No Man's Land

Design Document
CSEE 4840 – Embedded Systems, Spring 2026

Rohit Biswas (rb3908) Kambinachi Obioha (kno2117) Nicola Paparella (np2953)

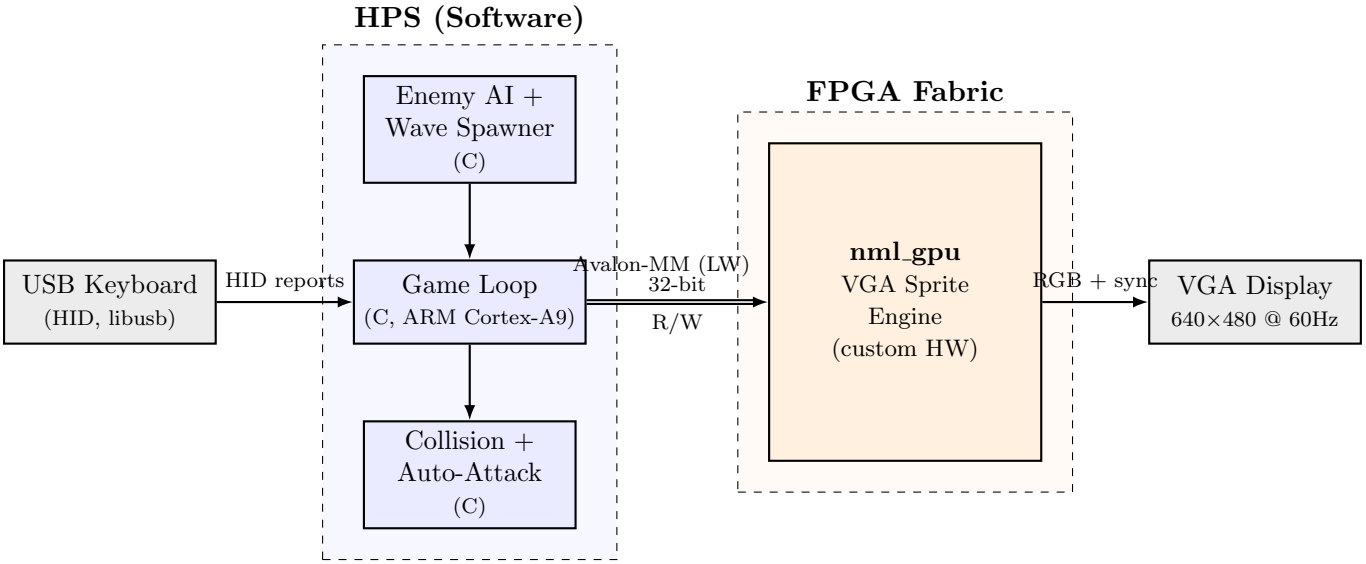
Instructor: Prof. Stephen Edwards
April 16, 2026

1 Project Summary

No Man's Land is a top-down WWI horde survival game on the DE1-SoC. WASD moves the player, arrow keys aim and fire (USB keyboard via libusb). Enemy waves spawn from screen edges and chase the player; each surviving wave grants an auto-attack item (mortar, barbed wire, mustard gas, artillery). A custom SystemVerilog peripheral (`nml_gpu`) handles VGA timing, tile-mapped background, and sprite compositing. Game logic, AI, collision, and auto-attack scheduling run in C on the ARM Cortex-A9 (HPS).

The thematic hook (waves shift toward unarmed figures; auto-attacks have no target filter; end-of-run kill breakdown) lives entirely in software wave tables and the end screen. Hardware contract is unchanged.

2 Top-Level Block Diagram



The HPS owns all game state and writes sprite lists, background state, and HUD fields to `nml_gpu` each frame over the Avalon-MM lightweight bus. `nml_gpu` owns all video timing and drives the DE1-SoC VGA DAC directly. USB keyboard is read on the HPS with libusb, as in lab 2.

2.1 Connections in Detail

- **HPS ↔ nml_gpu (Avalon-MM Lightweight Bridge):** 32-bit data, 14-bit byte address window (16 KB region). Single-master (HPS), single-slave (nml_gpu). All transactions are single-cycle reads/writes; no burst, no waitrequest. The peripheral is fully synchronous to the 50 MHz Avalon clock.
- **nml_gpu → VGA DAC:** 24-bit RGB (8-8-8), HSYNC, VSYNC, BLANK, pixel clock (25.175 MHz nominal, generated by an on-chip PLL inside `nml_gpu`).
- **USB Keyboard → HPS:** Standard USB 2.0 HID boot keyboard protocol over the HPS USB host port. Detail in Section 3.

3 USB Keyboard Protocol

USB HID Boot Keyboard protocol (as in lab 2): the keyboard enumerates under Linux on the HPS; we open it via libusb and read 8-byte interrupt-IN reports.

3.1 Boot Keyboard Report Format

Each report is 8 bytes:

Byte	Field	Meaning
0	Modifier mask	bit0 LCtrl, bit1 LShift, bit2 LAlt, bit3 LGUI, bits4–7 right-side mirrors
1	Reserved	always 0
2–7	Keycode[0..5]	up to six simultaneously pressed HID usage codes (0 = empty slot)

Held = code in current report. *Pressed this frame* = in current, not in previous. *Released this frame* = the inverse.

3.2 Keys We Use

Key	HID Usage Code (hex)	Game Action
W / A / S / D	0x1A / 0x04 / 0x16 / 0x07	Move N / W / S / E
↑ / ↓ / ← / →	0x52 / 0x51 / 0x50 / 0x4F	Aim direction (8-way via combinations)
Spacebar	0x2C	Manual fire
Enter	0x28	Menu confirm (item selection, restart)
1 / 2 / 3	0x1E / 0x1F / 0x20	Pick item slot 1/2/3 on level-up
Esc	0x29	Pause / quit

3.3 N-Key Rollover and Reader Thread

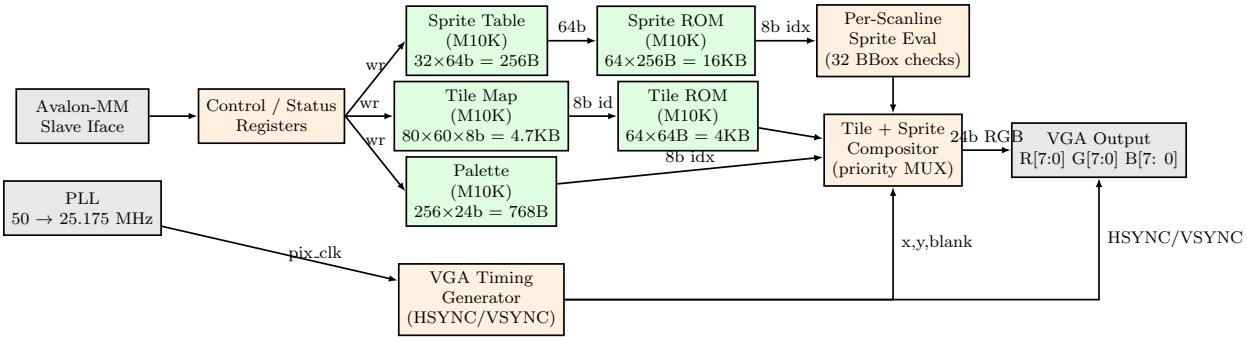
Boot protocol caps simultaneous keys at 6, enough for 2 movement + 2 aim + fire + 1 menu. On `ErrorRollOver` (`0x01` in all six slots), hold the previous frame's input; cap the hold at 3 consecutive frames, then zero.

The libusb reader runs on a pthread with two 16-bit `input_bitmask` buffers and an atomic `active_idx`. Each report decodes into the inactive buffer, then an atomic store flips the index. `usb_kbd_snapshot()` atomically loads `active_idx` and returns the corresponding buffer. Lock-free, no mutex in the hot path.

4 Custom Peripheral: `nml_gpu`

`nml_gpu` accepts writes from the HPS and renders a 640×480@60 Hz frame each refresh by compositing a tile-mapped background with up to 32 sprites.

4.1 Internal Block Diagram



4.2 Memory Inventory

All memories use Cyclone V M10K block RAM (10 Kbit each); total is well within DE1-SoC budget.

Memory	Width	Depth	Total	Access
Sprite Table	64 b	32	256 B	SW write, HW read (line buffer)
Tile Map	8 b	4800	4.7 KB	SW write, HW read (per pixel)
Palette	24 b	256	768 B	SW write, HW read (per pixel)
Sprite ROM	8 b	16384	16 KB	HW read only (`$readmemh`)
Tile ROM	8 b	4096	4 KB	HW read only (`$readmemh`)
Line Buffer A	24 b	640	1.9 KB	Internal compositor, double buffered
Line Buffer B	24 b	640	1.9 KB	Internal compositor, double buffered
Total			**≈ 29 KB**	

Widths: data paths are 8-bit (palette index, tile ID, sprite pixel) or 24-bit (RGB888). Avalon data is 32-bit. Sprite table entries are 64b packed (Section 5.3); the compositor reads one entry per cycle.

4.3 Rendering Pipeline (Per Pixel)

Line-buffer architecture: one line buffer is filled by the sprite fetch unit while the other is read out to the VGA DAC. Per scanline:

1. **Timing:** `vga_timing` produces (`x`, `y`, `blank`, `hsync`, `vsync`).
2. **Sprite eval** (during HBLANK preceding scanline N+1; for y=0, during the final HBLANK of VBLANK): scan the 32-entry sprite table, emit up to 8 whose bounding box intersects N+1, dropping lowest-priority-first.
3. **Sprite fetch** (still during HBLANK): walk the 8 candidates in ascending priority order. For each, read up to 16 palette-index bytes from Sprite ROM and write non-transparent bytes into the fill buffer at the sprite's on-screen x offset. Higher-priority sprites overwrite lower. Worst case $8 \times 16 = 128$ Sprite-ROM reads against a 160-pixel HBLANK + 640-pixel fill-idle budget.

4. **Visible scanline** N+1, per pixel:

   (a) Tile map lookup: (x>>3, y>>3) → tile ID (8b).

   (b) Tile ROM lookup: `tile_id, x[2:0], y[2:0]` → palette index (8b).

   (c) Read fill buffer at column x: sprite palette index (0 = transparent).

   (d) Priority MUX: non-transparent sprite pixel wins; otherwise tile pixel.

   (e) Palette lookup: 8b index → 24b RGB → VGA DAC.

5. At HSYNC the two line buffers swap roles; the new fill buffer is cleared to 0 during its fill phase.

Only one Sprite ROM read per pixel clock is needed, matching the M10K dual-port limit; no ROM replication required.
Handshake: the compositor never stalls (VGA must be deterministic). The sprite table is double-buffered: SW writes shadow, sets `CTRL.SWAP=1`, and the swap commits at next VSYNC with `STATUS.VBLANK` pulsing. Everything else is fire-and-forget.

4.4 HUD Overlay

Fixed overlay gated by `CTRL.HUD_ON=1`, reads only the Player State registers (Section 5.2). Occupies a reserved 16-pixel strip at the top of the screen ($y \in [0, 15]$):

- **HP bar** – cols 0–191, red/green gradient proportional to `hp`/100.
- **Wave** – cols 256–383, "WAVE N" from a 5×7 font in Tile ROM slots 56–63.
- **Score** – cols 448–639, 6-digit decimal of `SCORE`, same font.

HUD is a fourth input to the priority MUX, ranked above all sprites; does not count against the 8-sprite/line cap. Wave spawn logic confines game action to $y \geq 24$ so sprites do not compete with the strip.

5 Register Map

`nml_gpu` occupies a 16 KB window on the HPS-to-FPGA Lightweight bridge. Base offsets:

Region	Offset	Size	Description
Control/Status	`0x0000`	16 B	4 control/status registers (see below)
Player State	`0x0010`	16 B	player x, y, hp, score (consumed by HUD overlay)
Sprite Table	`0x0100`	256 B	32 entries × 8 B
Palette	`0x0400`	1 KB	256 entries × 4 B (RGB888 + pad)
Tile Map	`0x1000`	4.7 KB	4800 bytes (80 cols × 60 rows)

5.1 Control / Status Registers

Offset	Name	Width	Access	Behavior
0x00	CTRL	32 b	R/W	bit0 ENABLE (1 = enable video output), bit1 SWAP (write 1 to commit shadow sprite table at next VSYNC; auto-clears), bit2 HUD_ON, bits31:8 reserved
0x04	STATUS	32 b	R	bit0 VBLANK (1 during VBLANK), bit1 SWAP_PENDING, bits15:8 frame counter mod 256
0x08	BG_SCROLL	32 b	R/W	bits15:0 = scroll_x (signed), bits31:16 = scroll_y (signed). Implemented but not used in baseline; the arena is single-screen. Reserved for future level designs.
0x0C	IRQ_MASK	32 b	R/W	bit0 enables VBLANK interrupt to HPS (Phase 2; not used in baseline)

5.2 Player State Registers (0x0010–0x001F)

Pure mailbox registers. Software writes; hardware reads them only to drive an optional HUD overlay (HP bar, wave number, score). No game logic depends on them.

Offset	Field	Encoding
0x10	PLAYER_POS	bits15:0 = px (pixels), bits31:16 = py (pixels)
0x14	PLAYER_STATS	bits7:0 = hp, bits15:8 = wave, bits31:16 = level
0x18	SCORE	32-bit unsigned score
0x1C	KILL_COUNT	32-bit unsigned kills (HUD only)

5.3 Sprite Table Entry Format

Each sprite entry is 8 bytes (two 32-bit words). Software writes both words; hardware reads them as a single 64-bit value.

Word	Field	Bits	Meaning
W0	x	[15:0]	X position (signed pixels, −128 to 767 visible)
W0	y	[31:16]	Y position (signed pixels)
W1	sprite_id	[7:0]	Index into Sprite ROM. Only bits [5:0] are meaningful (0–63); bits [7:6] are reserved and must be written 0.
W1	reserved	[9:8]	reserved for future size encoding; must be 0. All sprites are 16×16 in baseline.
W1	flip	[11:10]	bit10 = HFlip, bit11 = VFlip
W1	priority	[14:12]	0 = highest, 7 = lowest (for per-scanline culling)
W1	active	[15]	1 = render this entry
W1	palette_offset	[23:16]	added to sprite-ROM palette index before lookup (allows team recoloring)
W1	reserved	[31:24]	0

Reads return last-written values (not the live shadow). **Writes** land in shadow immediately. Sprite table commits at next VSYNC after CTRL.SWAP=1 (Section 4.3). Palette and tile map use a

per-region 1-bit dirty flag: on any write, the flag sets; at the next `hsync` rising edge, active RAM is updated from shadow and the flag clears. No torn pixels within a scanline. SW can write any region at any time without waiting for VBLANK.

6 Software Architecture (C on HPS)

6.1 Source File Organization

File	Purpose
`main.c`	Entry point, mmap of `nml_gpu`, main loop, frame timing
`nml_gpu.h`	Register offsets, sprite struct, low-level write helpers
`nml_gpu.c`	Implementation of writers (`write_sprite`, `commit_frame`, etc.)
`usb_kbd.h/.c`	libusb keyboard reader thread, exposes input snapshot
`game.h/.c`	Top-level game state machine (menu / playing / level-up / dead)
`entity.h/.c`	Entity pool (player, enemies, projectiles, hazards) and update
`ai.c`	Enemy chase behavior, target selection
`collision.c`	AABB pairwise checks; emits damage events
`autoatk.c`	Auto-attack cooldowns and projectile spawning
`wave.h/.c`	Wave table, spawner, end-of-wave item selection
`render.c`	Translates entity pool into sprite-table writes
`audio.c`	(Stretch) sound effects via Wolfson WM8731

6.2 Header: `nml_gpu.h`

```c
#ifndef NML_GPU_H
#define NML_GPU_H

#include <stdint.h>

/* --- Register offsets (bytes from peripheral base) --- */
#define NML_REG_CTRL          0x0000
#define NML_REG_STATUS        0x0004
#define NML_REG_BG_SCROLL     0x0008
#define NML_REG_IRQ_MASK      0x000C

#define NML_REG_PLAYER_POS    0x0010
#define NML_REG_PLAYER_STATS  0x0014
#define NML_REG_SCORE         0x0018
#define NML_REG_KILL_COUNT    0x001C

#define NML_SPRITE_TABLE_BASE 0x0100   /* 32 entries * 8 B = 256 B */
#define NML_PALETTE_BASE      0x0400   /* 256 entries * 4 B = 1 KB */
#define NML_TILEMAP_BASE      0x1000   /* 80 * 60 = 4800 B */

/* --- CTRL bits --- */
#define NML_CTRL_ENABLE   (1u << 0)
#define NML_CTRL_SWAP     (1u << 1)
#define NML_CTRL_HUD_ON   (1u << 2)

/* --- STATUS bits --- */
#define NML_STATUS_VBLANK         (1u << 0)
#define NML_STATUS_SWAP_PENDING   (1u << 1)

/* --- Sprite descriptor (matches HW layout) --- */
```

```
typedef struct {
    int16_t  x;
    int16_t  y;
    uint8_t  sprite_id;
    uint8_t  flags;       /* bits1:0 reserved (size, must be 0),
                             bit2 hflip, bit3 vflip,
                             bits6:4 priority (3b), bit7 active */
    uint8_t  palette_off;
    uint8_t  reserved;
} nml_sprite_t;

/* --- Public API --- */
int  nml_open(void);                /* mmap and init */
void nml_close(void);
void nml_set_enable(int on);
void nml_write_sprite(int slot, const nml_sprite_t *s);
void nml_clear_sprites(void);
void nml_write_palette(int idx, uint8_t r, uint8_t g, uint8_t b);
void nml_write_tile(int col, int row, uint8_t tile_id);
void nml_set_player_state(int16_t px, int16_t py,
                          uint8_t hp, uint8_t wave, uint16_t level);
void nml_set_score(uint32_t score, uint32_t kills);
void nml_commit_frame(void);        /* sets SWAP, returns when committed */
int  nml_in_vblank(void);

#endif
```

6.3 Header: game.h

```
#ifndef GAME_H
#define GAME_H

#include <stdint.h>

#define MAX_ENTITIES   64    /* hard cap; sprite table caps render at 32 */
#define MAX_AUTOATK    8

typedef enum {
    ENT_NONE = 0,
    ENT_PLAYER,
    ENT_ENEMY_ARMED,         /* counts as combatant */
    ENT_ENEMY_UNARMED,       /* counts as non-combatant for end screen */
    ENT_PROJECTILE_PLAYER,
    ENT_PROJECTILE_AUTO,     /* mortar shell, gas puff, etc. */
    ENT_HAZARD_STATIC        /* barbed wire, gas cloud lingering */
} ent_kind_t;

typedef struct {
    ent_kind_t kind;
    int16_t x, y;            /* world position, pixels */
    int8_t  vx, vy;          /* per-frame velocity */
    uint8_t hp;
    uint8_t w, h;            /* AABB size */
    uint8_t sprite_id;
    uint8_t team;            /* 0 = player, 1 = enemy */
    uint16_t ttl;            /* frames until despawn (0 = persistent) */
    uint8_t  payload;        /* damage or auto-attack type */
```

```
} entity_t;

typedef enum {
    AA_MORTAR,
    AA_BARBED_WIRE,
    AA_GAS,
    AA_ARTILLERY,
    AA_COUNT
} autoatk_kind_t;

typedef struct {
    autoatk_kind_t kind;
    uint8_t  level;          /* 0 = not owned, 1..5 = upgraded */
    uint16_t cooldown;       /* frames remaining */
    uint16_t period;         /* frames between fires */
} autoatk_t;

typedef struct {
    entity_t  entities[MAX_ENTITIES];
    autoatk_t autoatks[AA_COUNT];
    uint16_t  wave;
    uint16_t  frame;
    uint32_t  score;
    uint32_t  kills_armed;
    uint32_t  kills_unarmed;
    uint8_t   player_level;
    uint16_t  xp;
    uint8_t   state;         /* MENU / PLAYING / LEVELUP / DEAD */
} game_t;

void game_init(game_t *g);
void game_tick(game_t *g, uint16_t input_bitmask);
void game_render(const game_t *g);   /* writes sprite table; must cull
                                        active entities down to <=32
                                        scored by: player > projectiles >
                                        nearby enemies > distant enemies
                                        > hazards. See render.c. */

#endif
```

6.4 Main Loop

Fixed-step 60 Hz, tied to VBLANK via busy-poll on `STATUS.VBLANK` (can be upgraded to a VBLANK interrupt in Phase 2).

```
int main(void) {
    nml_open();
    usb_kbd_start();             /* spawns reader thread */
    game_t g;
    game_init(&g);
    nml_set_enable(1);

    while (g.state != STATE_QUIT) {
        struct timespec t0, t1;
        clock_gettime(CLOCK_MONOTONIC, &t0);

        uint16_t input = usb_kbd_snapshot();
```

```
        game_tick(&g, input);
        game_render(&g);          /* writes sprites into shadow */
        nml_commit_frame();       /* SWAP at next VBLANK; blocks */

        clock_gettime(CLOCK_MONOTONIC, &t1);
        long us = (t1.tv_sec - t0.tv_sec) * 1000000L
                + (t1.tv_nsec - t0.tv_nsec) / 1000L;
        if (us > 16600) {         /* one frame at 60 Hz = 16.67 ms */
            fprintf(stderr, "frame overrun: %ld us\n", us);
        }
    }
    nml_close();
    return 0;
}
```

7 Hardware Architecture (SystemVerilog)

7.1 Module Hierarchy

```
nml_gpu                          (top-level peripheral)
|-- avalon_slave_iface           (Avalon-MM decode + register file)
|-- pll_25mhz                    (Quartus-generated PLL IP, 50 -> 25.175 MHz)
|-- vga_timing                   (HSYNC/VSYNC/x/y/blank generator, 640x480@60)
|-- sprite_table_ram             (M10K, 32 x 64b dual-port)
|-- tile_map_ram                 (M10K, 4800 x 8b dual-port)
|-- palette_ram                  (M10K, 256 x 24b dual-port)
|-- sprite_rom                   (M10K, 64 sprites x 256B, $readmemh)
|-- tile_rom                     (M10K, 64 tiles x 64B, $readmemh)
|-- sprite_eval                  (per-scanline bbox check + cull to 8)
|-- compositor                   (tile + sprite + palette pipeline)
|   |-- line_buffer (x2)         (640 x 24b, double buffered)
|-- hud_overlay                  (HP bar, wave, score; reads player regs)
```

7.2 Top Module Interface

```
module nml_gpu (
    // Avalon-MM slave (Lightweight bridge, 50 MHz)
    input  logic        clk,
    input  logic        reset_n,
    input  logic [13:0] avs_address,    // 16 KB window, byte addr
    input  logic        avs_read,
    input  logic        avs_write,
    input  logic [31:0] avs_writedata,
    input  logic [3:0]  avs_byteenable,
    output logic [31:0] avs_readdata,
    output logic        avs_waitrequest, // tied 0; always single-cycle

    // VGA (driven by internal pix_clk derived from PLL)
    output logic [7:0]  vga_r,
    output logic [7:0]  vga_g,
    output logic [7:0]  vga_b,
    output logic        vga_hs,
    output logic        vga_vs,
```

```
    output logic         vga_blank_n,
    output logic         vga_sync_n,
    output logic         vga_clk
);
```

7.3 Submodule Interfaces (Abridged)

```
module vga_timing (
    input  logic         pix_clk, reset_n,
    output logic [9:0]   x,           // 0..799 (640 visible + porches)
    output logic [9:0]   y,           // 0..524 (480 visible + porches)
    output logic         visible,
    output logic         hsync, vsync,
    output logic         vblank, hblank
);

module sprite_eval (
    input  logic         pix_clk, reset_n,
    input  logic [9:0]   next_scanline,          // y of upcoming line
    input  logic         eval_strobe,            // pulse during HBLANK
    // Sprite table read port
    output logic [4:0]   sprtab_raddr,
    input  logic [63:0]  sprtab_rdata,
    // Output: up to 8 active sprites for the next line
    output logic [7:0]   active_mask,            // which of 8 slots valid
    output logic [63:0]  line_sprites [0:7]
);

module compositor (
    input  logic         pix_clk, reset_n,
    input  logic [9:0]   x, y,
    input  logic         visible,
    input  logic [15:0]  scroll_x, scroll_y,
    // Tile map read
    output logic [12:0]  tilemap_raddr,
    input  logic [7:0]   tilemap_rdata,
    // Tile ROM read
    output logic [11:0]  tilerom_raddr,
    input  logic [7:0]   tilerom_rdata,
    // Sprite line input
    input  logic [7:0]   active_mask,
    input  logic [63:0]  line_sprites [0:7],
    output logic [13:0]  sprrom_raddr,
    input  logic [7:0]   sprrom_rdata,
    // Palette read
    output logic [7:0]   palette_raddr,
    input  logic [23:0]  palette_rdata,
    // Pixel output
    output logic [23:0]  rgb_out
);

module avalon_slave_iface (
    input  logic         clk, reset_n,
    input  logic [13:0]  avs_address,
    input  logic         avs_read, avs_write,
    input  logic [31:0]  avs_writedata,
    input  logic [3:0]   avs_byteenable,
```

```
    output logic [31:0] avs_readdata,
    // Register-file outputs to rest of design
    output logic        ctrl_enable,
    output logic        ctrl_swap_req,    // pulses 1 cycle on write
    output logic        ctrl_hud_on,
    output logic [31:0] bg_scroll,
    input  logic        vblank_in,
    input  logic        swap_pending_in,
    // Memory write fan-out (one strobe per region)
    output logic        sprtab_we, palette_we, tilemap_we,
    output logic [12:0] mem_waddr,
    output logic [31:0] mem_wdata,
    // Memory read fan-in for SW reads
    input  logic [31:0] sprtab_rdata_sw,
    input  logic [31:0] palette_rdata_sw,
    input  logic [31:0] tilemap_rdata_sw,
    // Player state out to HUD
    output logic [31:0] player_pos, player_stats, score, kill_count
);
```

7.4 Inter-Block Protocols

Most signals are plain wires of the widths given above, driven every cycle. Three interfaces have handshakes:

- **Avalon-MM slave:** `avs_waitrequest` tied 0. Writes commit on `avs_write` cycle; reads return combinationally same cycle. Every readable region is single-cycle accessible.
- **Sprite table swap:** SW writes shadow via `sprtab_we`, then `CTRL.SWAP=1`. Slave pulses `ctrl_swap_req` for one cycle; compositor latches a "swap pending" flag and flips active/shadow pointers on the next `vblank` rising edge. SW polls `STATUS.SWAP_PENDING` (or takes a Phase 2 VBLANK IRQ).
- **Sprite eval → compositor:** `sprite_eval` drives `active_mask` and `line_sprites[0:7]` during HBLANK; compositor samples them on the rising edge of `visible`. No backpressure: 160 idle pixel-clocks per HBLANK vs. 32 bbox checks at 1/cycle.

8 File Formats

The project reads two files at FPGA build time (sprite ROM, tile ROM) and one at program startup (wave table). No files are read during gameplay.

8.1 `sprite_rom.hex` – Sprite ROM

Intel-hex for `$readmemh`, one byte per line, no addresses. Exactly $64 \times 256 = 16384$ bytes. Each 256-byte block is one 16×16 sprite; pixel (u, v) of sprite S is at byte $S \cdot 256 + v \cdot 16 + u$. Byte value is a palette index; `0x00` = transparent.

8.2 `tile_rom.hex` – Tile ROM

Same format, $64 \times 64 = 4096$ bytes. Each 64-byte block is one 8×8 tile in row-major order. No transparency.

8.3 `waves.dat` – Wave Table

ASCII, one wave per line, whitespace-separated. Loaded once at startup into a static array; never modified at runtime.

```
# wave_num duration_sec spawn_period_frames armed_count unarmed_count enemy_hp
1   30   60   8    0   2
2   30   45  10    0   2
3   45   45  12    1   3
...
20  60   15  20   18   6
```

Lines starting with `#` are comments. The narrative arc (rising `unarmed_count`) lives entirely in this file.

9 Risk Register Update

Refinements from this design; the proposal's risk table still applies.

- **Per-scanline sprite cap 8**, enforced by `sprite_eval` priority-culling. The 32-entry table is a frame-level limit; only 8 highest-priority sprites render per line.
- **Frame-level cap 32**, enforced in SW by `game_render()`. Late waves produce 40+ enemies; without culling, overflow entries would silently not render and gameplay would desync from visuals.
- **Mid-video writes safe**: sprite table swaps at VSYNC after `CTRL.SWAP=1`; palette and tile map use HSYNC-edge latches from shadow (Section 5.3). SW never needs to gate writes on VBLANK.
- **PLL frequency**: Cyclone V PLL cannot produce exactly 25.175 MHz from 50 MHz; it locks to 25.000 or 25.200 MHz. Monitors tolerate ±0.5%; verify on physical VGA by week 2.
- **Sprite ROM port contention**: line-buffer architecture (Section 4.3) uses one sequential fetch stream, so one M10K dual-port ROM suffices. No replication needed.
- **Frame-overrun watchdog**: `main.c` logs frames >16.6 ms. Dev aid only; shipping builds silently drop overruns (VBLANK wait absorbs slack).

10 Milestones (Unchanged from Proposal)

Week	Milestone	Owner(s)
1–2	VGA tile rendering; USB keyboard input; basic sprite display	All
3–4	Player movement and shooting; enemy spawning, chase AI; SW collision	Rohit, Kambi
5	Auto-attacks from SW; item selection UI; narrative wave progression	Nicola, Rohit
6	Integration testing, visual polish, bug fixes, demo prep	All