_comb begin
    color_out = bg_color_d;
    if (plant_here_d && plant_rd_pixel != COL_TRANSPARENT)
        color_out = plant_rd_pixel;
    if (sunflower_here_d && sunflower_rd_pixel != COL_TRANSPARENT)
        color_out = sunflower_rd_pixel;
    if (pea_hit_d)
        color_out = COL_BRIGHT_GREEN;
```

```
        if (zombie_hit_d && zombie_rd_pixel != COL_TRANSPARENT)
            color_out = zombie_rd_pixel;
        if (cursor_hit_d)
            color_out = COL_YELLOW;
        if (sun_hit_d)
            color_out = COL_YELLOW;
        // Selector fills: always show both plants so the player can
        // see what's on offer.  Box 0 = peashooter (green), box 1 =
        // sunflower (orange).
        if (sel0_hit_d)
            color_out = COL_GREEN;
        if (sel1_hit_d)
            color_out = COL_ORANGE;
        // Selector border: only the chosen box wears the yellow
        // outline.  TAB flips selected_plant in software, which
        // moves the border from one box to the other.
        if (sel0_border_d && selected_plant_d == 2'd0)
            color_out = COL_YELLOW;
        if (sel1_border_d && selected_plant_d == 2'd1)
            color_out = COL_YELLOW;
    end

endmodule
```

`hw/bg_grid.sv`

```
/*
 * Background grid — 8 cols x 4 rows, 64x64 px each
 *
 * Game area: x in [64, 576), y in [112, 368).  The lawn light_cell pattern
 * is hardcoded (alternating dark/light green by (row+col) parity), so
 * no CPU writes or double buffering are needed.
 *
 * Output: given pixel (px, py), returns the background color index.
 *   - in game area: dark green or light green light_cell
 *   - outside     : blue
 *
 * Cell math uses bit slices because 64 = 2^6.
 */

module bg_grid(
    input  logic [9:0] px,
    input  logic [9:0] py,
    output logic [7:0] color_out
);

    // Color indices (must match color_palette.sv)
    localparam logic [7:0] COL_BLACK       = 8'd0;
    localparam logic [7:0] COL_DARK_GREEN  = 8'd1;
    localparam logic [7:0] COL_LIGHT_GREEN = 8'd2;
    localparam logic [7:0] COL_BLUE = 8'd13;

    // Game area bounds
    localparam logic [9:0] GRID_X = 10'd64;
    localparam logic [9:0] GRID_Y = 10'd112;
    localparam logic [9:0] GRID_W = 10'd512; // 8 cols x 64
    localparam logic [9:0] GRID_H = 10'd256; // 4 rows x 64

    wire in_grid = (px >= GRID_X) && (px < GRID_X + GRID_W) &&
                   (py >= GRID_Y) && (py < GRID_Y + GRID_H);

    // Within-grid offset (full 10 bits; high bit unused inside grid).
    // Cell index = offset >> 6 because cell size is 64.
    /* verilator lint_off UNUSED */
    wire [9:0] gx = px - GRID_X;
    wire [9:0] gy = py - GRID_Y;
    /* verilator lint_on UNUSED */

    // Checker pattern: alternate by (col + row) parity.  col is gx[8:6],
    // row is gy[7:6] — only the LSB of each matters for parity.
    wire light_cell = gx[6] ^ gy[6];
```

```
    always_comb begin
        if (!in_grid)
            color_out = COL_BLUE;
        else if (light_cell)
            color_out = COL_LIGHT_GREEN;
        else
            color_out = COL_DARK_GREEN;
    end

endmodule
```

`hw/color_palette.sv`

```
/*
 * 256-entry color palette LUT: 8-bit index -> 24-bit RGB
 *
 * Hardcoded with 13 MVP colors. Remaining entries default to black.
 *
 * Index  Color          R    G    B    Usage
 * -----  -----------   ---  ---  ---  -----
 *   0    Black          00   00   00   Unused
 *   1    Dark Green     1B   5E   20   Grid cell (dark)
 *   2    Light Green    2D   8B   2D   Grid cell (light)
 *   3    Brown          8B   45   13   Soil / stem
 *   4    Yellow         FF   D7   00   Cursor highlight
 *   5    Red            FF   00   00   Zombie body
 *   6    Dark Red       8B   00   00   Zombie head
 *   7    Green          00   80   00   Peashooter body
 *   8    Dark Green2    00   64   00   Peashooter stem
 *   9    Bright Green   00   FF   00   Pea projectile
 *  10    White          FF   FF   FF   HUD digits
 *  11    Gray           80   80   80   HUD background
 *  12    Orange         FF   A5   00   Sun indicator
 *  13    Sky Blue       87   CE   EB   Background
 */

module color_palette(
    input  logic [7:0]  index,
    output logic [7:0]  r,
    output logic [7:0]  g,
    output logic [7:0]  b
);

    always_comb begin
        case (index)
            8'd0:  {r, g, b} = {8'h00, 8'h00, 8'h00}; // Black
            8'd1:  {r, g, b} = {8'h1B, 8'h5E, 8'h20}; // Dark Green
            8'd2:  {r, g, b} = {8'h2D, 8'h8B, 8'h2D}; // Light Green
            8'd3:  {r, g, b} = {8'h8B, 8'h45, 8'h13}; // Brown
            8'd4:  {r, g, b} = {8'hFF, 8'hD7, 8'h00}; // Yellow
            8'd5:  {r, g, b} = {8'hFF, 8'h00, 8'h00}; // Red
            8'd6:  {r, g, b} = {8'h8B, 8'h00, 8'h00}; // Dark Red
            8'd7:  {r, g, b} = {8'h00, 8'h80, 8'h00}; // Green
            8'd8:  {r, g, b} = {8'h00, 8'h64, 8'h00}; // Dark Green 2
            8'd9:  {r, g, b} = {8'h00, 8'hFF, 8'h00}; // Bright Green
            8'd10: {r, g, b} = {8'hFF, 8'hFF, 8'hFF}; // White
            8'd11: {r, g, b} = {8'h80, 8'h80, 8'h80}; // Gray
            8'd12: {r, g, b} = {8'hFF, 8'hA5, 8'h00}; // Orange
            8'd13: {r, g, b} = {8'h87, 8'hCE, 8'hEB}; // Sky Blue
```

```
            default: {r, g, b} = {8'h00, 8'h00, 8'h00}; // Black
        endcase
    end

endmodule
```

hw/sprite_rom.sv

```
/*
 * Sprite ROM — 64x64 palette-indexed sprite
 *
 * Stores 4096 bytes loaded from MEM_FILE at synthesis / simulation.
 * Each byte is:
 *   0x00-0x0C : palette index (matches color_palette.sv)
 *   0xFF      : transparent (renderer must skip write)
 *
 * Read latency: 1 clock (inferred M10K block RAM).
 */
module sprite_rom #(
    parameter MEM_FILE = "peashooter_idx.mem"
) (
    input  logic        clk,
    input  logic [11:0] addr,   // 0..4095 = y*64 + x
    output logic [7:0]  pixel
);

    logic [7:0] rom [0:4095];

    initial begin
        $readmemh(MEM_FILE, rom);
    end

    always_ff @(posedge clk)
        pixel <= rom[addr];

endmodule
```

`hw/pvz_top_hw.tcl`

```
# pvz_top_hw.tcl — Platform Designer component descriptor for pvz_top
#
# Defines the Avalon-MM slave interface so Platform Designer can
# generate the bus fabric and device tree entries automatically.

package require -exact qsys 16.0

# Module properties
set_module_property DESCRIPTION "PvZ GPU — VGA shape-based display engine"
set_module_property NAME pvz_top
set_module_property VERSION 1.0
set_module_property INTERNAL false
set_module_property OPAQUE_ADDRESS_MAP true
set_module_property AUTHOR "CSEE4840 Team"
set_module_property DISPLAY_NAME "PvZ GPU"
set_module_property INSTANTIATE_IN_SYSTEM_MODULE true
set_module_property EDITABLE true
set_module_property REPORT_TO_TALKBACK false
set_module_property ALLOW_GREYBOX_GENERATION false
set_module_property REPORT_HIERARCHY false

# Device tree compatible string — must match kernel driver's of_device_id
set_module_assignment embeddedsw.dts.compatible "csee4840,pvz_gpu-1.0"
set_module_assignment embeddedsw.dts.group "pvz_gpu"
set_module_assignment embeddedsw.dts.vendor "csee4840"

# Source files
add_fileset QUARTUS_SYNTH QUARTUS_SYNTH "" ""
set_fileset_property QUARTUS_SYNTH TOP_LEVEL pvz_top
add_fileset_file pvz_top.sv SYSTEM_VERILOG PATH pvz_top.sv TOP_LEVEL_FILE
add_fileset_file vga_counters.sv SYSTEM_VERILOG PATH vga_counters.sv
add_fileset_file bg_grid.sv SYSTEM_VERILOG PATH bg_grid.sv
add_fileset_file entity_drawer.sv SYSTEM_VERILOG PATH entity_drawer.sv
add_fileset_file color_palette.sv SYSTEM_VERILOG PATH color_palette.sv
add_fileset_file sprite_rom.sv SYSTEM_VERILOG PATH sprite_rom.sv
add_fileset_file peashooter_idx.mem OTHER PATH peashooter_idx.mem
add_fileset_file zombie_idx.mem OTHER PATH zombie_idx.mem

# Clock interface
add_interface clock clock end
set_interface_property clock clockRate 0
add_interface_port clock clk clk Input 1

# Reset interface
add_interface reset reset end
set_interface_property reset associatedClock clock
```

```
set_interface_property reset synchronousEdges DEASSERT
add_interface_port reset reset reset Input 1

# Avalon-MM slave interface
add_interface s1 avalon end
set_interface_property s1 addressUnits WORDS
set_interface_property s1 associatedClock clock
set_interface_property s1 associatedReset reset
set_interface_property s1 bitsPerSymbol 8
set_interface_property s1 burstOnBurstBoundariesOnly false
set_interface_property s1 burstcountUnits WORDS
set_interface_property s1 explicitAddressSpan 0
set_interface_property s1 holdTime 0
set_interface_property s1 linewrapBursts false
set_interface_property s1 maximumPendingReadTransactions 0
set_interface_property s1 maximumPendingWriteTransactions 0
set_interface_property s1 readLatency 0
set_interface_property s1 readWaitTime 1
set_interface_property s1 setupTime 0
set_interface_property s1 timingUnits Cycles
set_interface_property s1 writeWaitTime 0

add_interface_port s1 address address Input 6
add_interface_port s1 writedata writedata Input 32
add_interface_port s1 write write Input 1
add_interface_port s1 chipselect chipselect Input 1

# VGA conduit interface (directly exported to top level)
add_interface vga conduit end
set_interface_property vga associatedClock clock
set_interface_property vga associatedReset reset

add_interface_port vga VGA_R r Output 8
add_interface_port vga VGA_G g Output 8
add_interface_port vga VGA_B b Output 8
add_interface_port vga VGA_CLK clk Output 1
add_interface_port vga VGA_HS hs Output 1
add_interface_port vga VGA_VS vs Output 1
add_interface_port vga VGA_BLANK_n blank_n Output 1
add_interface_port vga VGA_SYNC_n sync_n Output 1
```

`hw/soc_system_top.sv` (project-specific slice)

The rest of this file is lab3 boilerplate (pin declarations, HPS DDR3/Ethernet/USB/SPI/UART/I2C/GPIO wiring, "quiet warnings" tie-offs). The only project-specific addition is the PvZ GPU VGA conduit wiring inside the `soc_system soc_system0(...)` instantiation:

```
        // PvZ GPU VGA conduit (exported from Platform Designer)
        .pvz_top_0_vga_r              ( VGA_R ),
        .pvz_top_0_vga_g              ( VGA_G ),
        .pvz_top_0_vga_b              ( VGA_B ),
        .pvz_top_0_vga_clk            ( VGA_CLK ),
        .pvz_top_0_vga_hs             ( VGA_HS ),
        .pvz_top_0_vga_vs             ( VGA_VS ),
        .pvz_top_0_vga_blank_n        ( VGA_BLANK_N ),
        .pvz_top_0_vga_sync_n         ( VGA_SYNC_N )
```