No Man's Land

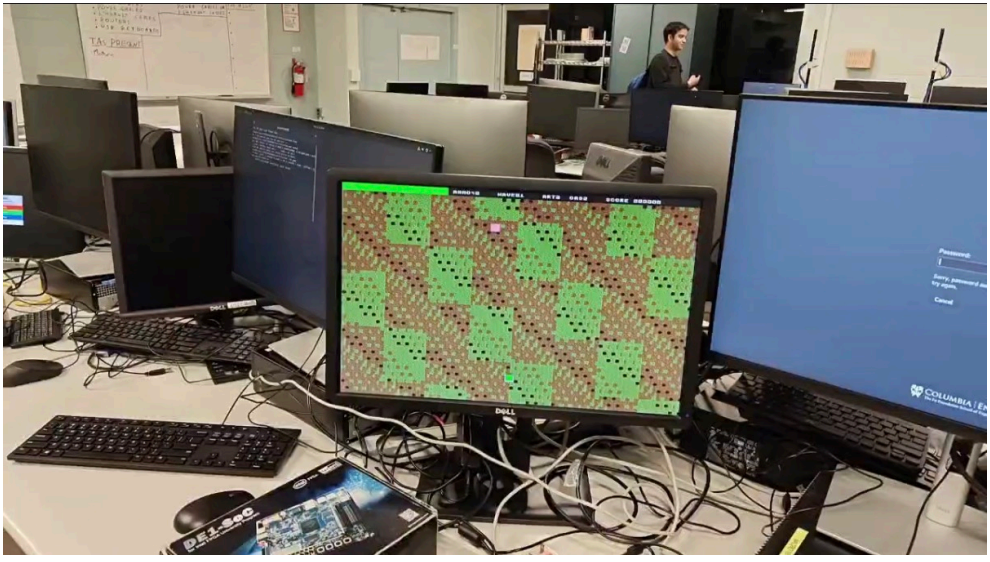
A custom SystemVerilog GPU and game on the DE1-SoC

Rohit Biswas Kambinachi Obioha Nicola Paparella

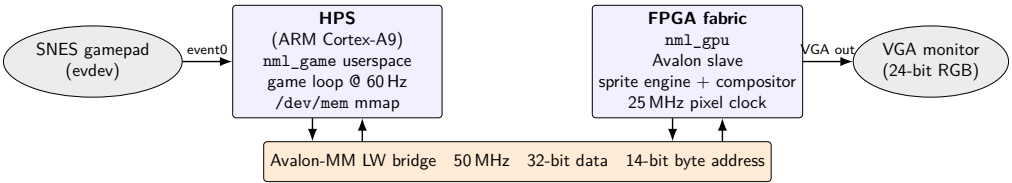
CSEE 4840 — Embedded System Design — Spring 2026

Instructor: Prof. Stephen Edwards

The system, running

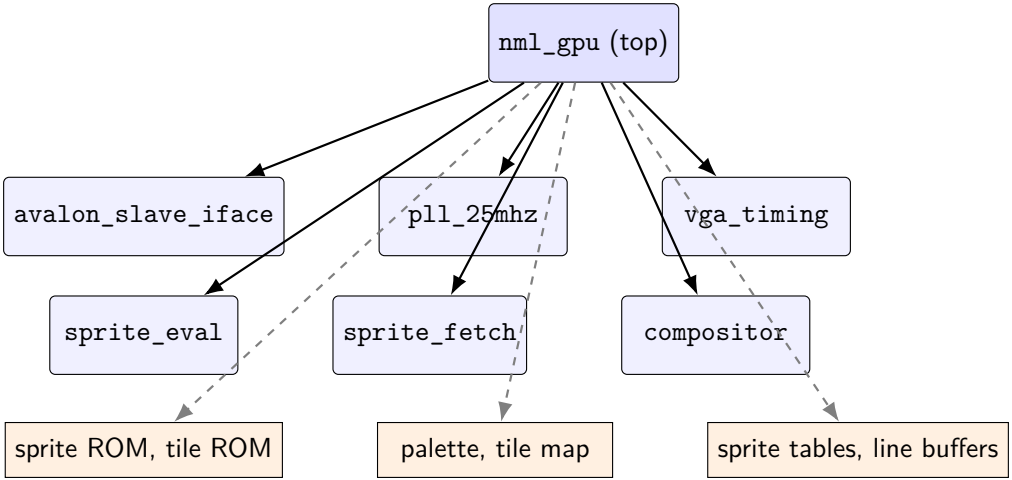


System block



HPS writes shadow state; FPGA latches at VSYNC; pixels always stream.

What's inside `nml_gpu`

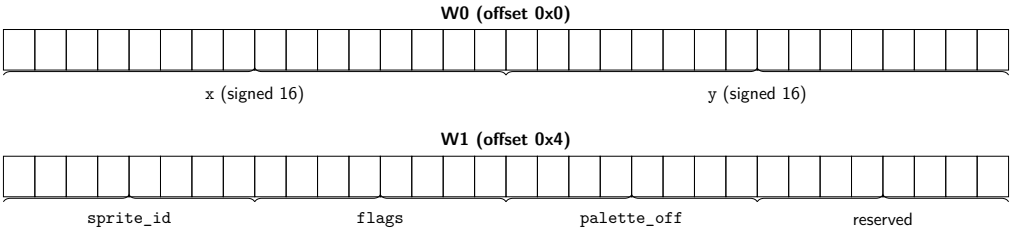


One 50 MHz domain (Avalon) + one 25 MHz domain (pixel). All RAMs/ROMs inferred as M10K.

Avalon register map — 16 KB window at 0xFF200000

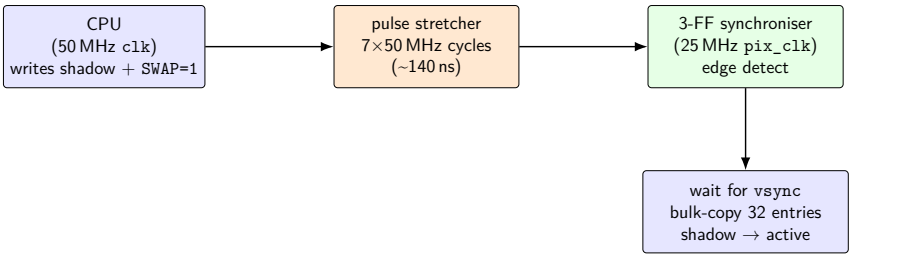
Offset	Name	Layout
0x0000	CTRL	[0]=ENABLE, [1]=SWAP (auto-clear), [2]=HUD_ON
0x0004	STATUS	[0]=VBLANK, [1]=SWAP_PENDING
0x0008	BG_SCROLL	[15:0]=scroll_x, [31:16]=scroll_y (signed)
0x000C	IRQ_MASK	[0]=VBLANK IRQ enable
0x0010	PLAYER_POS	[15:0]=px, [31:16]=py
0x0014	PLAYER_STATS	[7:0]=hp, [15:8]=wave (BCD), [31:16]=level
0x0018	SCORE	[23:0]=6-digit BCD score
0x001C	KILL_COUNT	32-bit kills
0x0020	HUD_AUX	[7:0]=ammo BCD, [11:8]=art, [15:12]=gas
0x0100–0x01FF	**Sprite table**	32 entries × 8 B
0x0400–0x07FF	**Palette**	256 entries × 4 B (RGB888)
0x1000–0x22BF	**Tile map**	80 cols × 60 rows, 1 byte/tile

Sprite-table entry — 8 bytes / 64 bits



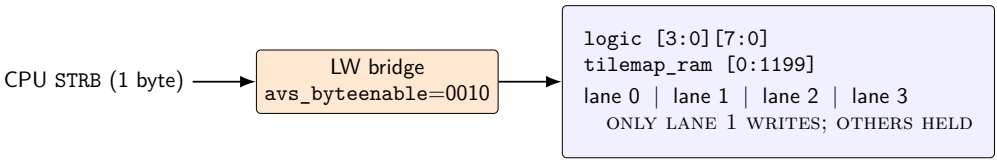
flags byte: [2] hflip [3] vflip [6:4] priority (0 = highest) [7] ACTIVE

Frame commit — shadow → active at VSYNC



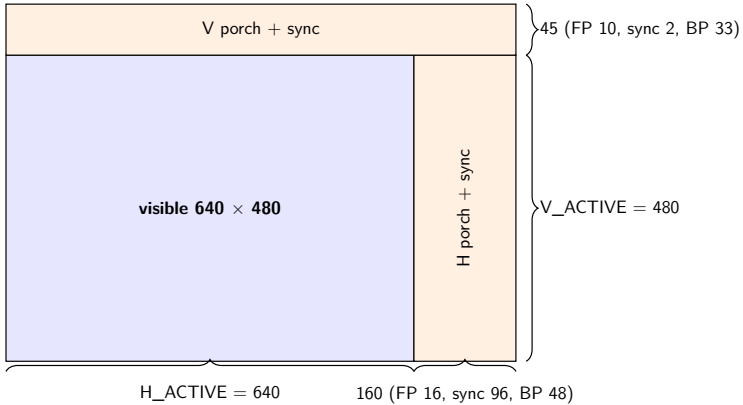
One 20 ns Avalon pulse would slip past the 40 ns pixel clock. Widen to 140 ns and the CDC is deterministic.

Tile-map byte writes — packed [3:0] [7:0]



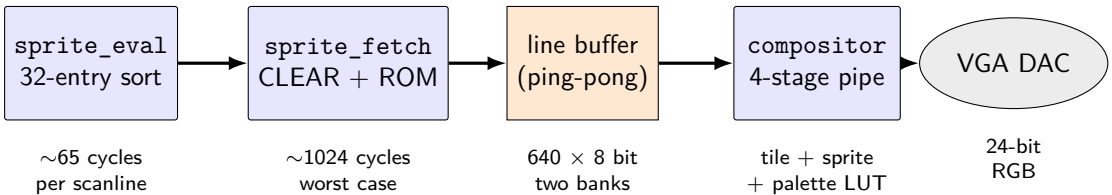
Why packed? Byte-organised storage forced bridge RMW to clobber 3 lanes per word. Packed inference uses M10K byteena_a: one lane updates, the other three are untouched.

VGA timing — 640×480 @ 60 Hz, 25 MHz dot



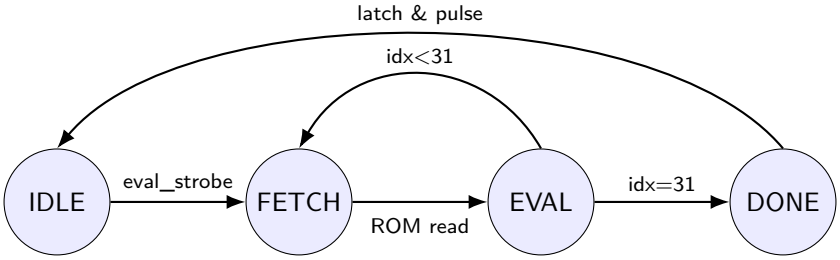
One frame = 800 × 525 dot cycles. HBLANK is the 160-cycle window we use to evaluate sprites for the next scanline.

Pixel pipeline — sprite engine + compositor



Eval runs inside HBLANK (160 cycles). Fetch spans HBLANK + visible (800 cycles).
Line buffers are double-buffered, so fetch lags read by one scanline. Compositor never stalls.

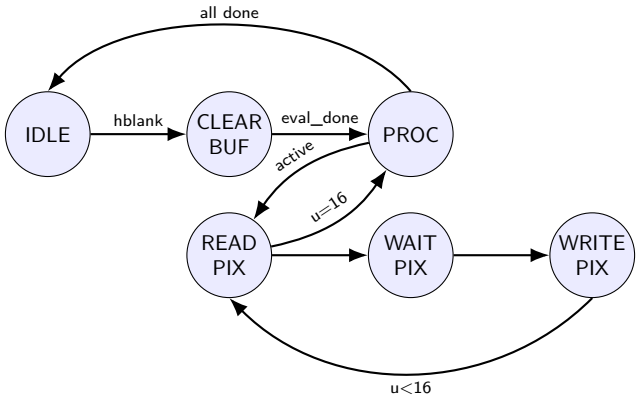
`sprite_eval` — per-scanline priority sort



Per sprite (1 cycle in EVAL):
`intersects = active && y`$\leq$`next_scanline < y+16`
insertion-sort into 8 priority-ordered slots (all 8 compares in parallel)

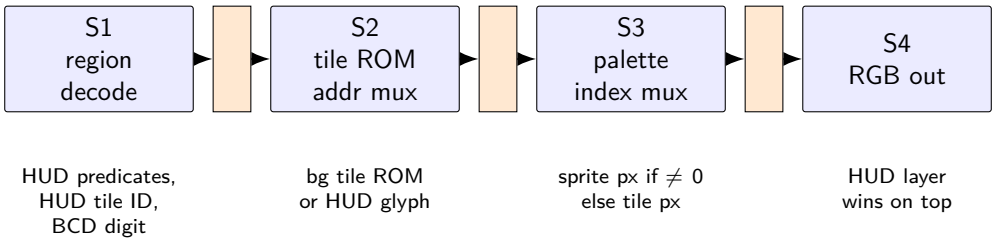
Total: 32 fetch+eval cycles + 1 DONE = $\sim$65 cycles. Fits easily in 160-cycle HBLANK.

`sprite_fetch` — ROM read + line-buffer fill



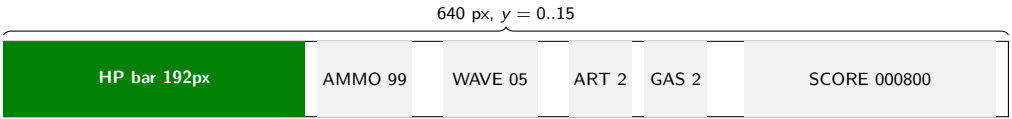
Worst-case cycle budget per scanline: CLEAR_BUF (640) + 8 sprites × 48 (384) =∼1024 cycles. HBLANK alone is 160; HBLANK + visible is **800**. Fetch lives one line behind the read, so the ping-pong line buffers absorb the latency.

Compositor — 4-stage pipeline



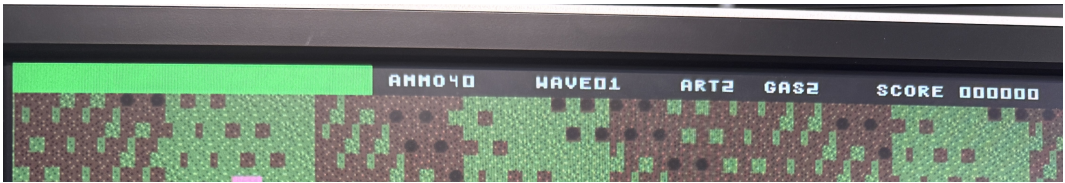
Latches between stages absorb the 1-cycle latency of tile ROM and palette M10K reads. Never stalls.

HUD strip — top 16 pixels of every frame



HP bar: 2 px per HP, clamped to 192. Glyphs: 8×8 from tile ROM, slots 16–41 (A–Z), 48–57 (0–9). Scores, ammo, wave packed BCD in registers.

HUD — on the monitor



Memory inventory (all inferred M10K)

Memory	Shape	Init	Bits
sprite ROM	16,384 × 8	`$readmemh`	131,072
tile ROM	4,096 × 8	`$readmemh`	32,768
tile map	1,200 × 32 (packed 4 × 8)	blank (HPS writes)	38,400
palette	256 × 24	`$readmemh` + HPS overrides	6,144
sprite table act.	32 × 64	via VSYNC bulk copy	2,048
sprite table shd.	32 × 64	HPS writes	2,048
line buffer A	640 × 8	cleared per HBLANK	5,120
line buffer B	640 × 8	cleared per HBLANK	5,120
Total			**86,144 bits**

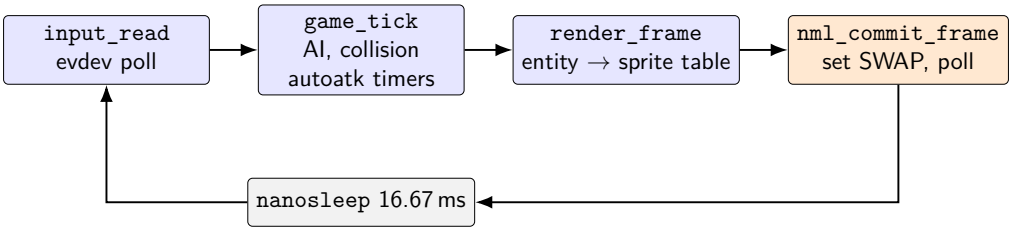
14 M10K blocks out of 397.

Resource utilisation — Cyclone V 5CSEMA5F31C6

Resource	Used	Available	%
ALMs	205	32,070	$< 1\%$
Registers	365	—	—
Pins	54	457	12%
Block memory bits	86,144	4,065,280	2%
M10K blocks	14	397	4%
DSP blocks	0	87	0%

Compile time: 63 s wall clock (Analysis 13 s, Fitter 30 s, Asm 12 s, STA 8 s).

Software architecture — 60 Hz game loop on the ARM



`game_t` holds a 64-entity pool. State machine: PLAYING ↔ LEVELUP → GAMEOVER. Mortar is timer-driven; artillery + gas fire from L/R shoulders.

What actually works

640×480 @ 60 Hz • tile-mapped battlefield • 32 hardware sprites • 256-entry palette
HUD: HP bar, AMMO, WAVE, ART, GAS, SCORE (BCD)
SNES gamepad via evdev; 60 Hz HPS game loop; frame-commit at VSYNC
Mortar, artillery, gas clouds, wave spawner, level-up, ammo drops, game-over

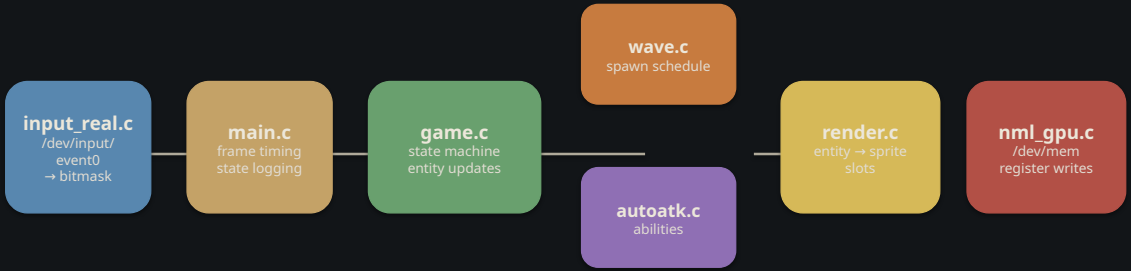
No Man’s Land

Program Deep Dive: from controller input to VGA sprite table

Focus: software program internals, data flow, and hardware interface

What the software actually does each frame

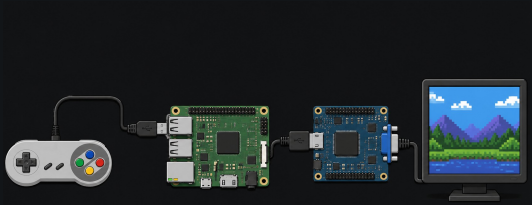
File-level responsibilities with the real call path



The main separation is intentional: gameplay logic never knows whether output is terminal text or the real FPGA.

Two builds, one game core

The same game_t and game_tick() are reused



Terminal build
render_terminal.c
prints sprite slots

FPGA build
render.c + nml_gpu.c
writes registers

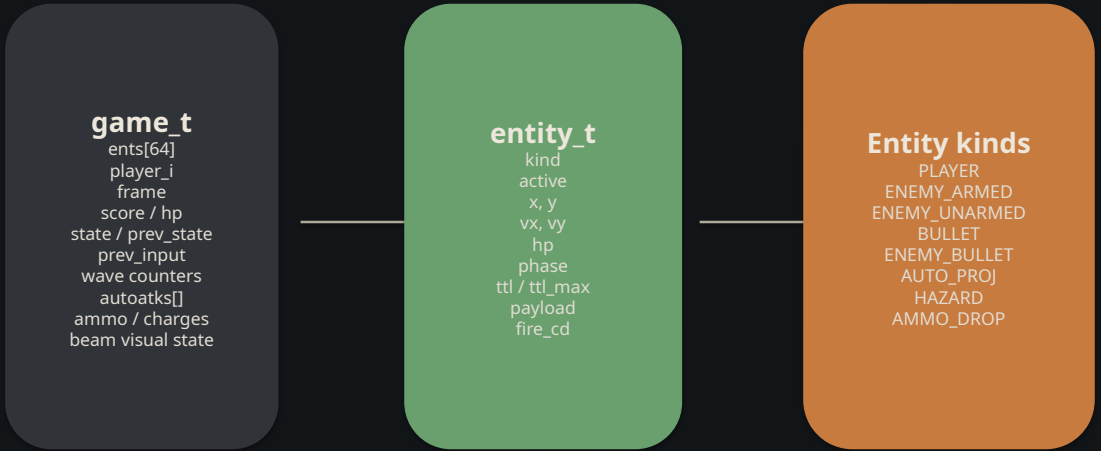
Shared core
main.c → input_read() → game_tick() → render_frame()

```
uint16_t input = input_read(game.frame);
       game_tick(&game, input);
         render_frame(&game);
```

Build-time choice: link render_terminal.c or render.c

The game state is just structured memory

game_t owns every object and counter



Core invariant: inactive entity slots are available for reuse

main.c controls timing, not game rules

The loop handles hardware init, signals, frame duration, and debug prints

```
if (!NML_TERMINAL_BUILD) {
        nml_open();
  init_palette_runtime();
   render_init_tilemap();
     nml_set_enable(1);
             }

   while (g_running) {
 input = input_read(frame);
  game_tick(&game, input);
    render_frame(&game);
             }
```

startup
FPGA path maps /dev/mem and paints palette/tilemap

per-frame
input → tick → render

transitions
prints wave clear, level-up, game-over logs

shutdown
SIGINT/SIGTERM turns off video cleanly

main.c stays thin so gameplay remains testable

input_real.c turns Linux events into a bitmask

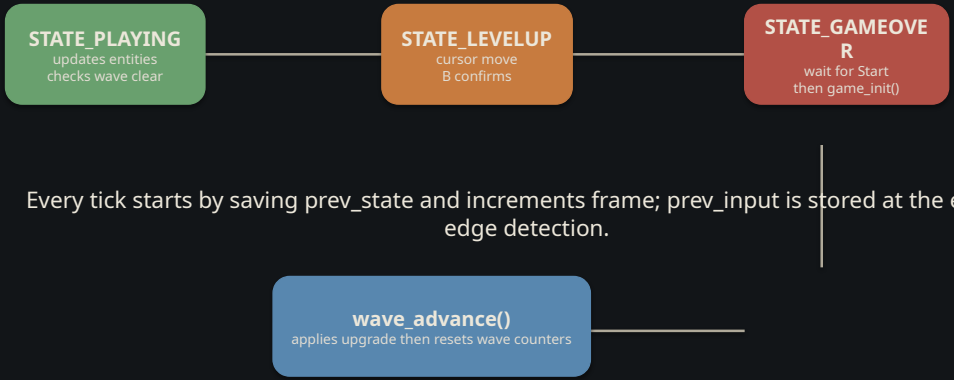
No game logic depends on Linux input details

Control	Linux code	Game bit
D-pad X/Y	ABS_X / ABS_Y	LEFT / RIGHT / UP / DOWN
B / Y / X / A	BTN_THUMB2 / TOP / TRIGGER / THUMB	fire down / left / up / right
L / R	BTN_TOP2 / BTN_PINKIE	artillery / gas
Start	BTN_BASE4	restart / confirm path

Debug mode: NML_INPUT_DEBUG=1 prints raw events for remapping

game_tick() is a state machine

Most bugs are easier to locate once the state transition is clear



game_tick(): routes PLAYING vs LEVELUP vs GAMEOVER behavior

Player update: movement, clamps, firing, abilities

update_player() consumes the bitmask produced by input_real.c

Move
D-pad adds ±4 px/frame to x/y

Clamp
x: 0..624
y: HUD_H..464

Fire
face buttons choose axis-aligned bullet

Abilities
L: artillery
R: gas cloud

```
  if (INPUT_FIRE_DOWN)  bullet( 0, +8);
else if (INPUT_FIRE_LEFT)  bullet(-8, 0);
else if (INPUT_FIRE_UP)    bullet( 0,-8);
else if (INPUT_FIRE_RIGHT) bullet(+8, 0);
```

```
if (fire_cooldown != 0) return;
    if (ammo <= 0) return;
      spawn ENT_BULLET;
     fire_cooldown = 5;
          ammo--;
```

Cooldown prevents multi-button firing from spawning four bullets in one frame

Enemy AI is simple but intentionally de-synchronized

Armed enemies add fire behavior on top of movement

Vertical pressure
e->y steps toward player row at wave speed

Horizontal choice
25% left, 25% right, 50% chase player

Armed firing
cooldown counts down then aims once

```
r = enemy_rng(frame, phase)
  if r < 64:   x -= speed
  elif r <128: x += speed
  else:        x → player.x
```

```
dx = player.x - enemy.x
dy = player.y - enemy.y
   norm = |dx| + |dy|
 v = 3 * (dx,dy) / norm
```

```
    phase = rand() & 0xff
 fire_cd = 60 + rand()%120
// avoids synchronized waves
```

Enemy bullets are not homing; they lock direction at spawn time

Motion updates are separate from collision handling

Each entity type has one simple update rule

Player bullet
x += vx, y += vy
despawn off-screen

Enemy bullet
same motion path
damages player later

Ammo drop
ttl-- until 0
collected by AABB

Beam
beam_ttl controls
tile column

This separation keeps behavior predictable: movement first, then a single collision pass resolves consequences.

```
ate_enemies(); update_bullets(); update_enemy_bullets(); update_ammo_drops(); autoatk_tick(); autoatk_update_proj(); handle_collisions
```

No collision is applied during the motion loops

Collision handling is ordered by gameplay meaning

handle_collisions() owns damage, pickups, and score bookkeeping

1	Ammo drop ↔ player	ammo += 10, cap at 99, drop disappears
2	Enemy bullet ↔ player	player HP −10, bullet disappears
3	Enemy ↔ player	player HP −10, enemy disappears, no score
4	Player bullet ↔ enemy	bullet disappears, apply_damage(enemy, 1)
5	Auto hazard/projectile ↔ enemy	damage + optional despawn policy by weapon kind

All enemy kills route through apply_damage() so score, kill counters, and ammo drops stay consistent.

AABB is 16×16 for normal entities; gas uses size-aware overlap

wave.c is a data-driven spawner

Difficulty lives in the WAVES table

Wave	period	total	armed	armed HP	speed
1	60	6	0	1	1
2	50	8	1	2	1
3	45	10	2	2	2
4	35	12	4	2	2
5+	25	16	6	3	3

wave_tick()
spawn when cooldown hits zero

alloc_entity()
find inactive entity slot

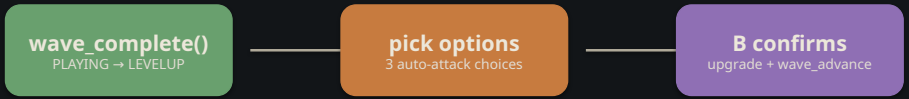
initialize enemy
kind, hp, phase, fire_cd

wave_complete()
spawned all + none alive

Wave 5 clamps and repeats as the hardest pattern

Level-up pauses the simulation before advancing the wave

The upgrade menu is still part of game_tick()




```
if LEVELUP:
LEFT/RIGHT moves cursor
  B confirms option
autoatk_upgrade(kind)
```

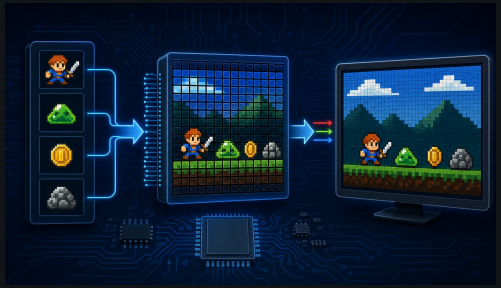
```
wave_advance():
  wave_index++
reset spawn counters
refill ammo/charges
```

Important detail: wave_advance() happens after the player picks an upgrade, not immediately when the wave ends.

Level-up is not a separate program; it is a game state

render.c converts entities into hardware sprite slots

This is the bridge from gameplay state to VGA-visible objects.



slot 0
always player

slots 1–31
entities in array order

gas hazard
expands into 1–4 sprites

leftover slots
nml_hide_sprite()

```
emit_sprite(&slot, e->x, e->y,
            entity_to_sprite_id(e),
            entity_palette_off(e),
            priority);
```

Mock renderer mirrors this flow but prints slot contents instead of writing registers

nml_gpu.c packs C structures into Avalon register writes

The program writes a 16 KB LW bridge window mapped at 0xFF200000

Region	Offset	Write meaning
CTRL	0x0000	ENABLE / SWAP / HUD_ON
PLAYER_POS	0x0010	x and y packed into one word
PLAYER_STATS	0x0014	hp + BCD wave + level
SCORE	0x0018	HUD score
HUD_AUX	0x0020	ammo + artillery + gas charges
SPRITE_TABLE	0x0100	32 × 8-byte sprite entries
PALETTE	0x0400	RGB888 entries
TILEMAP	0x1000	80 × 60 tile IDs

nml_open()
open("/dev/mem") → mmap(NML_LWFPGA_BASE)

nml_write_sprite()
W0 = y:x, W1 = sprite_id:flags:palette_off

nml_commit_frame()
sets CTRL.SWAP so hardware commits at VSYNC

Register packing matches nml_gpu.h and the design document

One sprite slot is two 32-bit words

This is what render.c ultimately writes per visible object

W0: position
bits 15:0 = x bits 31:16 = y

W1: identity + draw flags
sprite_id, flip bits, priority, active, palette offset

Field	Purpose
sprite_id	selects 16×16 art in sprite ROM
ACTIVE	visible / hidden
priority	scanline culling order
hflip / vflip	future sprite mirroring
palette_off	recoloring hook

```
w0 = ((uint32_t)(uint16_t)y << 16) | (uint16_t)x;
w1 = sprite_id | (flags << 8) | (palette_off << 16);
```

render.c never writes pixels; it writes sprite descriptors for the FPGA compositor

render_terminal.c is a diagnostic mirror of render.c

It helps verify slot allocation before touching the FPGA

mock_sprite_t
active, x, y, sprite_id, glyph

entity_glyph()
P, A, u, *, ^, +, M, ~

slot printout
frame, hp, score, wave, ammo, 32 slots

```
frame=150 hp=100 score=0 sprite_count=3
 slot 00: sprite_id=1 x=320 y=420 glyph=P
 slot 01: sprite_id=2 x=320 y=115 glyph=A
 slot 02: sprite_id=4 x=53  y=55  glyph=u
            slot 03: inactive
```

If terminal slots look wrong, FPGA output will also be wrong

Where to debug when something goes wrong

Symptoms map to specific files

Symptom	First place to look
controller moves nothing	input_real.c: event device, mapping, permissions
player moves but cannot fire	update_player(): ammo + fire_cooldown
enemies sync into columns	phase / enemy_rng() in update_enemies()
wave never ends	wave_complete(): spawned count + active enemies
sprites ghost on screen	render.c: hide leftover slots
FPGA shows blank	nml_open(), CTRL.ENABLE, palette/tilemap init
HUD wrong	nml_set_player_state(), nml_set_score(), HUD_AUX

Testing strategy: verify terminal build first, then the hardware writer