CPSC / EECS DESIGN DOCUMENT · APRIL 17, 2026

Go on DE1-SoC

Maximilian Comfere (mkc2182) · **Owen Cooper** (odc2106)

CONTENTS

1 Introduction

This document describes the design and implementation of a 9×9 Go game on the DE1-SoC FPGA board. Go is one of the oldest and most strategically deep board games in the world, yet its rules are elegantly simple: two players alternate placing black and white stones on a grid, competing to surround the most territory. The 9×9 board variant preserves the depth of full-size Go while being well-suited to the resource constraints of an embedded platform.

The system features a VGA-rendered graphical interface, USB keyboard input, audio feedback for stone placement and game events, and multiple AI opponents of varying difficulty. Players can choose between Player vs. Player (local) and Player vs. Computer modes, selecting from three bot difficulty levels ranging from a random-move beginner bot to a Monte Carlo tree search (MCTS) engine.

Design Decisions

The system partitions work between the Cyclone V FPGA fabric and the ARM Hard Processor System (HPS).

> **FPGA (hardware):** Handles VGA signal generation and audio playback because these require cycle-accurate timing that cannot be met by software running under Linux.

> **ARM HPS (software):** All rendering logic, game logic, and AI run on the HPS because they involve complex, variable-length computation (especially MCTS) that benefits from a general-purpose processor with access to DDR3 memory. The rule engine requires recursive flood-fill algorithms that map poorly to fixed hardware pipelines. Framebuffer rendering on the HPS is simpler to develop and debug than an on-the-fly hardware renderer, and the ARM Cortex-A9 at 800 MHz has ample throughput for 640×480 @ 60 Hz pixel writes.

Communication: The HPS writes control registers to the FPGA through memory-mapped I/O on the Avalon Lightweight HPS-to-FPGA bridge (base address `0xFF200000`). The FPGA reads pixel data from SDRAM via an Avalon master port. A separate SDRAM framebuffer is written by the HPS and read by the VGA controller.

2 System Block Diagram

The custom FPGA peripheral (`go_peripheral`) instantiates two sub-blocks: the VGA controller and the audio controller. Both share a single Avalon Lightweight HPS-to-FPGA slave interface for register access. The VGA controller additionally acts as an Avalon master to read framebuffer data from SDRAM.

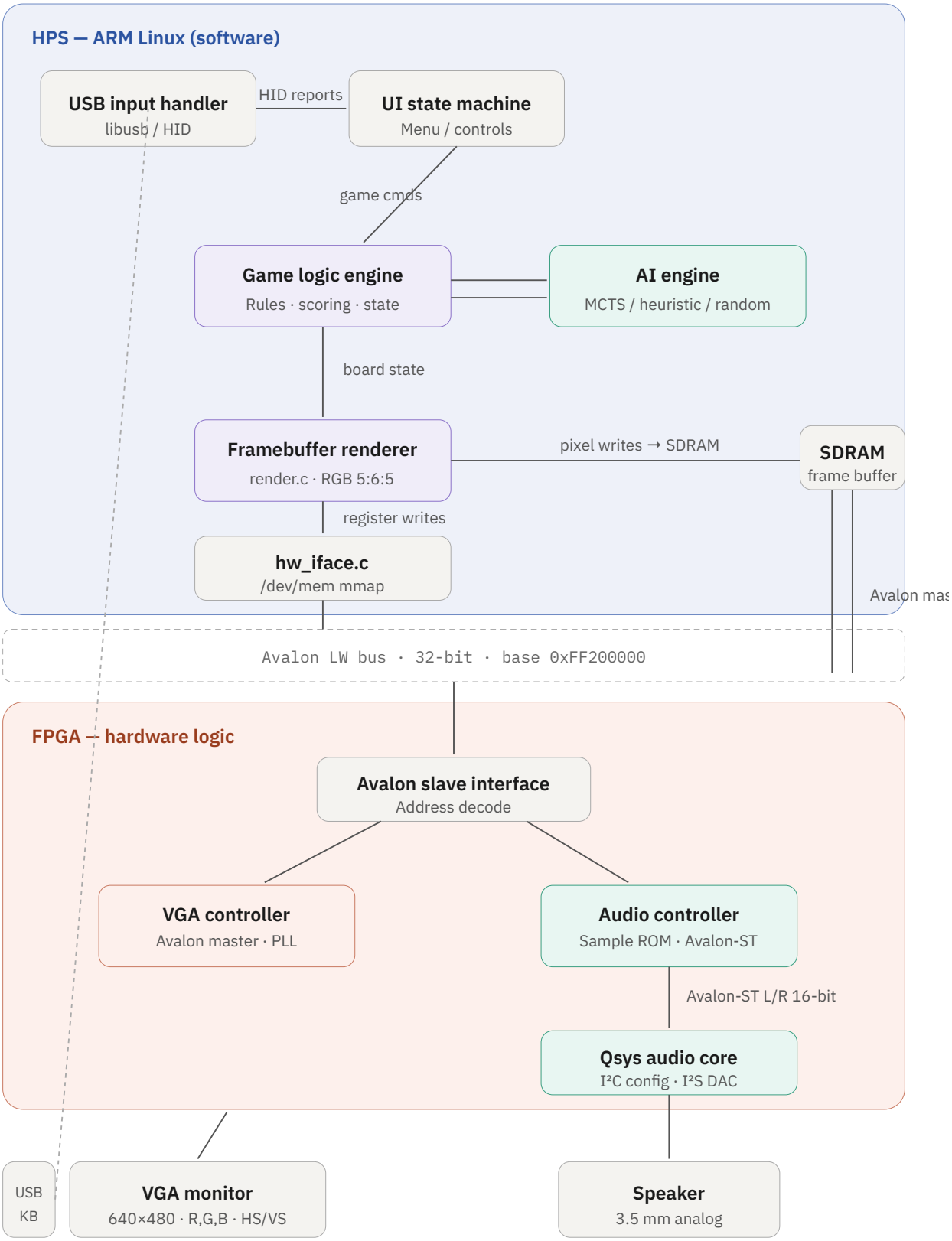
HPS — ARM Linux (software)
USB input handler
libusb / HID
HID reports
UI state machine
Menu / controls
game cmds
Game logic engine
Rules · scoring · state
AI engine
MCTS / heuristic / random
board state
Framebuffer renderer
render.c · RGB 5:6:5
pixel writes → SDRAM
SDRAM
frame buffer
register writes
hw_iface.c
/dev/mem mmap
Avalon mas
Avalon LW bus · 32-bit · base 0xFF200000
FPGA — hardware logic
Avalon slave interface
Address decode
VGA controller
Avalon master · PLL
Audio controller
Sample ROM · Avalon-ST
Avalon-ST L/R 16-bit
Qsys audio core
I²C config · I²S DAC
USB
KB
VGA monitor
640×480 · R,G,B · HS/VS
Speaker
3.5 mm analog

Figure 1 — DE1-SoC system block diagram. Blue = HPS/software domain; coral = FPGA/hardware domain.

Block Descriptions

USB Input Handler SW	Uses `libusb` to poll a USB keyboard via `interrupt_transfer`. Decodes HID key reports into game commands (cursor move, place stone, pass, menu navigate). Non-blocking design so the main game loop continues while waiting for input.
UI State Machine SW	Manages screen transitions: title screen, mode selection (PvP / PvC), difficulty selection (Level 1–3), active game, scoring screen. Reads commands from the input handler and dispatches to game logic or calls the appropriate render function.
Game Logic Engine SW	Maintains the authoritative board state as a `BoardState` struct with `Stone cells[9][9]`. Enforces all Go rules: validates move legality, counts liberties via flood fill, detects and executes captures, checks ko (via Zobrist hash) and suicide. After each valid move, calls the renderer to update the framebuffer.
AI Engine SW	Computes moves for Player vs. Computer mode. Three levels: Level 1 picks a random legal intersection; Level 2 uses heuristics (capture, defend atari, atari); Level 3 runs Monte Carlo tree search with 200 random playouts. All run on the ARM Cortex-A9 with access to DDR3 for MCTS node storage.
Framebuffer Renderer SW	`render.c` writes RGB 565 pixels directly into the SDRAM back buffer via `mmap`. Draws the Go board (wooden background, grid lines, star points), stones (filled circles), cursor highlight (unfilled ring), score panel, menus, and game-over screen. After rendering a frame, triggers an atomic buffer flip by writing `VGA_CTRL`.
hw_iface.c SW	Opens `/dev/mem`, `mmap`s the LW bridge span (`0xFF200000`, 2 MB). Provides `hw_read(offset)`, `hw_write(offset, value)`, and convenience wrappers for audio and VGA operations.
Avalon Slave Interface HW	Decodes register offsets and routes reads/writes to the VGA control registers and audio control registers.

VGA Controller HW	Generates 640×480 @ 60 Hz timing (25.175 MHz pixel clock via PLL). Reads pixel data from the SDRAM framebuffer via an Avalon master port, bursting 16 pixels per request into a 2×640-pixel ping-pong line buffer in BRAM. Outputs RGB 444 on VGA pins. Supports double-buffered display with vsync-synchronized buffer swaps.
Audio Controller HW	Contains ROM arrays loaded from `.vh` hex files at synthesis time via `$readmemh` (stone placement, capture, illegal move, and game-over sounds). When the `AUDIO_CMD` register is written by the HPS, the controller begins streaming 16-bit samples at 8 kHz through the Avalon-ST interface to the Qsys audio core. Asserts `AUDIO_STATUS[0]` (busy) until playback completes.
Qsys Audio Core + PLL HW	University Program audio core component and audio PLL (12.288 MHz). Handles WM8731 CODEC configuration via I²C at startup and I²S data transport. Receives audio samples from the audio controller via Avalon-ST streaming interface.

3 Algorithms

The core algorithms are the Go rule engine, the framebuffer renderer, the audio controller, and the AI engine. All game logic and rendering algorithms run on the ARM HPS in C. The audio playback algorithm runs in hardware on the FPGA.

3.1 Game Rule Engine (Software)

Every attempted placement calls `board_place()`, which enforces three constraints in order:

1. **Occupancy check:** the target intersection must be `EMPTY`.
2. **Capture and suicide check:** speculatively apply the move to a scratch board copy. Run a flood-fill liberty count on the placed group. If liberties == 0 and no opponent group was captured in the same move, the move is suicide — reject it.
3. **Ko check:** compare the resulting board hash (a 64-bit XOR Zobrist hash over all cells) against `prev_board_hash`. If they match, the move recreates the immediately previous position — reject it as a ko violation.

If all checks pass, the move is applied: opponent groups with zero liberties are removed and added to the capture count, the board hash is updated, and the turn is toggled. Passing is always legal and increments the consecutive-pass counter;

two consecutive passes end the game.

Liberty counting via flood fill:

```
function count_liberties(board[9][9], start_row, start_col):
    color = board[start_row][start_col]
    if color == EMPTY: return -1
    visited[9][9] = {false}
    liberties = 0
    queue = [(start_row, start_col)]
    while queue is not empty:
        (r, c) = dequeue(queue)
        if visited[r][c]: continue
        visited[r][c] = true
        for each (nr, nc) in neighbors(r, c):
            if out_of_bounds(nr, nc): continue
            if board[nr][nc] == EMPTY:
                liberties += 1
                visited[nr][nc] = true
            else if board[nr][nc] == color and not visited[nr][nc]:
                enqueue(queue, (nr, nc))
    return liberties, group
```

Capture detection after placing a stone at (row, col):

```
function process_captures(board[9][9], row, col):
    placed_color = board[row][col]
    opponent = opposite(placed_color)
    total_captured = 0
    for each (nr, nc) in neighbors(row, col):
        if board[nr][nc] == opponent:
            liberties, group = count_liberties(board, nr, nc)
            if liberties == 0:
                for each (gr, gc) in group:
                    board[gr][gc] = EMPTY
                total_captured += len(group)
    return total_captured
```

Move legality (integrating occupancy, suicide, ko):

```
function is_legal_move(board[9][9], row, col, color, previous_hash):
    if board[row][col] != EMPTY: return MOVE_ILLEGAL_OCCUPIED
    board[row][col] = color
    captures = process_captures(board, row, col)
    own_liberties, _ = count_liberties(board, row, col)
    if own_liberties == 0 and captures == 0:
        undo_move(board, row, col)
```

```
        return MOVE_ILLEGAL_SUICIDE
    if zobrist_hash(board) == previous_hash:
        undo_move(board, row, col)
        return MOVE_ILLEGAL_KO
    return MOVE_OK
```

Ko detection via Zobrist hashing: The `zobrist_table` is a precomputed array of 9×9×2 random 64-bit integers, initialized once at program start. Zobrist hashing provides O(1) incremental updates: when a stone is placed or removed, XOR the corresponding table entry into the running hash.

```
function zobrist_hash(board[9][9]):
    hash = 0
    for r in 0..8:
        for c in 0..8:
            if board[r][c] != EMPTY:
                hash ^= zobrist_table[r][c][board[r][c]]
    return hash
```

3.2 Scoring (Software)

At game end (both players pass consecutively), `board_score()` computes the score using Chinese-style area scoring: stones on the board + enclosed empty intersections. Territory is determined by flood-filling each empty region and checking which player's stones form the complete boundary. Contested intersections (bordered by both colors) are scored as neutral.

```
function compute_scores(board[9][9]):
    black_score = 0; white_score = 0
    // Count stones
    for r in 0..8:
        for c in 0..8:
            if board[r][c] == BLACK: black_score += 1
            if board[r][c] == WHITE: white_score += 1
    // Count territory: flood-fill each empty region
    for r in 0..8:
        for c in 0..8:
            if board[r][c] == EMPTY and not visited[r][c]:
                region, borders = flood_fill_empty(board, r, c, visited)
                if borders contains only BLACK: black_score += len(region)
                else if borders contains only WHITE: white_score += len(region)
    white_score += 5.5  // komi
    return black_score, white_score
```

3.3 Board Rendering Algorithm (Software)

The HPS renders the Go board and all UI elements by writing RGB 565 pixels directly into the SDRAM back buffer. After rendering is complete, the software triggers a buffer swap at the next vsync to achieve tear-free display.

Display layout (640×480): Board area is 384×384 pixels (8 gaps × 48 px per cell, 9 intersection lines), top-left corner at (128, 20). Each grid cell is 48×48 px. Intersections at `(128 + col*48, 20 + row*48)`. Stone radius: 20 px. Score/status panel at y = 420–470. 5×7 bitmap font stored in a C array (640 bytes for 128 ASCII characters).

```
#define CELL_SIZE    48
#define BOARD_LEFT   128
#define BOARD_TOP    20
#define STONE_RADIUS 20
#define COLOR_BG     0x2104   // dark gray
#define COLOR_BOARD  0xDEA0   // burlywood (RGB 5:6:5)
#define COLOR_BLACK  0x1082   // near-black
#define COLOR_WHITE  0xEF7D   // near-white
#define COLOR_CURSOR 0x07E0   // green

void render_board(const BoardState *b, int cursor_row, int cursor_col) {
    render_fill_rect(0, 0, 640, 480, COLOR_BG);
    render_fill_rect(BOARD_LEFT, BOARD_TOP, BOARD_SIZE, BOARD_SIZE, COLOR_BOARD);
    // 9 vertical + 9 horizontal grid lines
    for i in 0..8:
        render_fill_rect(BOARD_LEFT + i*CELL_SIZE, BOARD_TOP, 1, BOARD_SIZE, COLOR_LINE);
        render_fill_rect(BOARD_LEFT, BOARD_TOP + i*CELL_SIZE, BOARD_SIZE, 1, COLOR_LINE);
    // Star points (hoshi)
    for each (sr, sc) in [(2,2),(2,6),(6,2),(6,6),(4,4)]:
        render_circle(BOARD_LEFT + sc*CELL_SIZE, BOARD_TOP + sr*CELL_SIZE, 3, COLOR_LINE, filled=true);
    // Stones
    for r,c in 0..8:
        if b->cells[r][c] != EMPTY:
            render_circle(BOARD_LEFT+c*CELL_SIZE, BOARD_TOP+r*CELL_SIZE, STONE_RADIUS, color, filled=true)
    // Cursor ring
    render_circle(BOARD_LEFT+cursor_col*CELL_SIZE, BOARD_TOP+cursor_row*CELL_SIZE,
                  STONE_RADIUS+2, COLOR_CURSOR, filled=false);
}
```

Circle rendering uses the Midpoint (Bresenham) algorithm for efficient octant-symmetric pixel writes, supporting both filled and outline modes.

3.4 Audio Playback Algorithm (Hardware)

Sound effects are stored as 16-bit signed PCM samples in on-chip ROM, loaded from `.vh` hex files at synthesis time via `$readmemh`. The audio controller streams samples to the Qsys audio core through the Avalon-ST interface.

```
// ROMs loaded at synthesis
reg [15:0] place_sound[0:PLACE_LEN-1];
```

```
reg [15:0] capture_sound[0:CAPTURE_LEN-1];
reg [15:0] illegal_sound[0:ILLEGAL_LEN-1];
reg [15:0] gameover_sound[0:GAMEOVER_LEN-1];

localparam SAMPLE_PERIOD = 6250;  // 50 MHz / 8 kHz

always @(posedge clk) begin
    if (audio_cmd_write) begin
        active_sound <= audio_cmd_data[2:0];
        sample_index <= 0; sample_clock <= 0;
    end
    if (active_sound != 0) begin
        if (sample_clock >= SAMPLE_PERIOD) begin
            sample_clock <= 0;
            case (active_sound)
                3'd1: audio_sample <= place_sound[sample_index];
                3'd2: audio_sample <= capture_sound[sample_index];
                3'd3: audio_sample <= illegal_sound[sample_index];
                3'd4: audio_sample <= gameover_sound[sample_index];
            endcase
            if (sample_index >= current_sound_length)
                active_sound <= 0;
            else sample_index <= sample_index + 1;
        end else sample_clock <= sample_clock + 1;
    end
    // Avalon-ST output (mono: same on both channels)
    if (aso_left_ready)  aso_left_data  <= audio_sample;
    if (aso_right_ready) aso_right_data <= audio_sample;
end
```

Audio file preparation pipeline: Source files are converted with FFmpeg to 8 kHz, mono, 16-bit PCM WAV, then converted to `.vh` hex files via a Python script that writes one 4-digit hex value per line.

3.5 AI — Level 1: Random (Software)

Builds a list of all legal intersections (excluding suicide and ko), picks one uniformly at random. If no legal moves exist, passes. Time complexity: O(N²) for N=9.

```
function random_move(board, color, previous_hash):
    legal_moves = [all (r,c) where is_legal_move(...) == MOVE_OK]
    if legal_moves is empty: return PASS
    return random_choice(legal_moves)
```

3.6 AI — Level 2: Greedy Capture (Software)

Evaluates each legal move with a priority score. The highest-priority move is selected; ties broken randomly. Time

complexity: O(N^4) per turn.

P3	Move reduces an opponent group to 0 liberties. (+10 per captured stone)
P2	Move rescues a friendly group in atari (1 liberty remaining). (+8)
P1	Move places an opponent group in atari. (+2 per liberty reduced)
P0	Random legal move with slight center preference. (+0–3)

3.7 AI — Level 3: Monte Carlo Tree Search (Software)

Implements UCT-MCTS with random playouts. Each call to `ai_get_move()` runs 200 simulations from the current board position. A simulation selects a child node by UCB1 score, expands if unvisited, plays out randomly to depth 81, and back-propagates the win/loss result. UCB1: **wins/visits + C × √(ln(parent_visits)/visits)**, C = 1.41.

```
function mcts_move(board, color, num_simulations=200):
    root = new MCTSNode(board, color)
    for i in 1..num_simulations:
        node = root; state = copy(board)
        // Phase 1: Selection
        while node.untried_moves is empty and node.children not empty:
            node = best_ucb1_child(node); apply_move(state, node.move)
        // Phase 2: Expansion
        if node.untried_moves not empty:
            move = random_choice(node.untried_moves)
            apply_move(state, move)
            child = new MCTSNode(state, move)
            node.children.append(child); node = child
        // Phase 3: Simulation
        result = simulate_random_game(state)
        // Phase 4: Backpropagation
        while node is not null:
            node.visits += 1
            if result favors node's color: node.wins += 1
            node = node.parent
    return child of root with highest visits
```

The 9×9 board constrains the search space such that 200 simulations complete within approximately 1–2 seconds on the ARM Cortex-A9 at 800 MHz.

3.8 Sampling and Display

Audio samples are stored as raw 16-bit PCM, mono, 8,000 Hz, in on-chip BRAM (~34 KB total for four events). The VGA

framebuffer is 640×480 pixels × 2 bytes (RGB 565) = 614,400 bytes per buffer, double-buffered in SDRAM.

4 Hardware / Software Interface

The FPGA peripheral registers are accessible via the Avalon LW bridge, base address `0xFF200000`. All registers are 32-bit wide. Since USB keyboard input is handled entirely in software via `libusb`, the FPGA register set is minimal: audio control and VGA framebuffer management.

4.1 Register Map

OFFSET	NAME	R/W	BITS	DESCRIPTION
`0x00`	`AUDIO_CMD`	W	[2:0]	Trigger audio: 0=none, 1=place, 2=capture, 3=illegal, 4=game_over
`0x04`	`AUDIO_STATUS`	R	[0]	Audio busy (1 = sample currently playing)
`0x08`	`VGA_CTRL`	W	[0]	Write 1 to swap framebuffers at next vsync
`0x0C`	`VGA_STATUS`	R	[0]	In vertical blanking interval (1 = safe to swap)
`0x10`	`FB_ADDR_HI`	W	[7:0]	Upper 8 bits of framebuffer base address
`0x14`	`RESERVED`	—	—	Reserved for future use

4.2 Userspace Program

- **Keyboard input:** A dedicated input thread uses `libusb_interrupt_transfer` to read 8-byte HID reports. Arrow keys map to cursor movement, Enter places a stone, Space passes, Escape returns to menu. Decoded commands are written to a shared queue consumed by the main game loop.
- **Display:** `render_board()` writes RGB 565 pixels directly into the SDRAM back buffer via `mmap`. `hw_vga_swap()` triggers an atomic buffer flip at the next vsync by writing `VGA_CTRL[0] = 1`.
- **Audio:** `hw_play_audio(cmd)` writes to `AUDIO_CMD`. The hardware plays the corresponding sample from BRAM and asserts `AUDIO_STATUS[0]` until done.
- **AI turn:** When `mode == PVC` and `board.turn == AI_COLOR`, `ai_get_move()` is called instead of reading keyboard input.

4.3 Device Driver (Linux Userspace)

`hw_iface.c` opens `/dev/mem`, `mmap`s the LW bridge span (`0xFF200000`, 2 MB) and the SDRAM framebuffer region. Game state is held entirely in HPS memory (`BoardState` struct). No kernel module is needed.

`render.c` writes directly to the SDRAM framebuffer using a 5×7 bitmap font. Provides `render_board()`, `render_score_panel()`, `render_menu()`, `render_game_over()`, and primitive drawing functions.

4.4 FPGA Hardware Modules

- `go_peripheral` (top): Avalon slave, instantiates VGA and audio controllers, decodes register offsets 0x00–0x14, routes reads and writes.
- `vga_controller`: Generates 640×480 @ 60 Hz timing (25.175 MHz pixel clock via PLL), bursts 16 pixels/request from SDRAM into a 2×640-pixel line buffer (BRAM), outputs RGB 444 on VGA pins. Supports double-buffered framebuffers with vsync-synchronized swap.
- `audio_controller`: Stores 4 PCM sound effects in ROM (loaded from `.vh` files via `$readmemh`), streams samples at 8 kHz via Avalon-ST to the Qsys audio core. Asserts busy flag until playback completes.

4.5 FSM Controller

The Go game state machine runs entirely in software on the HPS. The main loop implements the states: `MENU → PLAYING (human turn) → PLAYING (AI turn) → GAME_OVER → MENU`. Transitions are driven by keyboard input and game-rule outcomes. The VGA and audio hardware are stateless peripherals triggered by register writes from this software FSM.

4.6 Write Sequence: Processing a Move

When the player places a stone at intersection (row, col), the HPS software performs:

```
1. Validate move (software-only)
   result = board_place(&board, row, col)
   if result != MOVE_OK:
       hw_play_audio(AUDIO_ILLEGAL); return

2. board_place() internally:
   - Sets board.cells[row][col] = current_color
   - Processes captures (removes groups with 0 liberties)
   - Updates board.captured_black / captured_white
   - Updates board.prev_board_hash (Zobrist)
   - Toggles board.turn

3. Render updated board to back buffer
   render_board(&board, cursor_row, cursor_col)
```

```
   render_score_panel(...)

4. Trigger audio
   if captures > 0: hw_play_audio(AUDIO_CAPTURE)
   else:            hw_play_audio(AUDIO_PLACE)

5. Swap framebuffers
   hw_vga_swap()   // writes VGA_CTRL[0] = 1
```

5 Resource Budgets

5.1 On-chip BRAM (FPGA Cyclone V)

BLOCK	SIZE	CONTENTS
VGA line buffer	2,560 B	Ping-pong (2 × 640 pixels × 16-bit)
Audio ROM — place	3,200 B	0.2 s, 8 kHz, 16-bit, 1600 samples
Audio ROM — capture	4,800 B	0.3 s, 8 kHz, 16-bit, 2400 samples
Audio ROM — illegal	2,400 B	0.15 s, 8 kHz, 16-bit, 1200 samples
Audio ROM — game over	24,000 B	1.5 s, 8 kHz, 16-bit, 12000 samples
Total on-chip	**~37 KB**	Well within Cyclone V M10K capacity (~500 KB)

5.2 SDRAM (HPS-side)

Framebuffer A (front)

614 KB

640×480 × 2 bytes (RGB 565)

Framebuffer B (back)

614 KB

Double buffer for tear-free display

Linux OS + userspace

~128 MB

MCTS tree (worst case)

~400 KB

Kernel, rootfs, application stack	~10K nodes × ~40 bytes each

5.3 HPS Software Memory

DATA	SIZE
`BoardState` struct	~200 bytes
Previous board hash (ko)	8 bytes
Zobrist table (9×9×2 × 8 bytes)	1,296 bytes
MCTS tree (200 simulations)	~400 KB
Bitmap font (128 chars × 5 bytes)	640 bytes
Total HPS memory	**< 1 MB**

5.4 Bandwidth

- **SDRAM framebuffer reads:** 640 × 480 × 2 bytes × 60 Hz ≈ 36.9 MB/s. The SDRAM controller provides ~1.6 GB/s peak bandwidth — framebuffer reads consume ~2.3%.
- **Framebuffer writes:** At 614 KB/frame × 60 fps ≈ 36 MB/s. Combined read+write ≈ 73 MB/s, well within SDRAM capacity.
- **Avalon LW bus:** Register writes are rare (a few per move + buffer swap per frame). Negligible bandwidth.

6 Verilog Module Interfaces

`go_peripheral` (top-level)

```
module go_peripheral (
    input  logic        clk,          // 50 MHz system clock
    input  logic        reset_n,      // Active-low reset
    // Avalon slave (register access from HPS)
    input  logic [4:0]  avs_address,
    input  logic        avs_read,
```

```
    input  logic        avs_write,
    input  logic [31:0] avs_writedata,
    output logic [31:0] avs_readdata,
    // VGA output
    output logic [3:0]  vga_r, vga_g, vga_b,
    output logic        vga_hs, vga_vs, vga_blank_n, vga_sync_n, vga_clk,
    // Avalon master (SDRAM framebuffer reads)
    output logic [31:0] avm_address,
    output logic        avm_read,
    input  logic [31:0] avm_readdata,
    input  logic        avm_waitrequest,
    // Avalon-ST audio source (left channel)
    output logic [15:0] aso_left_data,
    output logic        aso_left_valid,
    input  logic        aso_left_ready,
    // Avalon-ST audio source (right channel)
    output logic [15:0] aso_right_data,
    output logic        aso_right_valid,
    input  logic        aso_right_ready
);
    // Instantiates: vga_controller, audio_controller
endmodule
```

vga_controller

```
module vga_controller (
    input  logic        clk_50, reset_n,
    output logic [31:0] avm_address,
    output logic        avm_read,
    input  logic [31:0] avm_readdata,
    input  logic        avm_waitrequest,
    input  logic [31:0] fb_base_addr,
    input  logic        swap_request,
    output logic        vsync_flag,
    output logic [3:0]  vga_r, vga_g, vga_b,
    output logic        vga_hs, vga_vs, vga_blank_n, vga_sync_n, vga_clk
);
    // Instantiates: pll_25MHz (ALTPLL), line_buffer (2x640x16b BRAM)
endmodule
```

audio_controller

```
module audio_controller (
    input  logic        clk_50, reset_n,
    input  logic [2:0]  audio_cmd,
    output logic        busy,
```

```
    output logic [15:0] aso_left_data,
    output logic        aso_left_valid,
    input  logic        aso_left_ready,
    output logic [15:0] aso_right_data,
    output logic        aso_right_valid,
    input  logic        aso_right_ready
);
    // Instantiates: sample_rom (~34 KB total)
endmodule
```

7 C Software Interfaces

`board.h` — Board State & Rule Engine

```
typedef enum { EMPTY = 0, BLACK = 1, WHITE = 2 } Stone;
typedef enum { MOVE_OK, MOVE_ILLEGAL_SUICIDE,
               MOVE_ILLEGAL_KO, MOVE_ILLEGAL_OCCUPIED } MoveResult;

typedef struct {
    Stone    cells[9][9];
    Stone    turn;
    int      captured_black, captured_white;
    uint64_t prev_board_hash;
    int      consecutive_passes;
    int      game_over;
} BoardState;

void       board_init(BoardState *b);
MoveResult board_place(BoardState *b, int row, int col);
void       board_pass(BoardState *b);
int        board_score(const BoardState *b, int *black, int *white);
void       board_copy(BoardState *dst, const BoardState *src);
```

`ai.h` — AI Engine

```
typedef enum { AI_RANDOM = 1, AI_GREEDY = 2, AI_MCTS = 3 } AiLevel;
typedef struct { int row; int col; int pass; } Move;

#define MCTS_SIMULATIONS 200
#define MCTS_MAX_DEPTH   81

Move ai_get_move(const BoardState *b, AiLevel level);
```

`hw_iface.h` — Hardware Register Access

```
#define LW_BRIDGE_BASE   0xFF200000
#define REG_AUDIO_CMD    0x00
#define REG_AUDIO_STATUS 0x04
#define REG_VGA_CTRL     0x08
#define REG_VGA_STATUS   0x0C
#define REG_FB_ADDR_HI   0x10

#define AUDIO_NONE     0
#define AUDIO_PLACE    1
#define AUDIO_CAPTURE  2
#define AUDIO_ILLEGAL  3
#define AUDIO_GAME_OVER 4

int      hw_init(void);
void     hw_close(void);
uint32_t hw_read(uint32_t offset);
void     hw_write(uint32_t offset, uint32_t value);
void     hw_play_audio(uint8_t cmd);
void     hw_vga_swap(void);
```

`render.h` — Framebuffer Renderer

```
void render_init(void *fb_vaddr);
void render_board(const BoardState *b, int cursor_row, int cursor_col);
void render_score_panel(int black_score, int white_score,
                        int cap_black, int cap_white, Stone turn);
void render_game_over(int black_score, int white_score);
void render_menu(int selected_item);
void render_pixel(int x, int y, uint16_t color);
void render_fill_rect(int x, int y, int w, int h, uint16_t color);
void render_circle(int cx, int cy, int r, uint16_t color, int filled);
void render_string(int x, int y, const char *s, uint16_t color);
```

8 Game Rules and Mechanics

- **Stone Placement:** Black and white alternate placing stones on the intersections of a 9×9 grid. Black moves

first.

- **Liberties and Capture:** A stone or connected group is captured when all adjacent empty intersections (liberties) are occupied by the opponent.
- **Ko Rule:** A move that would recreate the immediately previous board position is forbidden. Detected via Zobrist hash comparison.
- **Passing and End of Game:** A player may pass their turn. The game ends when both players pass consecutively.
- **Scoring:** Chinese-style area scoring (stones on board + surrounded empty intersections). White receives 5.5 komi. Final score displayed at game end.
- **Suicide Rule:** A move that would leave the player's own group with zero liberties, without capturing an opponent group, is illegal.

9 Game Modes

MODE	DESCRIPTION
Player vs. Player	Two players share a USB keyboard to take turns. A cursor navigates the board; Enter places a stone, Space passes.
PvC — Level 1 (Random)	Places stones on random legal intersections. Suitable for learning the rules.
PvC — Level 2 (Greedy)	Prioritizes capturing opponent stones and defending groups in atari using simple heuristics.
PvC — Level 3 (MCTS)	Uses UCT-MCTS with 200 random playouts to evaluate moves. Provides a legitimately challenging opponent on a 9×9 board.

10 CODEC Interface

The WM8731 audio CODEC on the DE1-SoC is configured via the Qsys audio core at system startup. The Qsys audio core's built-in I²C controller writes a fixed configuration sequence to 10 WM8731 registers:

REGISTER	ADDRESS (7-BIT)	VALUE	FUNCTION
Left input	0x00	0x017	Line in mute, 0 dB
Right input	0x01	0x017	Line in mute, 0 dB
Left HP out	0x02	0x079	Headphone volume
Right HP out	0x03	0x079	Headphone volume
Analog path	0x04	0x010	DAC select, mic mute
Digital path	0x05	0x000	No de-emphasis, no mute
Power down	0x06	0x000	All blocks powered on
Format	0x07	0x001	I2S, 16-bit, slave mode
Sampling	0x08	0x002	Normal mode, 8 kHz
Active	0x09	0x001	Activate interface

After the configuration sequence completes, the audio core begins accepting sample data via its Avalon-ST sink interface. The `audio_controller` module pushes 16-bit mono samples at 8 kHz; the Qsys audio core handles I²S serialization and physical pin driving.

A Appendix: Verilog Module Hierarchy

```
go_peripheral  (top, Avalon slave + master)
 ├─ vga_controller  (Avalon master for SDRAM reads)
 │    ├─ pll_25MHz     (ALTPLL megafunction)
 │    └─ line_buffer   (2×640×16-bit BRAM ping-pong)
 └─ audio_controller  (Avalon-ST source to Qsys audio core)
      └─ sample_rom   (~34 KB ROM from .vh files)

Qsys components  (external to go_peripheral)
 ├─ audio_0       (University Program audio core)
 └─ audio_pll_0   (12.288 MHz audio clock PLL)
```

B Appendix: HID Keycode Map

HID KEYCODE (HEX)	KEY	GAME ACTION
0x52	↑ Up Arrow	Move cursor up
0x51	↓ Down Arrow	Move cursor down
0x50	← Left Arrow	Move cursor left
0x4F	→ Right Arrow	Move cursor right
0x28	Enter	Place stone at cursor
0x2C	Space	Pass turn
0x29	Escape	Return to main menu