SwarmPursuit

Hardware-Accelerated Multi-Agent Pursuit-Evasion on FPGA

Design Document

CSEE 4840 — Embedded System Design

Spring 2026 | Columbia University

Professor Stephen A. Edwards

Yen Chung Lo (yl5972) Junhao Qu (jq2434)

Kevin Liu (kl3755) Boxiong Li (bl3155)

1: Introduction

SwarmPursuit is a hardware-accelerated multi-agent pursuit-evasion game on the DE1-SoC FPGA. Two AI-controlled pursuers coordinate in real time to capture an evader in a tile-based 64×64 grid world with obstacles, rendered on VGA at 640×480 resolution and 60 frames per second. All three agents make decisions and move simultaneously in true hardware parallelism.

The system offers two play options sharing the same hardware infrastructure:

> **Option A — AI vs. AI.** All three agents are hardware-controlled. The two pursuers run the coordinated pursuit algorithm; the evader runs a complementary escape algorithm that maximizes distance from pursuers while seeking open corridors. The user watches the chase unfold and can reshape the obstacle layout in real time via the game controller to observe how agents adapt.
>
> **Option B — Human vs. AI.** The player controls the evader with a USB game controller, attempting to survive as long as possible against the two AI pursuers. The AI coordination quality becomes directly apparent as the pursuers attempt flanking maneuvers to corner the player.

2: System Block Diagram

This section presents two block diagrams as required by the design document guidelines: a top-level view (Figure 1) showing the HPS, the custom FPGA peripheral, and external hardware; and an internal view (Figure 2) showing the structure of the SwarmPursuit peripheral.

2.1 - Top-Level Block Diagram

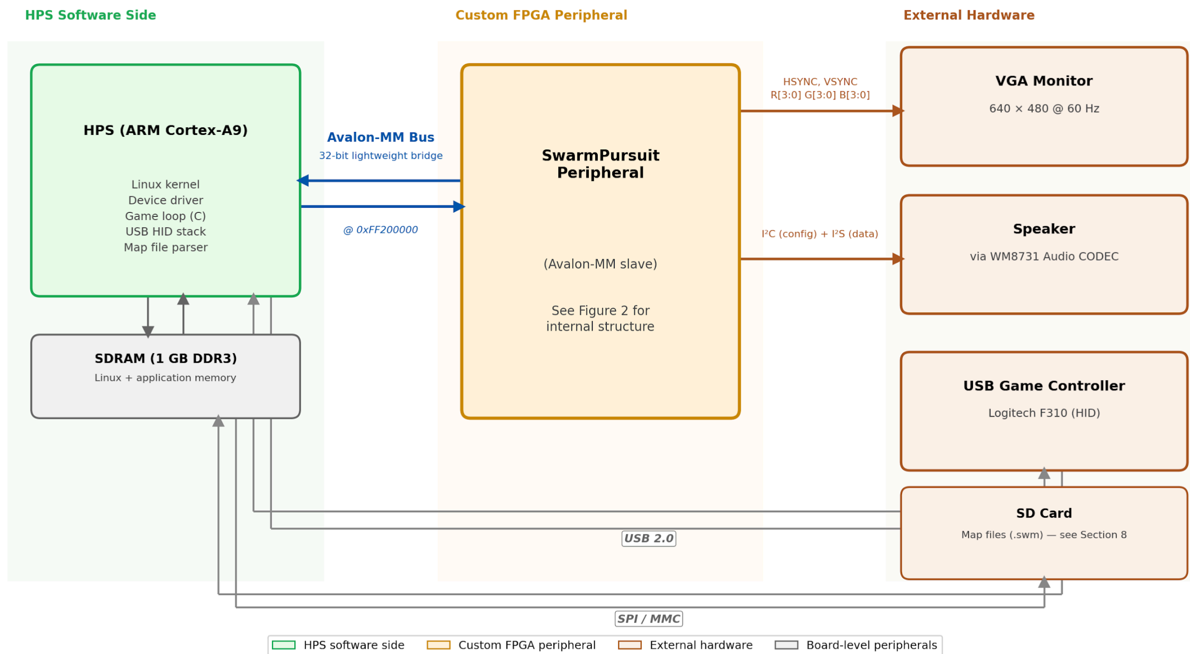

Figure 1: Top-Level System Block Diagram. The HPS (running Linux) communicates with the custom SwarmPursuit peripheral over the 32-bit Avalon-MM lightweight bridge. External peripherals are the VGA monitor, audio speaker (via WM8731 CODEC), USB game controller, and SD card (for scenario map files).

2.2 - External Communication Protocols

Avalon-MM Bus (HPS ↔ SwarmPursuit Peripheral)

The HPS communicates with SwarmPursuit via Platform Designer as an Avalon-MM slave. The bus is 32 bits wide with a 6-bit word-addressed offset (byte addresses 0x00–0x30, see Section 5). The kernel driver accesses the register region through the lightweight HPS-FPGA bridge at physical address 0xFF200000, mapped into kernel virtual space with `ioremap()`.

Write transaction: HPS asserts `avs_write=1`, drives `avs_address[5:0]` and `avs_writedata[31:0]` on the same clock edge. The peripheral latches writedata on the rising edge of the next clock. `avs_waitrequest` is held low (0-wait-state writes) for all registers except GRID_WR_DATA, which may assert waitrequest for 1 cycle while the M10K BRAM write completes.

Read transaction: HPS asserts `avs_read=1` with `avs_address` stable. `avs_readdata` is valid on the following clock edge (1-cycle read latency). `avs_waitrequest` is not asserted for reads.

USB HID (Logitech F310 ↔ HPS)

The Logitech F310 connects to the DE1-SoC's USB Type-A port and is enumerated by the Linux kernel as a USB HID gamepad. Software reads it via libusb-1.0 using interrupt transfers on the HID interrupt IN endpoint (endpoint 0x81, polling interval 10 ms).

F310 HID input report format (D-Input mode, 8 bytes):

Byte	Field	Meaning
0	Left stick X	0x00=left, 0x80=center, 0xFF=right
1	Left stick Y	0x00=up, 0x80=center, 0xFF=down
2	Right stick X	same encoding
3	Right stick Y	same encoding
4	[7:4] D-pad hat [3:0] A/B/X/Y buttons	0=N,1=NE,2=E,3=SE,4=S,5=SW,6=W,7=NW,0xF=centered bit per button (1=pressed)
5	Shoulder/trigger	LB, RB, LT, RT, Start, Back (bit per button)
6	Thumbstick clicks	L3, R3 clicks
7	Reserved	always 0x00

Table 1: F310 HID report format.

The `usb_input` thread extracts byte 4[7:4] (hat switch nibble) and maps it to a direction enum (DIR_STAY, DIR_N, DIR_E, DIR_S, DIR_W — diagonals and centered both map to DIR_STAY). Byte 5[4] (Start) toggles the MODE_SEL bit. Buttons A and B (byte 4[1:0]) place/remove obstacles at a controller-driven cursor. The result is written to EVADER_CMD each polling cycle.

WM8731 Audio CODEC (FPGA ↔ Speaker)

The DE1-SoC's WM8731 CODEC is controlled directly by FPGA logic via two protocols: I²C for configuration and I²S for audio data. At system startup, the audio_codec_init module issues approximately 10 I²C writes to configure the CODEC for 48 kHz sample rate, 16-bit samples, line-out enabled, and headphone muted. After initialization, the tone_generator module streams stereo 16-bit PCM samples over I²S continuously. See Section 3.6 for waveform generation details.

2.3 - Internal Peripheral Block Diagram

Figure 2 shows the internal structure of the SwarmPursuit custom peripheral. All blocks are synchronous to the 50 MHz system clock unless noted. The VGA pixel clock (25.175 MHz) is generated by a PLL from the 50 MHz clock. The I²S bit clock (~3.07 MHz for 48 kHz stereo 16-bit) is derived by a dedicated audio clock divider.

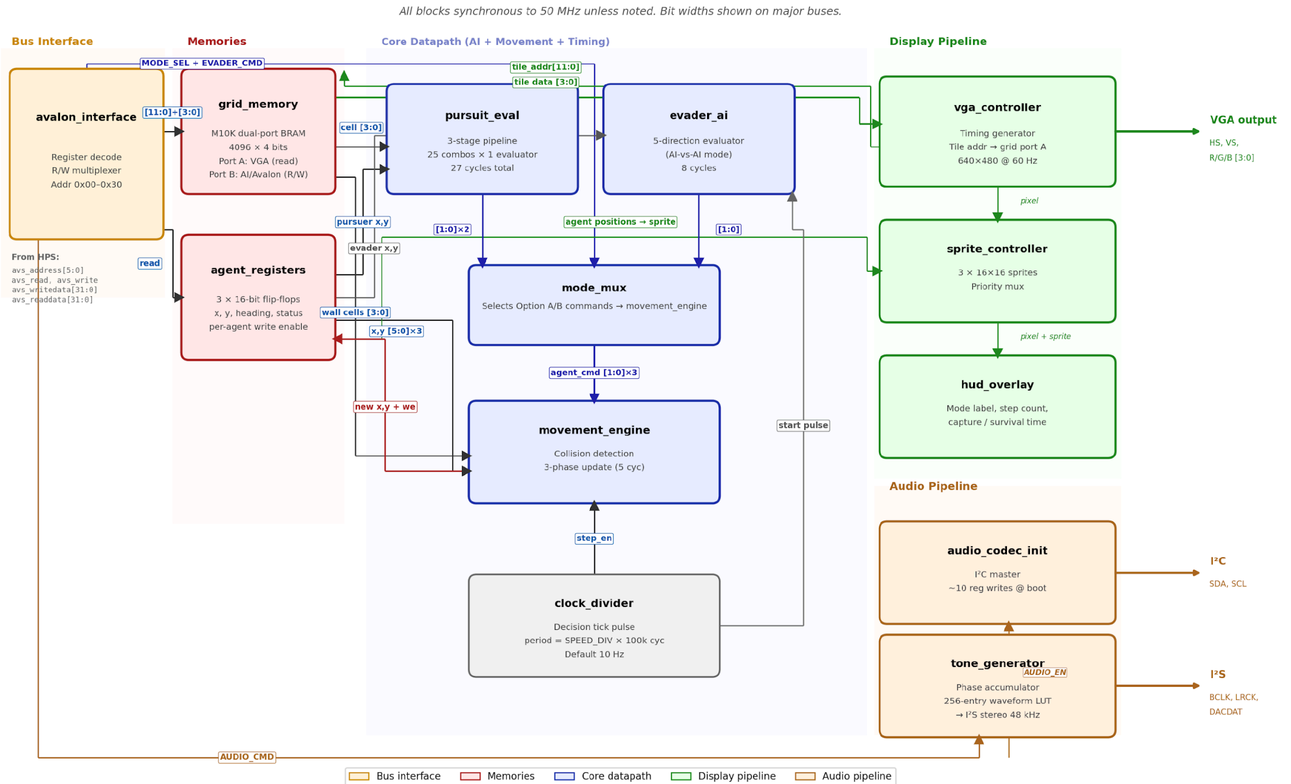

Figure 2: SwarmPursuit peripheral internal structure. Orange = Avalon bus interface, red = memories, blue = core AI/movement datapath, green = VGA display pipeline, tan = audio pipeline. Bit widths shown on major buses; arrows indicate data flow.

2.4 - Internal Connection Reference

The table below enumerates every connection in Figure 2 with its bit width and protocol. For simple connections (combinational, no handshake), only width and meaning are given. For pipelined connections with valid/ready handshake, the protocol is noted.

Source → Destination	Width	Protocol / Notes
Avalon → grid_memory (write)	[11:0]+[3:0]	Gated by avs_write to GRID_WR_* regs; may waitrequest 1 cyc
Avalon → agent_registers (read)	[31:0]	Combinational read mux; 1-cycle Avalon latency
Avalon → mode_mux (mode_sel)	[0]	Single bit from MODE_CTRL[0]

Avalon → evader_cmd	[2:0]	Direction from EVADER_CMD[2:0]; latched on step_en
Avalon → config regs (λ, μ, ν)	[7:0] × 3	Combinational, sampled each decision tick
Avalon → audio_cmd reg	[12:0]	Trigger bit self-clears; BGM_FREQ sampled continuously
grid_memory port A → vga_controller	[11:0] addr → [3:0] data	Driven by pixel counter; 1-cyc read latency
grid_memory port B → pursuit_eval	[11:0] addr → [3:0] data	25 sequential reads over pipeline fill
grid_memory port B → movement_engine	[11:0] addr → [3:0] data	3 reads for 3 agents' wall checks
agent_registers → pursuit_eval	[5:0]×2 (x,y per pursuer)	Broadcast; sampled on step_en
agent_registers → evader_ai	[5:0]×1 (evader x,y)	Broadcast
agent_registers → movement_engine	[5:0]×3 (all agents)	Broadcast current positions
agent_registers → vga_controller	[5:0]×3 (all agents)	Drives sprite position
pursuit_eval → mode_mux	[1:0]×2 (pursuer dirs)	Valid signal asserted when pipeline done (cycle 27)
evader_ai → mode_mux	[1:0]×1 (evader dir)	Valid signal asserted cycle 8
mode_mux → movement_engine	[1:0]×3 (all cmds)	Combinational select between AI and human evader
movement_engine → agent_registers	new_x/y[5:0]×3, we[2:0]	Committed on clock edge if move_valid
clock_divider → all modules	step_en [0]	1-cycle pulse every SPEED_DIV cycles
vga_controller → VGA pins	vga_hs, vga_vs, vga_r/g/b[3:0]	Standard 640×480@60Hz timing
audio_codec_init → WM8731	i2c_sda, i2c_scl	~10 register writes at startup, then idle
tone_generator → WM8731	i2s_bclk, i2s_lrck, i2s_dacdat	Continuous 48kHz stereo PCM
vsync_pulse → status reg	[0]	1-cycle pulse at start of vertical blanking

Table 2: Internal connections within the SwarmPursuit peripheral.

2.5 - Internal Memories

Memory	Type	Depth × Width	Ports	Purpose
grid_memory	M10K BRAM	4096 × 4 bits	Dual-port	Cell type for all 64×64 cells
agent_registers	Flip-flops	3 × 16 bits	3 write, broadcast read	Agent x, y, heading, status
sprite_rom	M10K BRAM	3 × 256 × 9 bits	Single-port read	Sprite patterns (16×16 × color+mask)
color_lut	LUT RAM	16 × 12 bits	Combinational read	Cell type → RGB mapping
waveform_lut	M10K BRAM	256 × 16 bits	Single-port read	Audio waveform sample table
score_pipeline	Flip-flops	~30 × 16 bits	Pipeline registers	Pursuit eval pipeline state

Table 3: Memories inside the SwarmPursuit peripheral.

3: Algorithms

3.1 - Coordinated Pursuit Evaluation

At each decision tick (default 10 Hz), the hardware evaluates all 25 joint move combinations for the two pursuers. Each pursuer can move North, East, South, West, or Stay, producing 5×5 = 25 candidate joint moves. The evaluation selects the combination yielding the highest coordination score.

3.1.1 Score Function

For a candidate placing Pursuer 1 at (x_1, y_1) and Pursuer 2 at (x_2, y_2), with the evader at (e_x, e_y):

$$\mathbf{Score} = -(d_1 + d_2) + \lambda \cdot \mathrm{spread} - \mu \cdot \mathrm{escape_corridors}$$

where:

d_1, d_2 are Manhattan distances from each pursuer candidate position to the evader: $d_i = |x_i - e_x| + |y_i - e_y|$. Minimizing total distance is the primary pursuit objective.

3.1.2 Avoiding Arctan: Cross-Product Flanking Metric

The proposal originally specified an angular separation term θ_sep between the two pursuer-to-evader vectors, rewarding flanking formations. Computing arctan in hardware is expensive (requires CORDIC or large lookup tables). We replace it with the absolute cross-product of the two relative vectors:

$$\mathrm{spread} = |\Delta x_1 \cdot \Delta y_2 - \Delta y_1 \cdot \Delta x_2|$$

where $\Delta x_i = x_i - e_x$ and $\Delta y_i = y_i - e_y$. Mathematically this equals $|v_1| \cdot |v_2| \cdot \sin(\theta)$, where θ is the angle between the two vectors. It is maximized when pursuers are at 90° relative to the evader — the ideal flanking geometry. Implementation cost is only two signed 8-bit multiplications and a subtraction, dramatically cheaper than arctan. This directly addresses the TA's feedback on efficient angular computation.

escape_corridors is the count of the evader's 4 cardinal neighbors that are neither walls nor occupied by a pursuer in the candidate configuration. It ranges 0–4 and requires 4 parallel comparators. Fewer open corridors means a more cornered evader.

λ, μ are tunable 8-bit fixed-point (4.4 format) weights stored in the CONFIG register, set at runtime via Avalon. Default values (λ=1.5, μ=2.0) will be tuned using the SDL2 desktop prototype.

3.1.3 Pipelined Evaluator Architecture (Primary)

Following TA feedback, we implement a 3-stage pipelined single evaluator as our primary design, rather than 25 parallel combinational evaluators. This is more area-efficient and avoids the long combinational path that 25-wide parallel evaluation would introduce (which risks timing closure at 50 MHz).

Stage	Operation	Latency	Key Hardware
1: Candidate Generation	Compute candidate positions (p1_x ± 1, p1_y ± 1, stay) for both pursuers. Wall check via grid port B read.	1 cyc	2 adders, 2 grid RAM reads, 2 wall comparators
2: Score Computation	Compute d1, d2 (Manhattan), Δx, Δy vectors, cross-product spread (2 multipliers), escape_corridor count (4 comparators).	1 cyc	2 signed 8-bit multipliers, 6 adders, 4 comparators
3: Accumulate & Compare	Apply weights λ, μ to spread and escape terms. Sum final score. Compare against running max, latch winning move pair if higher.	1 cyc	2 multipliers, 1 adder, 1 comparator, max register + move latch

Table 4: Pursuit evaluation pipeline stages. After a 3-cycle fill latency, one evaluation completes per clock cycle. All 25 combinations finish in 25 + 2 = 27 cycles. At 50 MHz this is 0.54 μs — 0.0005% of the 100 ms decision period.

3.2 - Evader AI (Option A: AI vs. AI)

In AI-vs-AI mode, the evader runs a simpler escape-maximizing algorithm. For each of 5 candidate directions (N, E, S, W, Stay), the evader computes:

$$\mathbf{Score_evade = min(d_1, d_2) + v \cdot corridors}$$

where d_1, d_2 are Manhattan distances from the candidate position to each pursuer, and corridors counts open (non-wall) cardinal neighbors of the candidate position. The weight v (8-bit fixed-point) encourages seeking open space. 5 evaluations complete in 8 cycles after pipeline fill. The evader_ai module shares the pipelined datapath with pursuit_eval via a time-multiplexing mux — pursuit_eval runs cycles 0–27, then evader_ai runs cycles 28–35. Total: 35 cycles = 0.70 μs per decision tick.

3.3 - Movement Engine and Collision Detection

After AI modules produce chosen directions for each agent, the movement_engine executes a three-phase synchronous update:

Phase 1 (1 cyc) — Candidate position. For each agent, add direction delta (dx, dy ∈ {−1, 0, +1}) to current position. Clamp to grid bounds [0, 63].

Phase 2 (3 cyc) — Wall check. Read grid cell at each candidate position via grid_memory port B (3 sequential reads). If the cell is a wall, that agent stays at its current position; status = BLOCKED.

Phase 3 (1 cyc) — Agent-agent collision. Compare all 3 candidate positions pairwise (3 comparisons). If two agents would occupy the same cell, the lower-index agent (Pursuer 1) gets priority; the other stays. Commit all valid positions simultaneously.

Total movement update: 5 clock cycles = 100 ns at 50 MHz. All updates happen between decision ticks, well within the 100 ms window.

3.4 - VGA Tile-Based Rendering

The vga_controller generates standard 640×480 @ 60 Hz timing using a 25.175 MHz pixel clock derived from the 50 MHz system clock via a PLL. The 64×64 grid maps to 10×7 pixel tiles, filling 640×448 pixels. The bottom 32-pixel strip is reserved for the HUD overlay.

For each pixel during active video, the renderer computes tile_col = h_count / 10 and tile_row = v_count / 7 (implemented as lookup ROMs, not division). It reads the 4-bit cell type from grid_memory port A and maps it through the Color LUT:

Cell Type	Value	Color (RGB 4:4:4)	Meaning
FLOOR	0x0	#D0D0D0 light gray	Empty walkable cell
WALL	0x1	#303030 dark gray	Impassable obstacle
PURSUER_TRAIL	0x2	#FF8800 orange	Recent pursuer path (decays)
EVADER_TRAIL	0x3	#0088FF blue	Recent evader path (decays)
HIGHLIGHT	0x4	#FFFF00 yellow	Controller cursor position
Reserved	0x5–0xF	—	Future use

Table 5: Cell type color mapping in the VGA Color LUT.

3.5 - Sprite Controller

Three 16×16-pixel sprites represent the agents. Sprite pixel position is derived from the grid position: sprite_x = grid_x × 10 − 3, sprite_y = grid_y × 7 − 4, centering the 16×16 sprite on the 10×7 tile. Sprite pattern ROM stores 3 patterns × 16 × 16 pixels × (8-bit color + 1-bit mask) ≈ 864 bytes. On each VGA pixel cycle, the sprite_controller checks whether any active sprite covers the current (h_count, v_count) and if so overrides the tile color. Priority is fixed: Pursuer 1 > Pursuer 2 > Evader. In normal gameplay sprites do not overlap (the movement engine prevents it), so priority affects only degenerate states.

3.6 - Audio Generation

The DE1-SoC's WM8731 CODEC is driven directly by FPGA logic without going through the HPS. Two hardware modules cooperate:

3.6.1 CODEC Initialization (I²C)

At startup, audio_codec_init issues approximately 10 I²C writes to configure the WM8731. The required register writes (from the WM8731 datasheet) are:

Reg	Addr	Value	Purpose
`R15`	`0x0F`	`0x0000`	Reset CODEC
`R6`	`0x06`	`0x0010`	Power on (DAC, output, digital core)
`R0`	`0x00`	`0x0017`	Left line in: 0 dB, unmuted
`R1`	`0x01`	`0x0017`	Right line in: 0 dB, unmuted

R2	0x02	0x0079	Left headphone: 0 dB
R3	0x03	0x0079	Right headphone: 0 dB
R4	0x04	0x0010	Analog path: DAC enabled, mic bypassed
R5	0x05	0x0000	Digital path: no de-emphasis, unmuted
R7	0x07	0x0002	Digital format: I^{2}S, 16-bit, slave
R8	0x08	0x0000	Sampling: 48 kHz normal mode
R9	0x09	0x0001	Active: digital interface on

Table 6: WM8731 I^{2}C initialization sequence. Each write is 3 bytes at 100 kHz — total init time approximately 3 ms. The init module is self-contained and triggers automatically on reset.

Phase accumulator increments by a value proportional to bgm_freq each sample tick, top bits index the 256-entry sine LUT, output scales and drives the left channel; sfx_trigger starts a one-shot envelope on the right channel that plays a short waveform keyed by sfx_code.

4: Resource Budgets

4.1 - On-Chip Memory

Resource	Type	Bits	KB	% of M10K	Notes
grid_memory	M10K BRAM	16,384	2.0	0.29%	64×64 × 4 bits, dual-port
agent_registers	Flip-flops	48	<0.01	—	3 × 16 bits
sprite_rom	M10K BRAM	6,912	0.84	0.12%	3 × 256 × 9 bits
color_lut	Registers	192	0.02	—	16 × 12 bits
waveform_lut	M10K BRAM	4,096	0.5	0.07%	256 × 16 bits
score_pipeline	Flip-flops	~480	0.06	—	Pipeline state registers
config_regs	Flip-flops	32	<0.01	—	λ, μ, ν, SPEED_DIV
Total		**~28,144**	**~3.5**	**~0.5%**	**Well within 5,570 Kbits**

Table 8: On-chip memory budget. The Cyclone V SE 5CSEMA5F31C6 provides 5,570 Kbits of M10K block RAM. Our design uses approximately 3.5 KB (0.5% utilization). As the TA noted, memory is not a concern for tile-based rendering at this grid size.

4.2 - Logic Utilization (Estimated)

Module	Est. ALMs	DSP Blocks	Notes
Avalon interface	~100	0	Address decode, R/W mux
grid_memory wrapper	~50	0	Port arbitration, address decode
agent_registers	~50	0	3 × 16-bit FF arrays
movement_engine	~150	0	3-agent parallel FSM + collision
clock_divider	~30	0	8-bit counter + pulse
pursuit_eval (pipelined)	~400	4	2 signed multipliers × 2 pipeline stages
evader_ai	~100	1	Time-multiplexed with pursuit_eval
mode_mux	~20	0	2:1 direction mux × 3 agents
vga_controller	~250	0	Timing gen + tile decoder
sprite_controller	~200	0	3 sprites, position compare + priority
hud_overlay	~100	0	Simple bitmap text in bottom strip
audio_codec_init	~80	0	I²C master FSM
tone_generator	~150	0	Phase accumulator + I²S serializer
Misc glue logic	~100	0	Config regs, status reg, reset logic
Total	**~1,780**	**5**	**~5.5% of 32,075 ALMs, ~5.7% of 87 DSPs**

Table 9: Logic utilization estimate. With the pipelined pursuit evaluator we fit comfortably under 10% of Cyclone V SE resources, leaving substantial headroom for the parallel 25-core upgrade (which would add ~3,500 ALMs, bringing total to ~17%).

4.3 - Timing Budget

Operation	Cycles	Time @ 50 MHz	Deadline
Pursuit evaluation (25 combos, pipelined)	27	0.54 μs	100 ms (10 Hz tick)
Evader evaluation (5 dirs)	8	0.16 μs	100 ms
Movement + 3-agent collision	5	0.10 μs	100 ms
Total per decision tick	~40	0.80 μs	100 ms (0.001%)
VGA pixel output	continuous	25.175 MHz	Hard real-time
I²S audio sample	continuous	48 kHz	Hard real-time
I²C CODEC init	~150,000	3 ms	Once at startup

Table 10: Timing budget. AI and movement consume negligible time; the critical real-time constraints are VGA pixel streaming and I²S audio sample streaming, both of which are continuous low-bandwidth pipelines.

5: Hardware/Software Interface

All HPS↔FPGA communication uses 32-bit Avalon-MM word-addressed registers at base address 0xFF200000 via the lightweight HPS-FPGA bridge. The kernel driver maps this region with `ioremap()` and exposes the `/dev/swarm0` character device to user space with `ioctl`-based access.

5.1 - Register Map

Offset	Name	R/W	Description
0x00	MODE_CTRL	R/W	Mode select, reset, start/stop, audio enable
0x04	STATUS	R	Vsync flag, capture flag, sim active, step count
0x08	AGENT0_STATE	R	Pursuer 1: x, y, heading, status
0x0C	AGENT1_STATE	R	Pursuer 2: x, y, heading, status
0x10	AGENT2_STATE	R	Evader: x, y, heading, status
0x14	EVADER_CMD	W	Human evader direction command
0x18	GRID_WR_ADDR	W	Target cell row/col for write
0x1C	GRID_WR_DATA	W	Cell type to write; triggers M10K write
0x20	GRID_RD_ADDR	W	Cell address for read
0x24	GRID_RD_DATA	R	Cell type at GRID_RD_ADDR
0x28	CONFIG	R/W	AI weights λ, μ, ν; speed divider
0x2C	AUDIO_CMD	W	SFX event code + BGM frequency + enables
0x30	SCORE_DBG	R	Debug: last winning pursuit score

Table 11: Complete register map.

6: Verilog Module Interfaces

This section provides port declarations for every SystemVerilog module, equivalent to function declarations in a C header. All modules are clocked on the rising edge of clk (50 MHz) unless noted as purely combinational.

6.1 - Module Hierarchy

```
swarm_top.sv                                  // Platform Designer component top
├── avalon_interface.sv                       // Avalon-MM slave register decode
├── grid_memory.sv                            // M10K dual-port BRAM 4096×4b
├── agent_registers.sv                        // 3 × 16b agent state
├── clock_divider.sv                          // Decision tick generator
├── movement_engine.sv                        // Collision detect + position commit
├── mode_mux.sv                               // Routes Option A vs B move cmds
├── pursuit_eval.sv                           // 3-stage pipelined evaluator
├── evader_ai.sv                              // 5-dir escape evaluator
├── vga_controller.sv                         // VGA timing + tile addressing
│   ├── tile_decoder.sv                       // 4-bit cell → 12-bit RGB
│   └── sprite_controller.sv                  // 3-sprite overlay
├── hud_overlay.sv                            // Bottom strip text rendering
├── audio_codec_init.sv                       // I²C config master
└── tone_generator.sv                         // Phase accumulator + I²S
```

6.2 - Top-Level Module

swarm_top.sv

```
module swarm_top (
  input  logic        clk, rst_n,                 // 50 MHz, active-low reset
  // Avalon-MM slave
  input  logic [5:0]  avs_address,
  input  logic        avs_read, avs_write,
  input  logic [31:0] avs_writedata,
  output logic [31:0] avs_readdata,
  output logic        avs_waitrequest,
  // VGA pins
  output logic        vga_hs, vga_vs,
  output logic [3:0]  vga_r, vga_g, vga_b,
  // WM8731 audio CODEC pins
  inout  logic        i2c_sda,                     // bidirectional
  output logic        i2c_scl,
  output logic        i2s_bclk, i2s_lrck, i2s_dacdat
);
```

6.3 - Memory Modules

grid_memory.sv — M10K dual-port BRAM

```
module grid_memory (
  input  logic        clk,
  // Port A: VGA read (continuous raster)
  input  logic [11:0] rd_addr_a,
  output logic [3:0]  rd_data_a,
  // Port B: AI/Avalon R/W
  input  logic        we_b,
  input  logic [11:0] addr_b,
  input  logic [3:0]  wr_data_b,
```

```
  output logic [3:0]  rd_data_b
);
```

agent_registers.sv

```
module agent_registers (
  input  logic        clk, rst,
  input  logic [2:0]  we,                          // per-agent write enable
  input  logic [5:0]  new_x [2:0], new_y [2:0],
  input  logic [1:0]  new_heading [2:0], new_status [2:0],
  output logic [5:0]  x [2:0], y [2:0],
  output logic [1:0]  heading [2:0], status [2:0]
);
```

6.4 - AI Modules

pursuit_eval.sv — 3-stage pipelined evaluator

```
module pursuit_eval (
  input  logic        clk, rst,
  input  logic        start,                       // pulse to begin 25-combo eval
  input  logic [5:0]  pursuer_x [1:0], pursuer_y [1:0],
  input  logic [5:0]  evader_x, evader_y,
  input  logic [7:0]  lambda, mu,                  // weights from CONFIG
  output logic [11:0] grid_addr_out,               // to grid_memory port B
  input  logic [3:0]  grid_data_in,                // cell type read
  output logic [1:0]  best_dir [1:0],              // winning move pair
  output logic [15:0] best_score,                  // debug/tuning
  output logic        valid                        // asserted cycle 27
);
```

evader_ai.sv

```
module evader_ai (
  input  logic        clk, rst, start,
  input  logic [5:0]  pursuer_x [1:0], pursuer_y [1:0],
  input  logic [5:0]  evader_x, evader_y,
  input  logic [7:0]  nu,                          // corridor bonus weight
  input  logic [3:0]  grid_data_in,
  output logic [11:0] grid_addr_out,
  output logic [2:0]  evader_dir,                  // 3-bit dir
  output logic        valid
);
```

mode_mux.sv — combinational

```
module mode_mux (
  input  logic        mode_sel,                    // 0=Opt B, 1=Opt A
  input  logic [1:0]  pursuit_dir [1:0],
  input  logic [1:0]  ai_evader_dir,
  input  logic [2:0]  human_evader_dir,
  output logic [1:0]  agent_cmd [2:0]
);
```

6.5 - Movement and Timing

movement_engine.sv

```
module movement_engine (
  input  logic         clk, rst, step_en,
  input  logic [5:0]  x [2:0], y [2:0],
  input  logic [1:0]  cmd [2:0],                 // from mode_mux
  input  logic [3:0]  fwd_cell [2:0],            // cell at each candidate pos
  output logic [5:0]  new_x [2:0], new_y [2:0],
  output logic [1:0]  new_heading [2:0], new_status [2:0],
  output logic [2:0]  agent_we,                  // write enable per agent
  output logic [11:0] fwd_addr [2:0]             // addresses for fwd_cell reads
);
```

clock_divider.sv

```
module clock_divider (
  input  logic         clk, rst, run,
  input  logic [7:0]  speed_div,                 // period = speed_div × 100k cyc
  output logic         step_en                   // 1-cycle pulse
);
```

6.6 - Display Modules

vga_controller.sv

```
module vga_controller (
  input  logic         clk_50, clk_pixel, rst,
  output logic [11:0] tile_addr,                 // to grid_memory port A
  input  logic [3:0]  tile_data,
  input  logic [5:0]  agent_x [2:0], agent_y [2:0],
  input  logic [1:0]  agent_status [2:0],
  input  logic         mode_sel,
  input  logic [12:0] step_count,
  input  logic         capture_flag,
  output logic         vga_hs, vga_vs,
  output logic [3:0]  vga_r, vga_g, vga_b,
  output logic         vsync_pulse
);
```

sprite_controller.sv, tile_decoder.sv, hud_overlay.sv

These submodules of vga_controller each expose the current pixel position and grid/agent state and return the overlay color (sprite_controller), tile base color (tile_decoder), or HUD text color (hud_overlay). Port lists follow the same pattern as vga_controller with pixel (h_count, v_count) inputs and 12-bit RGB outputs. Full port lists included in the Verilog source.

6.7 - Audio Modules

audio_codec_init.sv — I²C master for startup config

```
module audio_codec_init (
  input  logic  clk, rst,
  output logic  i2c_scl,
  inout  logic  i2c_sda,                         // bidirectional
  output logic  init_done                        // asserted when all writes complete
);
```

tone_generator.sv — I²S audio source

```
module tone_generator (
  input  logic        clk, rst,
  input  logic        audio_en,
  input  logic [3:0]  sfx_code,
  input  logic        sfx_trigger,               // 1-cyc pulse from AUDIO_CMD write
  input  logic [7:0]  bgm_freq,
  output logic        i2s_bclk, i2s_lrck, i2s_dacdat
);
```

7: C Software Interface

This section contains the complete C header (`swarm.h`) defining constants, types, and function prototypes for all software interacting with the SwarmPursuit peripheral. This is the interface between the kernel driver, the user-space game loop, the USB input thread, and the map loader.

```
#ifndef SWARM_H
#define SWARM_H
#include <stdint.h>

/* Register offsets */
#define SWARM_REG_MODE_CTRL     0x00
#define SWARM_REG_STATUS        0x04
#define SWARM_REG_AGENT0        0x08
#define SWARM_REG_AGENT1        0x0C
#define SWARM_REG_AGENT2        0x10
#define SWARM_REG_EVADER_CMD    0x14
#define SWARM_REG_GRID_WR_ADDR  0x18
#define SWARM_REG_GRID_WR_DATA  0x1C
#define SWARM_REG_GRID_RD_ADDR  0x20
#define SWARM_REG_GRID_RD_DATA  0x24
#define SWARM_REG_CONFIG        0x28
#define SWARM_REG_AUDIO_CMD     0x2C
#define SWARM_REG_SCORE_DBG     0x30

/* MODE_CTRL bit fields */
#define MODE_HUMAN_VS_AI  0
#define MODE_AI_VS_AI     1
#define MODE_CTRL_SEL(m)  ((m) & 0x1)
#define MODE_CTRL_RESET   (1 << 1)
#define MODE_CTRL_RUN     (1 << 2)
#define MODE_CTRL_AUDIO   (1 << 3)

/* Cell type encoding (GRID_WR_DATA[3:0]) */
typedef enum {
    CELL_FREE          = 0,
    CELL_WALL          = 1,
    CELL_PURSUER_TRAIL = 2,
    CELL_EVADER_TRAIL  = 3,
    CELL_HIGHLIGHT     = 4
} cell_type_t;

/* Direction encoding (EVADER_CMD[2:0]) */
typedef enum {
    DIR_STAY = 0, DIR_N = 1, DIR_E = 2, DIR_S = 3, DIR_W = 4
} direction_t;

/* STATUS field extraction */
#define STATUS_VSYNC(r)       ((r) & 0x1)
#define STATUS_CAPTURE(r)     (((r) >> 1) & 0x1)
#define STATUS_SIM_ACTIVE(r)  (((r) >> 2) & 0x1)
#define STATUS_STEP_COUNT(r)  (((r) >> 3) & 0x1FFF)

/* CONFIG build macro */
#define CONFIG_BUILD(lam, mu, nu, spd) \
    (((spd) & 0xFF) << 24 | ((nu) & 0xFF) << 16 | \
     ((mu)  & 0xFF) << 8  | ((lam) & 0xFF))

/* AUDIO_CMD bit fields */
typedef enum {
```

```
    SFX_SILENCE         = 0x0,
    SFX_CAPTURE         = 0x1,
    SFX_MODE_SWITCH     = 0x2,
    SFX_DECISION_TICK   = 0x3,
    SFX_OBSTACLE_PLACED = 0x4
} sfx_code_t;

#define AUDIO_CMD_BUILD(sfx, trig, bgm) \
    (((bgm) & 0xFF) << 5 | ((trig) & 0x1) << 4 | ((sfx) & 0xF))

/* Parsed agent state */
typedef struct {
    uint8_t x, y;    /* 0-63 */
    uint8_t heading; /* 0=N 1=E 2=S 3=W */
    uint8_t status;  /* 0=active 1=blocked 2=captured 3=idle */
} agent_state_t;

/* Driver API */
int  swarm_open(void);
void swarm_close(void);

void swarm_set_mode(int mode);        /* MODE_HUMAN_VS_AI or MODE_AI_VS_AI */
void swarm_reset(void);
void swarm_run(int run);

void swarm_read_agent(int idx, agent_state_t *out);

void swarm_write_cell(uint8_t row, uint8_t col, cell_type_t type);
cell_type_t swarm_read_cell(uint8_t row, uint8_t col);

void swarm_send_evader_cmd(direction_t dir);

uint32_t swarm_read_status(void);
void     swarm_wait_vsync(void);

void swarm_set_config(uint8_t lambda, uint8_t mu,
                      uint8_t nu, uint8_t speed_div);

void swarm_trigger_sfx(sfx_code_t code);
void swarm_set_bgm(uint8_t freq);
void swarm_audio_enable(int on);

int  swarm_load_map(const char *path);     /* see Section 8 */

#endif /* SWARM_H */
```

7.1 - Main Game Loop

```
int main(int argc, char *argv[]) {
    swarm_open();
    swarm_set_config(0x18, 0x20, 0x10, 50);   /* defaults, 10 Hz */
    swarm_audio_enable(1);
    swarm_set_bgm(100);                        /* ~100 Hz BGM */

    if (argc > 1) swarm_load_map(argv[1]);     /* preset map */

    swarm_set_mode(MODE_HUMAN_VS_AI);
    swarm_reset();
    swarm_run(1);
```

```
    pthread_t usb_thread;
    pthread_create(&usb_thread, NULL, usb_input_loop, NULL);

    while (1) {
        swarm_wait_vsync();
        uint32_t st = swarm_read_status();

        if (STATUS_CAPTURE(st)) {
            swarm_trigger_sfx(SFX_CAPTURE);
            display_game_over(STATUS_STEP_COUNT(st));
            swarm_reset();
        }

        agent_state_t a[3];
        for (int i = 0; i < 3; i++) swarm_read_agent(i, &a[i]);
    }
    return 0;
}
```

8: File Formats

The software reads scenario map files from the SD card (mounted as part of the Linux root filesystem) to populate the grid with preset obstacle layouts. This enables the demo to cycle through interesting scenarios (open field, corridors, chokepoints) without requiring manual obstacle placement each time.

8.1 - Map File Format (.swm)

Map files use a simple human-readable text format with extension .swm. A file contains a header line followed by 64 lines of 64 characters each, representing the grid row-by-row (top to bottom). Each character encodes one cell type:

Character	Cell Type	Rendered As
`.`	CELL_FREE (0x0)	Light gray tile
`#`	CELL_WALL (0x1)	Dark gray tile
`P`	Pursuer 1 spawn	Cell set to FREE; agent 0 initialized here
`Q`	Pursuer 2 spawn	Cell set to FREE; agent 1 initialized here
`E`	Evader spawn	Cell set to FREE; agent 2 initialized here

Parsing rules:

(1) First line is a magic header "SWMAP v1 64x64\n". Reject the file if absent. (2) Exactly 64 subsequent lines, each exactly 64 characters plus newline. Shorter/longer rows cause a parse error. (3) Exactly one P, one Q, and one E must be present. (4) Any character other than {., #, P, Q, E} causes a parse error. (5) Lines beginning with ';' after the header are comments and ignored.

The parser (`swarm_load_map` in `swarm_driver.c`) walks the file character-by-character, calling `swarm_write_cell` for each cell via GRID_WR_ADDR/GRID_WR_DATA. Agent spawn

positions are written to hardware via a reset-with-spawn sequence. A full map load takes approximately 2 ms (4,096 register writes at ~500 ns each).

8.2 - Included Preset Maps

Four preset maps are shipped on the SD card and selectable at startup via command-line argument:

- `open.swm` — Empty 64×64 arena. Demonstrates pursuit in open space; flanking dynamics are clearest here.
- `corridor.swm` — Two parallel vertical corridors connected by horizontal openings. Tests AI behavior in constrained spaces.
- `maze.swm` — Dense random maze. Tests path-awareness despite lack of explicit path planning.
- `arena.swm` — Central obstacle cluster with open perimeter. Good for audience demos; chase is visually dynamic.